SPERRY

# XENIX® Development System

# Programmer's Guide

Personal Computer Library
UP-11484

This document was typeset with an IMAGEN® 8/300 Laser Printer.

Document Number: G-2-14-85-1.3/1.0

# Contents

# Chapter 1
# Introduction

## 1.1 Overview

This guide explains how to use the XENIX Development System to create and maintain C language and assembly language programs. The system provides a broad spectrum of programs and commands to help you design and develop applications and system software. These programs and commands enable you to create C and assembly language programs for execution on the XENIX system. They also let you debug these programs, automate their creation, and maintain different versions of the programs you develop.

The following sections introduce the programs and commands of the XENIX Development System, and explain the steps you can take to develop programs for the XENIX system. Most of the programs and commands in these introductory sections are fully explained later in this guide. Some commands mentioned here are part of the XENIX Operating System. These are explained in the XENIX *User's Guide* and XENIX *Operations Guide*.

## 1.2 Creating C Language Programs

All C language programs start as a collection of C program statements in a source file. The XENIX system provides a number of text editors that let you create source files easily and efficiently. The most convenient editor is the screen-oriented editor **vi**. **Vi** provides many editing commands that let you easily insert, replace, move, and search for text. All commands can be invoked from command keys or from a command line. **Vi** also has a variety of options that let you modify its operation.

Once a C language program has been written to a source file, you can create an executable program by using the **cc** command. The **cc** command invokes the XENIX C compiler which compiles the source file. This command also invokes other XENIX programs to prepare the compiled program for execution.

You can debug an executable C program with the XENIX debugger **adb**. **Adb** provides a direct interface to the machine instructions that make up an executable program.

If you wish to check a program before compiling it, you can use **lint**, the XENIX C program checker. **Lint** checks the content and construction of C language programs for syntactical and logical errors. It also enforces a strict set of guidelines for proper C programming style. **Lint** is normally used in the early stages of program development to check for illegal and improper usage of the C language.

Another way to check a program is with **cb**, the XENIX C program beautifier. **Cb** improves readability of C programs, making detection of logical errors easier.

## 1.3 Creating Other Programs

The C programming language can meet the needs of most programming projects. In cases where finer control of execution is required, you may create assembly language programs using the XENIX assembler **as**. **As** assembles source files and produces relocatable object files that can be linked to your C language programs with **ld**. The **ld** program is the XENIX linker. It links relocatable object files created by the C compiler or assembler to produce executable programs. Note that the **cc** command automatically invokes the linker and the assembler, so use of either **as** or **ld** is optional.

You can create source files for lexical analyzers and parsers using the program generators **lex** and **yacc**. Lexical analyzers are used in programs to pick patterns out of complex input and convert these patterns into meaningful values or tokens. Parsers are used in programs to convert meaningful sequences of tokens and values into actions. The lex program is the XENIX lexical analyzer generator. It generates lexical analyzers, written in C program statements, from given specification files. The **yacc** program is the XENIX parser generator. It generates parsers, written in C program statements, from given specification files. **Lex** and **yacc** are often used together to make complete programs.

You can preprocess C and assembly language source files, or even **lex** and **yacc** source files using the **m4** macro processor. The **m4** program performs several preprocessing functions, such as converting macros to their defined values and including the contents of files into a source file.

## 1.4 Creating and Maintaining Libraries

You can create libraries of useful C and assembly language functions and programs using the **ar** and **ranlib** programs. **Ar**, the XENIX archiver, can be used to create libraries of relocatable object files. **Ranlib**, the XENIX random library generator, converts archive libraries to random libraries and places a table of contents at the front of each library

The **lorder** command finds the ordering relation in an object library. The **tsort** command topologically sorts object libraries so that dependencies are apparent.

## 1.5 Maintaining Program Source Files

You can automate the creation of executable programs from C and assembly language source files and maintain your source files using the **make** program and the SCCS commands.

The **make** program is the XENIX program maintainer. It automates the steps required to create executable programs, and provides a mechanism for ensuring up-to-date programs. It is used with medium-scale programming projects.

The Source Code Control (SCCS) commands let you maintain different versions of a single program. The commands compress all versions of a source file into a single file containing a list of differences. These commands also restore compressed files to their original size and content.

Many XENIX commands let you carefully examine a program's source files. The **ctags** command creates a tags file so that C functions can be quickly found in a set of related C source files. The **mkstr** command creates an error message file by examining a C source file.

Other commands let you examine object and executable binary files. The **nm** command prints the list of symbol names in a program. The **hd** command performs a hexadecimal dump of given files, printing files in a variety of formats, one of which is hexadecimal. The **size** command reports the size of an object file. The **strings** command finds and prints readable text (strings) in an object or other binary file. The **strip** command removes symbols and relocation bits from executable files. The **sum** command computes a checksum value for a file and a count of its blocks. It is used in looking for bad spots in a file and for verifying transmission of data between systems. The **xstr** command extracts strings from C programs to implement shared strings.

## 1.6 Creating Programs With Shell Commands

In some cases, it is easier to write a program as a series of XENIX shell commands than it is to create a C language program. Shell commands provide much of the same control capability as the C language, and give direct access to all the commands and programs normally available to the XENIX user.

The **csh** command invokes the C-shell, a XENIX command interpreter. The C-shell interprets and executes commands taken from the keyboard or from a command file. It has a C-like syntax which makes programming in this command language easy. It also has an aliasing facility, and a command history mechanism.

## 1.7 Using This Guide

This guide is intended for programmers who are familiar with the C programming language and with the XENIX system.

Chapter 1 introduces the XENIX software development programs provided with this package.

Chapter 2 explains how to compile C language programs using the **cc** command.

Chapter 3 explains how to check C language programs for syntactical and semantical correctness using the C program checker **lint**.

Chapter 4 explains how to automate the development of a program or other project using the **make** program.

Chapter 5 explains how to control and maintain all versions of a project's source files using the SCCS commands.

Chapter 6 explains how to debug C and assembly language programs using the XENIX debugger **adb**.

Chapter 7 explains how to assemble assembly language programs using the XENIX assembler **as**.

Chapter 8 explains how to create lexical analyzers using the program generator **lex**.

Chapter 9 explains how to create parsers using the program generator **yacc**.

Chapter 10 explains how to use XENIX as a cross-development environment to create DOS programs.

Chapter 11 explains how to write device drivers.

Chapter 12 includes sample device drivers, and explains the syntax and logic used.

Appendix A explains how to write C language programs that can be compiled on other XENIX systems.

Appendix B explains how to use to create and process macros using the **m4** macro processor.

Appendix C discusses library routines available for XENIX and DOS cross development.

Appendix D explains compiler, assembler and linker error messages.

C language programmers should read Chapters 2, 3, and 6 for an explanation of how to compile and debug C language programs.

Assembly language programmers should read Chapter 7 for an explanation of the XENIX assembler and Chapter 6 for an explanation of how to debug programs.

Programmers who wish to automate the compilation process of their programs should read Chapter 4 for an explanation of the **make** program. Programmers who wish to organize and maintain multiple versions of their programs should read Chapter 5 for an explanation of the Source Code Control System (SCCS) commands.

Special project programmers who need a convenient way to produce lexical analyzers and parsers should read Chapters 8 and 9 for explanations of the **lex** and **yacc** program generators.

XENIX programmers who want to write programs executable under DOS should read Chapter 10 and Appendix C to learn to use **cc**, **dosld**, and the XENIX-DOS common libraries for DOS compilation.

## 1.8 Notational Conventions

This guide uses a number of special symbols to describe the syntax of XENIX commands. The following is a list of these symbols and their meaning.

| | |
|---|---|
| [ ] | Brackets indicate an optional command argument. |
| ... | Ellipses indicate that the preceding argument may be repeated one or more times. |
| SMALL | Small capitals indicate a key to be pressed. |
| **bold** | Boldface characters indicate a command or program name. |
| *italics* | Italic characters indicate a placeholder for a command argument. When typing a command, a placeholder must be replaced with an appropriate filename, number, or option. |

# Chapter 2
# Cc: A C Compiler

## 2.1 Introduction

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. For a description of the C language, see the XENIX *Microsoft C Reference Manual*.

## 2.2 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 filename, if desired, but must separate each item with one or more spaces.

The **cc** command options let you 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 let you 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 *cc*(CP) in the XENIX *Reference Manual*.

## 2.3 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 filename 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 *Programmer's Reference*.

### 2.3.1 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".

### 2.3.2 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 that are called by this "main" function or by other functions in the program. The files must not contain calls to functions that are 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 basename 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.

## 2.3.3 Naming the Output File

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

–o *filename*

where *filename* is a valid filename or pathname. If a filename is given, the program file is stored in the current directory. If a full pathname 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.

## 2.4 Creating Small, Middle, and Large Programs

The **cc** command lets you create 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 that are 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 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 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 which 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 several bytes of 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 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.

### 2.4.1 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.

### 2.4.2 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.

### 2.4.3 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.

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. This means the **-i** option, used with pure-text small model programs, is not required for middle model programs.

### 2.4.4 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.

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

## 2.5 Using Object Files and Libraries

The **cc** command lets you 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 also lets you 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 section CP of the XENIX *Reference Manual*, 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.

### 2.5.1 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 filename 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 object files 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.

## 2.5.2 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, then linked with *add.o* and *mult.o* to create the executable file *a.out*.

## 2.5.3 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 filename (see *Intro*(S) in the XENIX *Reference Manual* 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 *Programmer's Reference*.

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.

## 2.6 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.

### 2.6.1 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 *main.c* and *add.c*.

You may 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.

### 2.6.2 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**(CP). See the XENIX *Reference Manual* for details.

### 2.6.3 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 option, makes the smallest possible program, it should be used with great care. Removing stack probes from a program whose stack use is not well known can cause execution errors.

## 2.7 Preparing Programs for Debugging

The **cc** command provides a variety of options to prepare a program that is under development 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.

### 2.7.1 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. 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 your 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** commands suppress subsequent compilation of the source file; they imply the **-c** option. Thus, no program file is created and no linking is performed.

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 option in the command line must appear before the names of the files for which the assembly listing is to be saved.

---

*Note*

The assembly language files created by the **-S** and **-L** options are not suitable as input to the XENIX assembler **as**.

---

### 2.7.2 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 and *monitor* function are described in detail in *prof*(CP) and *monitor*(S) in the XENIX *Reference Manual.*

The **-p** option must be given in any command line that references object files that contain 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.

## 2.8 Controlling the C Preprocessor

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

### 2.8.1 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 option lets you assign 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 does not contain an 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" using the –D option. 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 add.c
```

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

### 2.8.2 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

–I*directoryname*

where *directoryname* is a valid pathname 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.

### 2.8.3 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*.

### 2.8.4 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 **#line** 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.

## 2.9 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. For a complete list of error messages, see Appendix D, "Compiler, Assembler, and Linker Messages"

### 2.9.1 C Compiler Messages

The C compiler displays messages about syntactical 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 which may affect execution of the program. For a description of **lint**, see Chapter 3, "Lint: A C Program Checker"

### 2.9.2 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 which 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

| Level | Warning |
|---|---|
| 0 | Suppresses all warning messages. Only messages about actual syntactical or semantic errors are displayed. |
| 1 | Warns about potentially missing statements, non-reachable 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 warning messages. The command

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

produces no warning messages. Note that the **-w** option has the same effect as **-W0.**

## 2.10 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.

### 2.10.1 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: An Assembler," and must have the extension ".s"

When assembly language source files are given, the **cc** command invokes the XENIX assembler, **as**, 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 assemble language source file *add.s*. The resulting object files, *main.o* and *add.o*, are linked to form a single program.

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 which fail to provide this interface will cause errors. See Appendix A, "Assembly Language Interface," in the XENIX *Programmer's Reference*.

### 2.10.2 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** lets you access data and functions in segments outside of the program. In middle and large model programs, **near** lets you access data with just an offset.

The examples in the following table illustrate the **far** and **near** keywords as used in declarations in a small model program.

**Uses of near and far Keywords**

| 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)[1] |
| char far * far s; | far (32 bits) | 32 bits (far pointer)[2] |
| int foo(); | near (16 bits) | function returning 16 bits |
| int far foo(); | far (32 bits) | function returning 16 bits[3] |

Notes:

[1] This example has no meaning; it is shown for syntactic completeness only.

[2] This is similar to accessing data in a long model program.

[3] 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.

### 2.10.3 Changing Word Order in Programs

The Microsoft C compiler automatically uses the standard 8086/286 word order for **long** type values (the **-M2** option). This order may cause problems when reading data files from programs created by other C compilers. You can change the word order for a given program by using the **-Mb0** 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. Refer to the XENIX *Development System Release Notes* to see if this is the option to use for 8086 code generation.

Note that there are other portability issues which must be considered when creating C programs intended for several different XENIX systems. For an explanation of these issues, see Appendix B, "C Language Portability," in this guide.

### 2.10.4 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 hexadecimal 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 is set (usually either a fixed stack of 2K bytes or variable stack). Refer to the **machine**(M) page in the XENIX *Reference Manual* for the default stack used with a specific machine.

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.

### 2.10.5 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" given 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.)

For a complete description of the **–dos** option and the cross development tools available under XENIX, see Chapter 10, "XENIX to MS-DOS: A Cross Development System"

## 2.11 Compiler Summary

The following sections summarize cc options and memory models.

### 2.11.1 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.

–dos Makes DOS executable files. Uses #include files in */usr/include*dos. Uses libraries in */usr/lib/dos*. Uses linker in */usr/bin/dosld*.

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

–F num
Sets the size of the program stack.

–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
Search 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. The *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" are mutually exclusive.

–nl num
Sets the maximum length of external symbols.

–ND name
Sets the data segment name.

–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.

### 2.11.2 Memory Models

The following table defines the number of text and data segments for the four different program memory models. This table also lists the segment register values.

| 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 module programs, text and data occupy the same segment. In pure-text programs, they occupy different segments.

### 2.11.3 Pointer and Integer Sizes

The following table defines the sizes (in bits) of integers (**int** type), and text and data pointers, in each program memory model.

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

### 2.11.4 Segment and Module Names

The following table lists the default text and data segment names, and the default module name, for each object file.

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

# Chapter 3
# Lint: A C Program Checker

## 3.1 Introduction

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 compiles C language source files, it does not perform the sophisticated type and error checking required by many programs, though syntax is gone over. The *lint* program, provides additional checking of source files without compiling.

## 3.2 Invoking *lint*

You can invoke *lint* program by typing

```
lint [ option ] ... filename ... lib ...
```

where *option* is a command option that defines how the checker should operate, *filename* is the name of the C language source file to be checked, and *lib* is the name of a library to check. You can give more than one option, filename, or library name in the command. If you give two or more filenames, *lint* assumes that the files belong to the same 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 three of the source file *main.c*, is not used anywhere in the file.

## 3.3 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 in 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 on 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 larger, 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 accidentally have 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 declaration 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 that are intended to be used in other source files and are not explicitly used within the file. Since *lint* can only check the given file, it assumes that such variables or functions are unused and reports them as such.

## 3.4 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

```
char 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.

## 3.5 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 case constructions or loops containing break and continue statements.

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.

## 3.6 Checking for Infinite Loops

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

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

and

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

are both considered infinite loops. While the statements

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

or

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

are never executed.

It is relatively common for valid programs to have such loops, but they are generally considered errors.

## 3.7 Checking Function Return Values

The *lint* program checks that a function returns a meaningful value if necessary. Some functions return values which 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, then *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 some of the noise messages produced by *lint*.

## 3.8 Checking for Unused Return Values

The *lint* program checks for cases where a function returns a value, but the value is usually ignored. *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.

## 3.9 Checking Types

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

1. Across certain binary operators and implied assignments

2. At the structure selection operators

3. Between the definition and uses of functions

4. In the use of enumerations

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

## 3.10 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.

## 3.11 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**.

## 3.12 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.

## 3.13 Checking for Strange Constructions

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

```
*p++ ;
```

the star (*) 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

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

is equivalent to

```
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 ) ...
```

or

```
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 that are 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.

## 3.14 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.

## 3.15 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.

## 3.16 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 up, function arguments will probably be best evaluated from right to left; on machines with a stack running down, 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.

In order that the efficiency of C on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the 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
```

## 3.17 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* flags. 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, it is desirable 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.

## 3.18 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 like ordinary source files. The only difference is that functions that are defined in a library file, but are not used in a source file, draw no comments. *Lint* does not simulate a full library search algorithm, and checks 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 programs that are normally loaded when a C program is run. When the -p option is in effect, the portable library 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.

Lint library files are named "/usr/lib/ll*". The programmer may wish to examine the lint libraries directly to see what lint thinks a function should passed and return. Printed out, lint libraries also make satisfactory skeleton quick-reference cards.

# Chapter 4
# Make: A Program Maintainer

## 4.1 Introduction

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 verifies 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. Make updates a 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*(CP) in the XENIX *Reference Manual*.

## 4.2 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 filename of the file to be updated, *dependent* is the filename of the 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 filename or dependent filename if desired. Each filename must be separated from the next by at least one space. The target filenames must be separated from the dependent filenames by a colon (:). Filenames must be spelled as defined by the XENIX system. Shell metacharacters, such as star (*) and question mark (?), can also be used.

You may give a sequence of commands on the same line as the target and dependent filenames, 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*(C), must appear on single lines; they must not contain the backslash (\) and newline character combination.

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 place 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 (\) and a newline character.

The makefile should be kept in the same directory as the given source files. For convenience, the filenames **makefile**, **Makefile**, *s.makefile*, and *s.Makefile* are provided as default filenames. These names are 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 filename begins with the *s.* prefix, **make** assumes that it is an SCCS file and invokes the appropriate SCCS command to retrieve the lastest 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 which defines how to update the given target file.

## 4.3 Invoking Make

Once you have a makefile and wish to update and modify one or more target files in the file, 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 filename of the file to be updated. It 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. In this case, make searches for the files makefile, Makefile, *s.makefile*, and *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 current date of the *prog* program with the current date each of the object files *x.o*, *y.o*, and *z.o*. It recreates *prog* if any changes have been made to any object file since *prog* was last created. It also compares the current dates of the object files with the dates of the four source files *x.c*, *y.c*, *z.c*, or *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 have been altered since the last time *prog* was created, make announces this fact and stops. No files are changed.

You can direct make to update a given target file by giving the filename 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 pathname 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 *filename*. **Make** reads the standard input until the end-of-file character is encountered.

You may use the 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** 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 of this, 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.

## 4.4 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 either 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.

## 4.5 Using Macros

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

A macro definition is a line containing a name, an equal sign (=), and a value. The equal sign must not be 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 macro name with 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 a macro for the names of object files to be link and one for the names 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 load the three object files with the *lex* library specified with the **-lln** option.

You may include a macro definition in a command line. A macro definition in a command line has the same form as a macro definition in a makefile. If spaces are to be used in the definition, double quotation marks must be used to enclose the definition. Macros in a command line override corresponding definitions found in the makefile. For example, the command

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

loads assigns the library options **-lln** and **-lm** to "LIBES".

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. |
| **$@** | Contains the full pathname of the current target. It may be used in dependency lines with user-defined target names. |
| **$<** | Contains the filename 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. |
| **$?** | Contains the filenames of the dependents that are more recent than the given target. It may be used in dependency lines with user-defined target names. |
| **$%** | Contains the filename 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** descriptor must not be used with the **$?** macro.

## 4.6 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 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" variables, 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** options 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
```

## 4.7 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 filename as the target or dependent instead of the filename 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 for a given file is given in a makefile, **make** automatically checks the default dependents of the file. It then forms 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, then 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, it 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 objects 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.

## 4.8 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*) file 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 :
          cc68 $< -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*.

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.

## 4.9 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".

## 4.10 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.

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 recommended only as a last resort.

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.

## 4.11 Using Make: An Example

As an example of the use of **make**, examine the makefile, given in Figure 4-1, used to maintain the **make** itself. The code for **make** is spread over a number of C source files and a *yacc* grammar.

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

```
     make
```

in a directory containing only the source and 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 were 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.

**Figure 4-1. Makefile Contents**

```
# 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
#<TAB>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)
```

# Chapter 5 SCCS: A Source Code Control System

## 5.1 Introduction

The Source Code Control System (SCCS) is a collection of XENIX commands that create, maintain, and co.trol special files called SCCS files. The SCCS commands let you create and store multiple versions of a program or document in a single file, instead of one file for each version. The commands let you 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. The SCCS system 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, and other documentation.

This chapter explains how to make SCCS files, how to update the files contained in SCCS files, and how to maintain the SCCS files once they are created. The following sections describe the basic information you need to start using the SCCS commands. Later sections describe the commands in detail.

## 5.2 Basic Information

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

— Files and directories

— Deltas and SIDs

— SCCS working files

— SCCS command arguments

— File administration

### 5.2.1 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 s-files only, and should have read and examine permission for everyone, and write permission for the user only.

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

### 5.2.2 Deltas and SIDs

Unlike an ordinary text file, an SCCS file (or s-file for short) contains nothing more than lists of changes. Each list corresponds to the changes needed to construct exactly one version of the file. The lists can then be 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, the SCCS system 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.

The SCCS system 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.

### 5.2.3 SCCS Working Files

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

| | |
|---|---|
| s-file | A permanent file that 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 file prefix *s.* at the beginning of the original filename. |
| x-file | A temporary copy of the s-file. It is created by SCCS commands which change the s-file. It is used instead of the s-file to carry out the changes. When all changes are complete, the SCCS system removes the original s-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. |
| 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 filename 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 p-file exists until all currently retrieved files have been saved in the s-file; it is then deleted. 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 filename. |
| 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 modifes an SCCS file, it creates a z-file and copies its own process ID to it. Any other command which 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 filename. |
| 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 filename. |

| | |
|---|---|
| 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 p-file. The 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 delete these files to ensure proper operation of subsequent SCCS commands.

### 5.2.4 SCCS Command Arguments

Almost all SCCS commands accept two types of arguments: options and filenames. 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 filename indicates the file to be acted on. The syntax for SCCS filenames is like other XENIX filename syntax. Appropriate pathnames 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 and unreadable files, these are ignored. A filename must not begin with a minus sign (-).

The special symbol – may be used to cause the given command to read a list of filenames from the standard input. These filenames are then used as names for the files to be processed. The list must terminate with an end-of-file character.

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

Filenames 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.

### 5.2.5 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 it. Before other users can access the file, the administrator must ensure that they have adequate access. Several SCCS commands let the administrator define who has access to the versions in a given s-file. These are described later.

## 5.3 Creating and Using S-files

The s-file is the key element in the SCCS system. 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.

### 5.3.1 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, since 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 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.

## 5.3.2 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 which 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*. When giving multiple s-file names in a command, you must separate each with at least one space. When the get command displays information about the files, it places the corresponding filename before the relevent information.

### 5.3.3 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 filename if you wish. If you do, you must separate each name with a space.

The command retrieves the lastest 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 named *demo.c*. You may edit the file just as you do any other text file.

If you give more than one filename, 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 filename 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 a *p.*. The user must not access the p-file directly.

## 5.3.4 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 0, 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.

## 5.3.5 Retrieving a Specific Version

You can retrieve any version you wish from an s-file by using the **-r** (for "retrieve") 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 filename. The names must be separated with spaces.

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 filenames 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 release.

## 5.3.6 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, -r*rel-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 filename.

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.

## 5.3.7 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 whose SID 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.

## 5.3.8 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.

### 5.3.9 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.

### 5.3.10 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. This option also directs all output normally sent to the standard output, such as the SID of the retrieved file, to the standard error file. Thus, 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 is copied to the standard error file.

### 5.3.11 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*, then saves a copy of this version in the file *demo.c*. You may display the file as desired, but you cannot edit the file.

### 5.3.12 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 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

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

The help command can be used at any time.

## 5.4 Using Identification Keywords

The SCCS system provides several special symbols, called identification keywords, which may be used in the text of a program or document to represent a predefined value. Keywords represent a wide range of values, 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, the SCCS system automatically replaces any keywords 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 the section *get*(CP) in the XENIX *Reference Manual.*

### 5.4.1 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 (%). No special characters are required. For example, "%I%" is the keyword representing the SID of the current version, and "%H%" is the keyword representing the current date.

When the program 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 data in a 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. The system assumes that you wish 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.

## 5.4.2 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", you must 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, the SCCS system 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 section *get*(CP) in the XENIX *Reference Manual*. You can change keyword values at any time.

## 5.4.3 Forcing Keywords

If a version is found to contain no keywords, you can force a fatal error 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. 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*. If the given version does not contain keywords, subsequent **delta** or **admin** commands that access this file print an error message.

Note that if you attempt to set the i flag at the same time as 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.

## 5.5 Using S-file Flags

An s-file flag is a special value that defines how a given SCCS command 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 operations such as 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 section *admin*(CP) in the XENIX *Reference Manual.*

### 5.5.1 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".

### 5.5.2 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 version of a file, which 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.

### 5.5.3 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 which does not use the −r option will retrieve version 1.1.

### 5.5.4 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 means of indicating 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.

### 5.5.5 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 −d*flag* *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 which 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 and −i options must not be used at the same time.

## 5.6 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.

### 5.6.1 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 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.

### 5.6.2 Changing Comments

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

cdc -r*SID*  *s.filename*

where -r*SID* gives the SID of the version whose comment is 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 they are 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.

## 5.6.3 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 which 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 you give more than one request, you must separate them with spaces and 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.

## 5.6.4 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".

### 5.6.5 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** 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 to initialize the description field when creating the s-file. 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 –t 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 filename. For example, the command

```
admin  -t  s.demo.c
```

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

## 5.7 Printing from an S-file

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

### 5.7.1 Using a Data Specification

You can explicitly define the information to be printed from an s-file 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 -dspec  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: represent the filename of the given s-file, :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 section *prs*(CP) in the XENIX *Reference Manual.*

### 5.7.2 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.

### 5.7.3 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 which 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 which succeed version 1.4 (e.g., 1.5, 1.6, and 2.1).

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

## 5.8 Editing by Several Users

The SCCS system allows any number users to access and edit versions of a given s-file. Since users are likely to access different versions of the s-file at the same time, the system is designed to allow concurrent editing of different versions. Normally, the system allows only one user at a time to edit a given version, but you can 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.

### 5.8.1 Editing Different Versions

The SCCS system 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 edit version 1.1. There is no limit to the number of versions which may be edited at any given time.

When several users edits 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.

### 5.8.2 Editing a Single Version

You can let a single version of a file be edited by more than one user by setting the j flag in the given 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, the **get** command 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 another user retrieves version 1.4 for editing before the first user has saved his changes, the 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.

### 5.8.3 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 name of 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.

## 5.9 Protecting S-files

The SCCS system uses the normal XENIX system file permissions to protect s-files from changes by unauthorized users. In addition to the XENIX system protections, the SCCS system provides two 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 who are 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.

### 5.9.1 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.

### 5.9.2 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*.

### 5.9.3 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 a release number less than 2, an error will result.

### 5.9.4 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 a release number greater than 5, an error will result.

### 5.9.5 Locking a Version

The lock flag, **l**, lists by release number all versions in a given s-file which 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*.

## 5.10 Repairing SCCS Files

The SCCS system carefully maintains all SCCS files, making damage to the files very rare. However, damage can result from hardware malfunctions, which 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.

### 5.10.1 Checking an S-file

You can check a file for damage by using the **-h** option of the **admin** command. This 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 filename. 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.

### 5.10.2 Editing an S-file

When an s-file is discovered to be damaged, it is a good idea to restore a backup copy of the file from a backup disk rather than attempting to repair the file. (Restoring a backup copy of a file is described in the XENIX *Operations 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 section *sccsfile* (F) in the XENIX *Reference Manual*, to locate the part of the file which is damaged. Use extreme care when making changes; small errors can cause unwanted results.

### 5.10.3 Changing an S-file's Checksum

After repairing a damaged s-file, you must change the file's checksum by 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.

### 5.10.4 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 affect 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.

### 5.10.5 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.

## 5.11 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.

### 5.11.1 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 ...
```

### 5.11.2 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 filename. For example, the command

```
admin -i s.demo.c <demo.c
```

causes **admin** to create a new s-file named *s.demo.c* which uses 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* which contains the first version of the concatenated files *mod1.c* and *mod2.c*.

### 5.11.3 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.

### 5.11.4 Adding a Comment to the First Version

You can add a comment to the first version of 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.

### 5.11.5 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.

### 5.11.6 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 automatically apply changes to a version while retrieving it for editing. 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 using the **–i** and **–x** options, **get** compares them with the deltas that are 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.

## 5.11.7 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. This option causes the **get** command to create an l-file which 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. The option performs the same function as the –l options except it copies the list to the standard output file. 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.

### 5.11.8 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. This 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 -m option is typically used to review the history of each line in a given version.

### 5.11.9 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. This 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.

### 5.11.10 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. This option causes the command to display the differences, in a format similar to the output of the XENIX **diff** command.

### 5.11.11 Displaying File Information

You can display information about a given version by using the -g option of the **get** command. This 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 -g 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.

### 5.11.12 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 (see the section "Protecting S-files" given earlier in this chapter). The command will also refuse to remove a delta which 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. Type indicators are described in full in the section *delta*(CP) in the XENIX *Reference Manual*.

### 5.11.13 Searching for Strings

You can search for strings in files created from an s-file by using the **what** command. This command searches for the symbol #(@) (the current value of the %Z% keyword) in the given file. It then prints, on the standard output, all text immediately following the symbol, up to the next double quote (" ), greater than (>), backslash (\), newline, or (non-printing) NULL character. For example, if the s-file *s.demo.c* contains the following 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 that have not been created by SCCS commands.

## 5.11.14 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.

# Chapter 6

# Adb: A Program Debugger

## 6.1 Introduction

**Adb** is a debugging tool for C and assembly language programs. It carefully controls the execution of a program while letting you examine and modify the 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, breakpoint, and 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.

## 6.2 Starting and Stopping Adb

**Adb** provides a powerful set of commands to let you 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.

### 6.2.1 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 named "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 filename. 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.

### 6.2.2 Starting With a Core Image File

**Adb** also lets you 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 and and the program file. The command line has the form

adb *programfile corefile*

where *programfile* is the filename of the program that caused the error, and *corefile* is the filename of 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 such a file 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 filename. For example, the command

```
adb sample –
```

prevents **adb** from searching your current working directory for a core file.

### 6.2.3 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 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.

### 6.2.4 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 lets you write directly to memory after executing the given program. See the section "Patching Binary Files" later in this chapter.

### 6.2.5 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 the *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 there is 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.

### 6.2.6 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 CNTRL-D.

You cannot stop **adb** command by pressing the INTERRUPT or QUIT keys. These keys are caught by **adb** and cause it to to wait for a new command.

## 6.3 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. If the question mark (?) is given, the *programfile* is examined. If the slash (/) is given, the *corefile* is examined.

### 6.3.1 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 upper and lowercase). The following are valid numbers

| Decimal | Octal | Hexadecimal |
|---|---|---|
| 34 | 042 | 0x22 |
| 4090 | 07772 | 0xffa |

Although decimal numbers are displayed with trailing decimal point (.), you must not use the decimal point when typing the number.

## Symbols

Symbols are the names of globol 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*(CP)).

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 displaying 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.

| | |
|---|---|
| b | base address of data segment |
| d | size of data |
| m | execution type |
| s | size of stack |
| t | size of text |

A user can access locations by using the **adb** defined variables. The

```
$v
```

request prints these variables.

**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 an variable in an expression by preceding the variable name with an 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. The command displays any variable whose value 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 the . and the " 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** lets you use the current value of the CPU registers in expressions. You can give the value of the 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 flag |
| 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 the 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 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"

### 6.3.2 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 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; **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.

It is possible to 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" cause 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 "*C" 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 "@@"

### 6.3.3 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" 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=
```

## 6.3.4 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 the 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.

### 6.3.5 An Example: Simple Formatting

This example illustrates how to combine formats in ? or / commands to display different types of values when 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 view a variable in a variety of formats. For example, you can display the **long** variable "lnum" as a 4-byte 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
```

## 6.4 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. This means it is not possible to refer to individual C statements when using the debugger. In order 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: a C Compiler").

### 6.4.1 Executing a Program

You can execute a program by using the **:r** command. The command has the form

```
[ 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 filenames 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 5 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 removes the contents of all registers and destroys the current stack before starting the program. This kills any previous copy of the program you may have been running.

### 6.4.2 Setting Breakpoints

You can set a breakpoint in a program by using the **:b** 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] :b command
```

where *address* must be 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:b
```

sets a breakpoint at the start of the function named "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 which 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:b
```

sets a breakpoint at the fifth invocation of the function "light". The breakpoint does not stop the function until it has been called at least five times.

Note that no more than 16 breakpoints at a time are allowed.

### 6.4.3 Displaying Breakpoints

You can display the location and count of each currently defined breakpoint by using the **$b** command. The command displays a list of the breakpoints given by address. If the breakpoint has a count and/or a command, these are given as well.

The **$b** command is useful if you have creating several breakpoints in your program.

### 6.4.4 Continuing Execution

You can continue the execution of a program after it has been stopped by a breakpoint by using the **:c** command. The command has the form

```
[address] [,count] :c [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*(S) in the XENIX *Reference Manual*).

If no *address* is given, the program starts at the next instruction after the breakpoint. If a *count* is given, **adb** ignores the first *count* breakpoints.

### 6.4.5 Stopping a Program with Interrupt and Quit

You can stop execution of a program at any time by pressing the INTERRUPT (CTRL-\) or QUIT (DEL) keys. These keys stop the current program and return control to **adb**, The key are especially useful for programs that have 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 **:c** 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

```
:c 0
```

The command argument "0" prevents a pending signal from being sent to the program.

### 6.4.6 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* must be 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 a *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 5 instructions in the function *main*.

### 6.4.7 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.

### 6.4.8 Deleting Breakpoints

You can delete a breakpoint from a program by using the **:d** command. The command has the form

```
address:d
```

where *address* is the address of the breakpoint you wish to delete.

The **:d** 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:d
```

### 6.4.9 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 which 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 upto 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.

## 6.4.10 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.

## 6.4.11 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 that have been 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
Iscadr:     02
__iob:      01664
_edata:     0
```

### 6.4.12 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 **:b** command. For example, to set a breakpoint at the start of the function "f", type

```
        f:b
```

You can use similar commands for the "g" and "h" functions. Once you can 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 the "f" function), 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:p
```

and continue the program by typing

```
:c
```

**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 commands 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 10 breakpoints by typing

```
,10:c
```

**Adb** starts the program and display the running message again. It does not stop the program until exactly ten breakpoints have been encountered. It 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:
```

## 6.5 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.

### 6.5.1 Displaying the Memory Maps

**Adb** interprets these different file formats and provides access to the different segments through a set of maps. To print the maps type: **$m** command. The command has the form

```
$m [segment]
```

In nonshared files, both text (instructions) and data are intermixed. This makes it impossible for **adb** to differentiate data from instructions and some of the printed symbolic addresses look incorrect; for example, printing data addresses as offsets from routines.

In shared text, the instructions are separated from data and the

```
?*
```

accesses the data part of the *a.out* file. This request tells **adb** to use the second part of the map in the *a.out* file. Accessing data in the *core* file shows the data after it was modified by the execution of the program. Notice also that the data segment may have grown during program execution. In shared files the corresponding *core* file does not contain the program text.

If you have started **adb** but have not executed the program, the **$m** command display has the form

```
?map    'a.out'
b1=0              e1  =03700          f1=040
b2=0              e2  =0              f2=03740
/map    '_'
b1=0              e1  =0100000000     f1=0
b2=0              e2  =0              f2=0
```

The b, e, and f fields are used by **adb** to map addresses into file addresses. The "f1" field is the length of the header at the beginning of the file (0x34 bytes for an a.out file and 02000 bytes for a *core* file). The "f2" field is the displacement from the beginning of the file to the data. For unshared files with mixed text and data this is the same as the length of the header; for shared files this is the length of the header plus the size of the text portion.

The "b" and "e" fields are the starting and ending locations for a segment. Given an address, "A", the location in the file (either *a.out* or *core*) is calculated as:

$$b1 \leq A \leq e1 \Rightarrow \text{file address} = (A-b1)+f1$$
$$b2 \leq A \leq e2 \Rightarrow \text{file address} = (A-b2)+f2$$

## 6.6 Miscellaneous Features

The following sections explain how to use a number of useful commands and features of **adb**.

### 6.6.1 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 a **l** command. For example, the commands

```
?l 'Th'; ?s
```

search for and display a string that begins with the characters "Th"

### 6.6.2 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 filename. 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
```

### 6.6.3 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 lineprinter or special terminal. For example, the command

```
120$w
```

sets the display width to 120 characters, a common maximum width for lineprinters.

### 6.6.4 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 that are 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 that are 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.

### 6.6.5 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.

### 6.6.6 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 restores control **adb**.

### 6.6.7 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 easier 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.

### 6.6.8 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, then start **adb** with the name of the directory, redirecting its input to the script.

First, you can create a script file named *script*. A directory file normally contains one or more entries. Each entry consists of an unsigned "inumber" and a 14 character filename. 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 filename 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 filename of the device associated with 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 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 hyphen (-) 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
```

## 6.7 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.

### 6.7.1 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 2 byte values; L for 4 bytes.

The

```
          ?l
```

commands 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.

### 6.7.2 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 2 byte values; **W** writes 4 bytes. For example, the following commands change the word "This" to "The "

```
?l ´Th´
?W ´The´
```

Note that **W** is used to change all four characters.

### 6.7.3 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.

# Chapter 7
# As: An Assembler

## 7.1 Introduction

This chapter describes the usage and input syntax of the XENIX 8086/186/286 assembler, **as**. The assembler produces relocatable object modules from 8086, 186, and 286 assembly language files. Object modules contain relocation information and a complete symbol table, and can be linked to other objects modules using the XENIX linker, **ld**.

**As** is designed to be used in those rare cases where C programs do not satisfy a programming requirement. Thus, you can make complete programs by combining **as** object modules with object modules created by the XENIX C compiler, **cc**.

This chapter does not teach assembly language programming, nor does it give a detailed description of 8086, 186, and 286 instructions. For information on these topics, you will need other references.

## 7.2 Command Usage

**As** is invoked as follows:

as [ *options* ] *filename*

The *options* are one or more assembler options, and *filename* is the name of an assembly language source file. The source file name should have the ".s" extension. Source files with this filename extension can also be assembled using the cc command. See Chapter 2, "Cc: A C Compiler."

Although **as** has several options, the most commonly used are the **−l** and **−o** options. The **−l** option causes the assembler to create a program listing that includes the source, the assembled code, and any error messages. The listing is given the ".lst" filename extension. The **−o** option directs **as** to place the object module in the named file. The option has the form:

−o *outfile*

where *outfile* is the name of the file to receive the object module. If you do not use the **−o** option, the object files created by **as** have the same name as the source file except that the ".s" filename extension is replaced with a ".o".

For a complete list of the assembler options, see *as*(CP) in the XENIX *Reference Manual*.

## 7.3 Characters, Numbers, and Names

All assembly language programs consist of a combination of characters, numbers, and names. Names and numbers are used to identify values in instruction statements. Characters are used to form the names or numbers, or to form character constants. The following sections describe what characters can be used in a program and how to form numbers and names.

### 7.3.1 Character Set

**As** recognizes the following character set:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
? @ _ $ : . [ ] ( ) < > + − / * &

### 7.3.2 Integers

**Syntax**

*digits*
*digits*B
*digits*Q
*digits*O
*digits*D
*digits*H

An integer represents an integer number. It is a combination of binary, octal, decimal, or hexadecimal *digits* and an optional radix. The *digits* are a combination of one or more digits of the specified radix: B, Q, O, D, or H. If no radix is given, as uses decimal by default. The following table lists the digits that can be used with each radix.

| Radix | Type | Digits |
|---|---|---|
| B | Binary | 0 1 |
| Q<br>O | Octal | 0 1 2 3 4 5 6 7 |
| D | Decimal | 0 1 2 3 4 5 6 7 8 9 |
| H | Hexadecimal | 0 1 2 3 4 5 6 7 8 9 A B C D E F |

Hexadecimal numbers must always start with a decimal digit (0−9). The hexadecimal digits A through F can be given as either upper or lower case.

The maximum number of digits in an integer depends on the instruction or directive in which the integer is used.

**Examples**

```
01011010B      132Q   5AH   90D   90
01111B         17O    0FH   15D   15
```

You can override the default radix by using the .RADIX directive. See section, ".RADIX Directive," given later in this chapter.

### 7.3.3 Real Numbers

**Syntax**

*digits.digits*E [ +|− ] *digits*

A real number represents a number having an integer, a fraction, and an exponent. The *digits* can be any combination of decimal digits. Digits before the decimal point (.) represent the integer part, and those after the point represent the fraction. The digits after the exponent mark (E) represent the exponent. The exponent is optional. If an exponent is given, the plus (+) and minus (−) signs can be used to indicate its sign.

Real numbers can only be used with the **DD**, **DQ**, and **DT** directives. The maximum number of digits in the number and the maximum range of exponent values depends on the directive.

**Examples**

```
25.23     2.523E1  2523.0E-2
```

## 7.3.4 Encoded Real Number

**Syntax**

*digits***R**

An encoded real number is an 8, 16, or 20–digit hexadecimal number that represents a real number in encoded format. An encoded real number has a sign field, a biased exponent, and a mantissa. These values are encoded as bit fields within the number. The exact size and meaning of each bit field depends on the number of bits in the number. The *digits* must be hexadecimal digits. The number must begin with a decimal digit (0–9).

Encoded real numbers can only be used with the **DD**, **DQ**, and **DT** directives. The maximum number of digits for the encoded numbers used with **DD**, **DQ**, and **DT** must not exceed 8, 16, and 20 digits, respectively. (If a leading zero is supplied, the number must not exceed 9, 17, and 21 digits.)

**Example**

```
3F800000R               ; 1.0 for DD
3FF0000000000000R       ; 1.0 for DQ
```

## 7.3.5 Packed Decimal Numbers

**Syntax**

[+ | –] *digits*

A packed decimal number represents a decimal integer that is to be stored in packed decimal format. Packed decimal storage has a leading sign byte and 9 value bytes. Each value byte contains two decimal digits. The high order bit of the sign byte is 0 for positive values, and 1 for negative values.

Packed decimals have the same format as other decimal integers except that they can take an optional plus (+) or minus (–) sign and can only be defined with the DT directive. A packed decimal must not have more than 18 digits.

**Examples**

```
1234567890              ; encoded as 00000000001234567890
-1234567890             ; encoded as 80000000001234567890
```

## 7.3.6 Character and String Constants

**Syntax**

' *characters* '
" *characters* "

A character constant is a constant composed of a single ASCII character. A string constant is a constant composed of two or more ASCII characters. Constants must be enclosed in matching single quotation or double quotation marks.

**Examples**

```
'a'
'ab'
"a"
"This is a message."
```

## 7.3.7 Names

**Syntax**

*characters...*

A name is a combination of letters, digits, and special characters that can be used in instruction statements to labels, variables, and symbols. Names have the following formatting rules:

1. A name must begin with a letter, an underscore (_), a question mark (?), a dollar sign ($), or an at sign (@).

2. A name can have any combination of upper and lowercase letters. Upper and lowercase letters are unique unless the **−Mu** or **−Mx** option is used. (See *as*(CP) in the XENIX *Reference Manual*.)

3. A name can have any number of characters, but only the first 31 characters are used. All other characters are ignored.

**Examples**

```
subrout3
Array
_main
```

## 7.3.8 Reserved Names

A reserved name is any name that has a special, predefined meaning to the assembler. Reserved names include instruction and directive mnemonics, register names, and predefined group and segment names. These names can only be used as defined and must not be redefined.

The following is a list of all reserved names except instruction mnemonics. For a complete list of instruction mnemonics, see "Instruction Mnemonics" given later in this chapter.

| | | | | | |
|---|---|---|---|---|---|
| %OUT | DD | EQU | LE | SEG | .286c |
| AH | DGROUP | ES | LENGTH | SEGMENT | .286p |
| AL | DH | EVEN | LOW | SHL | .287 |
| AND | DI | EXTRN | LT | SHORT | .8086 |
| ASSUME | DL | FAR | MOD | SHR | .8087 |
| AX | DQ | GE | NAME | SI | .LFCOND |
| BH | DS | GROUP | NE | SIZE | .LIST |
| BL | DT | GT | NEAR | SP | .RADIX |
| BP | DW | HIGH | NOT | SS | .SFCOND |
| BX | DWORD | IF | OFFSET | SUBTTL | .TFCOND |
| BYTE | DX | IF1 | OR | TBYTE | .TYPE |
| CH | ELSE | IF2 | ORG | THIS | .XLIST |
| CL | END | IFDEF | PAGE | TITLE | = |
| COMMENT | ENDIF | IFE | PROC | TYPE | _BSS |
| CS | ENDP | IFNDEF | PTR | WORD | _DATA |
| CX | ENDS | INCLUDE | PUBLIC | XOR | _TEXT |
| DB | EQ | LABEL | QWORD | .186 | |

All upper and lowercase combinations of these names are considered to be the same name. For example, the names "Length" and "LENGTH" are the same name for the LENGTH operator.

## 7.4 Statements and Comments

All assembly language source files consist of one or more statements. Statements define the actions to be taken by the assembler, such as the generation of instruction code or the declaration of a variable.

Assembly language source files can also also contain comments. Comments are programmer–supplied text that describes the action of the program or the purpose of declared variables or labels.

The following sections describe the format of statements and comments in detail.

### 7.4.1 Statements

**Syntax**

[ *name* ] *mnemonic* [ *operands* ]

A statement is a combination of a name, an instruction or directive mnemonic, and one or more operands. A statement represents an action to be taken by the assembler, such as generating a machine instruction or generating one or more bytes of data.

Statements have the following formatting rules:

1. A statement can begin in any column.

2. Statements with names normally start in column 1.

3. A statement must not be longer than one line (128 characters).

4. A statement must be terminated by a newline character. This includes that last statement in the source file.

**Examples**

```
count   db      0
mov     ax, bx
assume cs:_TEXT, ds:DGROUP
_main   proc    far
```

### 7.4.2 Comments

**Syntax**

*;text*

A comment is any combination of characters preceded by a semicolon (;) and terminated by a newline character. Comments describe the action of a program at the given point. For this reason, the assembler completely ignores comments.

Comments can be placed anywhere in a program, including on the same line as a statement. The comment must be placed after all names, mnemonics, and operands have been given. A comment must not be longer than one line, that is, it must not contain any embedded newline characters. For very long comments, the COMMENT directive can be used.

**Examples**

```
; This comment is alone on a line.
        mov     ax, bx  ; This comment follows a statement.
; Comments can contain reserved words like _TEXT.
```

Although comments are not a required part of a program, they are strongly recommended.

## 7.5 Source Files

**Syntax**

```
statement
.
.
.
END
```

An assembly language source file is any combination of statements and comments that ends with an END directive. All source files to be assembled by as must have this form.

In general, as imposes no restriction on the content of a source file. This means a source file can represent a complete program, a part of a program, or just symbols to be used by a program.

In XENIX, source files that define a complete or partial program should contain one or more of the XENIX predefined segments: TEXT, DATA, BSS, and CONST. Object files created from assembly language source files that use the predefined segments are guaranteed to be compatible with object files created from C language source files and with all XENIX libraries. The formats of the TEXT, DATA, BSS, and CONST segments are defined in the following sections. The segments can appear in any order.

Note that, like all other statements in a source file, the statement containing the END directive must terminate with a newline character. As ignores any text it finds on lines after this statement. **Example**

```
        name    Sample
DGROUP          group  _DATA
        assume cs:_TEXT, ds:DGROUP, ss:DGROUP, es:DGROUP

        public _main
        extrn _printf:near

_DATA   segment word public 'DATA'
string  db      'Hello.', 0ah, 0
_DATA   ends

_TEXT   segment  word public 'CODE'
_main   proc     near
        push     bp
        mov      sp, bp
        push     si
        push     di

        mov      ax, offset DGROUP:string
        push     ax
        call     _printf
        add      sp, 2

        pop      di
        pop      si
        mov      bp, sp
        pop      bp
        ret
_main   endp
_TEXT   ends

        end
```

In this example, the module named "Sample" contains two segments: "_TEXT" and "_DATA." _TEXT is the program code segment. It contains a procedure named "_main". _DATA is the program data segment. It contains the definition for the variable "string." This module represents a small model program.

## 7.6 Segments

A segment is a named collection of statements that define a program's code, data, or uninitialized data space. All assembly language source files consist of zero or more segments.

In XENIX, there are four segment types:

TEXT
DATA
BSS
CONST

A TEXT segment defines program code, a DATA segment defines data, a BSS segment defines uninitialized data space, and a CONST segment defines constant data. Each segment type has a unique naming convention and content requirement. These conventions are based on the memory model chosen for the program.

XENIX has four different memory models:

Small (impure text)
Small (pure text)
Middle
Large

A memory model defines the number of actual memory segments a program can occupy when loaded into memory. You select a memory model for an assembly language by choosing the appropriate segment names in your source file and by linking with the appropriate XENIX object modules and libraries when you create the executable program.

The following sections define the formatting rules for TEXT, DATA, BSS, and CONST segment in small, middle, and large assembly language programs.

## 7.6.1 Text Segments

**Syntax**

```
name_TEXT       SEGMENT WORD PUBLIC 'CODE'
                statements
name_TEXT       ENDS
```

A text segment defines a module's program code. It contains *statements* that define instructions and data within the segment. A text segment must have the name *name*_TEXT, where *name* can be any valid name. For middle and large module programs, the module's own name is recommended. For small model programs, "_TEXT" only should be used.

A segment can contain any combination of instructions and data statements. These statements must appear in an order that creates a valid program. All instructions and data addresses in a text segment are relative to the CS segment register. Therefore, the statement

```
assume cs: name_TEXT
```

must appear at the beginning of the segment. This statement ensures that each label and variable declared in the segment will be associated with the CS segment register (see the section, "ASSUME Directive" given later in this chapter).

Text segments should have "PUBLIC" combination type, and must have the class name "CODE." These define loading instructions that are passed to the linker. Although other segment attributes are available, they should not be used. For a complete description of the attributes, see the section, "SEGMENT and ENDS Directives," given later in this chapter.

*Small Model Programs.* Only one text segment is allowed. The segment must not exceed 64 Kbytes. If the segment's complete definition is distributed among several modules, the statement

```
IGROUP group     _TEXT
```

should be used at the beginning of each module to ensure that the segment is placed in a single 64 Kbyte physical segment. All procedure and statement labels should have the NEAR type.

**Example**

```
IGROUP group     _TEXT
        assume cs:_TEXT

_TEXT   segment  word public 'CODE'
_main   proc near
        .
        .
        .
_main   endp
_TEXT   ends
```

*Middle and Large Model Programs.* Multiple text segments are allowed, however, no segment can be greater than 64 Kbytes. To distinguish one segment from another, each should have its own name. Since most modules contain only one text segment, the module's name is often used as part of the text segment's name. All procedure and statement labels should have the FAR type, unless they will only be accessed from within the same segment.

**Example**

```
        assume cs:SAMPLE_TEXT

SAMPLE_TEXT   segment  word public 'CODE'
_main   proc          far
        .
        .
        .
_main   endp
SAMPLE_TEXT   ends
```

## 7.6.2 Data Segments – Near

**Syntax**

```
_DATA   SEGMENT WORD PUBLIC 'DATA'
        statements
_DATA   ENDS
```

A near data segment defines initialized data that is in the segment pointed to by the DS segment register when the program starts execution. The segment is "near" because all data in the segment is accessible without giving an explicit segment value. All programs have exactly one near data segment. Only large model programs can have additional data segments (see the next section).

A near data segment's name must be "_DATA." The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64 Kbytes of data. All data addresses in the segment are relative to the predefined group "DGROUP". Therefore, the statements

```
DGROUP          group _DATA
        assume ds: DGROUP
```

must appear at the beginning of the segment. These statements ensure that each variable declared in the data segment will be associated with the DS segment register and DGROUP (see the sections, "ASSUME Directive" and "GROUP Directive" given later in this chapter).

Near data segments must be "WORD" aligned, must have "PUBLIC" combination type, and must have the class name "DATA." These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see the section, "SEGMENT and ENDS Directives," given later in this chapter.

**Example**

```
DGROUP          group    _DATA
        assume ds: DGROUP

_DATA   segment word public 'DATA'
count   dw      0
array   dw      10 dup(1)
string  db      "Type CANCEL then press RETURN", 0ah, 0
_DATA   ends
```

## 7.6.3 Data Segments – Far

**Syntax**

```
name_DATA       SEGMENT WORD PUBLIC 'FAR_DATA'
                statements
name_DATA       ENDS
```

A far data segment defines data or data space that can only be accessed by specifying an explicit segment value. Only large model programs can have far data segments.

A far data segment's name must be *name*_DATA, where *name* can be any valid name. The name of the first variable declared in the segment is recommended. The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64 Kbytes of data. All data addresses in the segment are relative to the ES segment register. When accessing a variable in a far data segment, the ES register must be set to the appropriate segment value. Also, the segment override operator must be used with the variable's name (see the section, "Attribute Operators," given later in this chapter).

Far data segments must be "WORD" aligned, must have "PUBLIC" combination type, and should have the class name "FAR_DATA." These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see the section, "SEGMENT and ENDS Directives," given later in this chapter.

**Example**

```
ARRAY_DATA   segment  word public 'FAR_DATA'
array     dw     0
          dw     1
          dw     2
          dw     4
table     dw     1600 dup(?)
ARRAY_DATA   ends
```

## 7.6.4 Bss Segments

**Syntax**

```
_BSS     SEGMENT WORD PUBLIC 'BSS'
         statements
_BSS     ENDS
```

A bss segment defines uninitialized data space. A bss segment's name must be "_BSS." The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64 Kbytes. All data addresses in the segment are relative to the predefined group "DGROUP" Therefore, the statements

```
DGROUP         group _BSS
         assume ds: DGROUP
```

must appear at the beginning of the segment. These statements ensure that each variable declared in the bss segment will be associated with the DS segment register and DGROUP (see the sections, "ASSUME Directive" and "GROUP Directive," given later in this chapter).

---

*Note*

The group name DGROUP must not be defined in more than one GROUP directive in a source file. If your source file contains both a DATA and BSS segment, the directive

```
DGROUP group _DATA, _BSS
```

should be used.

---

A bss segment must be "WORD" aligned, must have "PUBLIC" combination type, and must have the class name "BSS." These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see the section, "SEGMENT and ENDS Directives," given later in this chapter.

**Example**

```
DGROUP          group    _BSS
        assume ds: DGROUP

_BSS    segment word public 'BSS'
count   dw      ?
array   dw      10 dup(?)
string  db      30 dup(?)
_BSS    ends
```

### 7.6.5 Constant Segments

**Syntax**

```
CONST   SEGMENT WORD PUBLIC 'CONST'
        statements
CONST   ENDS
```

A constant segment defines constant data that will not change during program execution. Constant segments are typically used in large model programs to hold the segment values of far data segments.

The constant segment's name must be "CONST." The segment can contain any combination of data *statements* defining constants to be used by the program. The segment must not exceed 64 Kbytes. All data addresses in the segment are relative to the predefined group "DGROUP". Therefore, the statements

```
DGROUP          group CONST
        assume ds: DGROUP
```

must appear at the beginning of the segment. These statements ensure that each variable declared in the constant segment will be associated with the DS segment register and DGROUP (see the sections, "ASSUME Directive" and "GROUP Directive," given later in this chapter).

---

*Note*

The group name DGROUP must not be defined in more than one GROUP directive in a source file. If your source file contains DATA, BSS, and CONST segments, the directive

```
DGROUP group _DATA, _BSS, CONST
```

should be used.

---

A constant segment must be "WORD" aligned, must have "PUBLIC" combination type, and must have the class name "CONST." These define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see the section, "SEGMENT and ENDS Directives," given later in this chapter.

**Example**

```
DGROUP          group    CONST
        assume ds: DGROUP

CONST   segment  word public 'CONST'
seg1    dw       ARRAY_DATA
seg2    dw       MESSAGE_DATA
CONST   ends
```

In this example, the constant segment receives the segment values of two far data segments: ARRAY_DATA and MESSAGE_DATA. These data segments must be defined elsewhere in the module.

## 7.7 Labels, Variables, and Symbols

Labels, variables, and symbols are named items that represent instruction addresses, data addresses, and other values. Labels, variables, and symbols that are used in a program must be explicitly defined. Defining an item means associating a type, offset, and value to it. The following sections describe how to define labels, variables, and symbols.

### 7.7.1 Labels

**Syntax**

*name* LABEL [ NEAR | FAR ]

A label definition creates a label *name* and sets its type to NEAR or FAR. The label then represents the address of the following instruction and can be used in **jmp**, **call**, and **loop** instructions to direct execution control to the given instruction.

When a label definition is encountered, the assembler sets the label's value to the value of the current location counter and sets its type to NEAR or FAR. If the label has FAR type, the assembler also sets its segment value to that of the enclosing segment.

NEAR labels can be used with **jmp**, **call**, and **loop** instruction in the enclosing segment only. FAR labels can be used in any segment of the program.

A NEAR label can also be defined using a colon (:). The definition has the form

*name*:

The definition can appear on a line by itself or on a line with an instruction.

**Examples**

```
again           label    near
start           label    far
clear_screen:   mov al,20H
subroutine3:
```

### 7.7.2 Simple Variables

**Syntax**

```
name   DB   init-value
name   DW   init-value
name   DD   init-value
name   DQ   init-value
name   DT   init-value
```

A simple variable represents a single value stored at a single address. The *name* must be the name of the new variable, and *init−value* is the variable's initial value. If the question mark (?) is given, the initial value is undefined.

When a simple variable definition is encountered, the assembler sets *name* to the current offset of the enclosing segment. It sets the variable's type to BYTE, WORD, DWORD, QWORD, or TBYTE, for DB, DW, DD, DQ, and DT, respectively.

**Examples**

```
count          DB     0
start_move     DW     1
diameter       DQ     3.5
temp           DB     ?
```

### 7.7.3 Multiple−Value Variables

**Syntax**

```
name   DB   count   DUP(init-value)
name   DW   count   DUP(init-value)
name   DD   count   DUP(init-value)
name   DQ   count   DUP(init-value)
name   DT   count   DUP(init-value)
```

A multiple−value variable is a collection of one or more values all known by the same *name*. The *count* defines the number of elements in the variable, and DUP(*init−value*) defines the initial value of each element. Each element has the size defined by the given directive.

Multiple dimensional arrays can be defined by giving a list of initial values, or including another DUP directive in the initial value list. If more than one initial value is given, the values must be separated by commas.

Multiple−value variables can also be created using a list of initial values. The definition has the form

```
name   DB   init-value1, init-value2,..., init-valuen
```

where each *init−value* must be separated from the preceding by a comma.

**Examples**

```
1.      Array    DB     100   DUP(1)
2.      table    DW     20   DUP( 1,2,3,4 )
3.      threeD   DB     5   DUP( 5 DUP( 5 DUP (1)))
4.      temp     DD     14   DUP(?)
5.      Set      DB     1, 2, 3
```

Example 1 creates the variable "Array". The array has 100 elements. Each element, a byte, is initialized to 1.

Example 2 creates a two-dimensional array "table". The array has 80 elements, each a word in length. The initial values of the first four elements are 1, 2, 3, and 4, respectively. This pattern is repeated to the end of the array.

Example 3 creates a three-dimensional array "threeD". The array has 125 elements, each one byte in length. The initial value for all elements is 1.

Example 4 creates a variable "temp" that has 14 elements and undefined initial values.

Example 5 creates a 3 element variable whose initial values are 1, 2, and 3.

## 7.7.4 Symbols

**Syntax**

*name* EQU *expression*

A symbol is a name that represents a number, a string, a variable, or an instruction. A symbol definition sets *name* to the value or meaning given by *expression*. The *name* must be a unique name, and *expression* must be a number, string, symbol, an instruction mnemonic, a valid expression, or any other entry such as labels, variables, or memory operands.

**Examples**

```
seven    equ    7h
movb     equ    mov
array1   equ    array
sum      equ    1+4
frame    equ    [bp]
```

## 7.7.5 Absolute Symbols

**Syntax**

*name* = *expression*

An absolute symbol is a name that represents an integer number. The *name* is the name of the symbol, and *expression* can be any valid expression that resolves to a number.

Absolute symbols can be redefined at any time.

**Examples**

```
fifteen  =   0FH
base     =   $+2
skip15   =   base+15
```

## 7.8 Operands

An operand is a constant, label, variable, or symbol that is used in an instruction or directive to represent a value or location to be acted on.

There are the following operand types:

Immediate
Register
Direct Memory
Based
Indexed
Based Indexed

### 7.8.1 Immediate Operands

**Syntax**

*number*|*string*|*symbol*

An immediate operand is a constant value that does not change during execution of the program. An immediate operand can be a number, string constant, absolute symbol, or expression.

**Examples**

```
mov    ax, 9
mov    al, 'c'
mov    bx, local
mov    bx, offset DGROUP:table
```

### 7.8.2 Register Operands

**Syntax**

*reg−name*

A register operand is the name of a CPU register. Register operands direct instructions to carry out their actions on the contents of the given registers. The *reg−name* can be any one of the following:

| | | | | | |
|---|---|---|---|---|---|
| ax | ah | al | bx | bh | bl |
| cx | ch | cl | dx | dh | dl |
| cs | ds | ss | es | sp | bp |
| di | si | | | | |

The **ax**, **bx**, **cx**, and **dx** registers are 16−bit general purpose registers. They can be used for any data or numeric manipulation. The **ah**, **bh**, **ch**, **dh** registers represent the high 8−bits of the corresponding general purpose registers. Similarly, **al**, **bl**,

**cl**, and **dl** represent the low–order 8–bits of the general purpose registers.

The **cs**, **ds**, **ss**, and **es** registers are the segment registers. They contain the current segment address of the code, data, stack, and extra segments, respectively. All instruction and data addresses are relative to the segment address in one of these registers.

The **sp** register is the 16–bit stack pointer register. The stack pointer contains the current top of stack address. This address is relative to the segment address in the **ss** register and is automatically modified by instructions that access the stack.

The **bx**, **bp**, **di**, and **si** registers are 16–bit base and index registers. These are general purpose registers that are typically used for pointers to program data.

The 16–bit flag register contains nine 1–bit flags whose positions and meaning are defined below:

| Flag Bit | Meaning |
|---|---|
| 0 | carry flag |
| 2 | parity flag |
| 4 | auxiliary flag |
| 5 | trap flag |
| 6 | zero flag |
| 7 | sign flag |
| 9 | interrupt–enable flag |
| 10 | direction flag |
| 11 | overflow flag |

Although no name exists for the 16–bit flag register, the contents of the register can be accessed using the **LAHF**, **SAHF**, **PUSHF**, and **POPF** instructions.

## 7.8.3 Direct Memory Operands

**Syntax**

*name*

or

*segment* : *number*

A direct memory operand represents the address of one or more bytes of memory. The *name* must be the name of a variable. The *segment* can be a segment register name (CS, DS, SS, or ES), a segment name, or a group name. The *number* must be a integer.

**Examples**

```
mov     ax, fred
mov     dx, ss:0031H
mov     cx, _DATA:0100H
mov     al, DGROUP:2
```

### 7.8.4 Based Operands

**Syntax**

```
disp[ bp ]
disp[ bx ]
```

A based operand represents a memory address relative to one of the base registers: **bp** or **bx**. The *disp* can be any immediate or direct memory operand. It must resolve to an absolute number or memory address. If no *disp* is given, 0 is assumed.

The effective address of a based operand is the sum of the *disp* value and the contents of the given register. If **bp** is used, the operand's address is relative to the segment pointed to by the **ss** register. If **bx** is used, the address is relative to the segment pointed to by the **ds** register.

Based operands have a variety of alternate forms. The following illustrate a few of these forms.

```
[disp][bp]
[disp +bp]
disp.[bp]
[bp]+disp
```

**Examples**

```
mov     ax, [ bp ]
mov     ax, [ bx ]
mov     ax, 12[ bx ]
mov     ax, fred[ bp ]
```

### 7.8.5 Indexed Operands

**Syntax**

```
disp[ si ]
disp[ di ]
```

An indexed operand represents a memory address that is relative to one of the index registers: **si** or **di**. The *disp* can be any immediate or direct memory operand. It must resolve to an absolute number or memory address. If no *disp* is given, 0 is assumed.

The effective address of an indexed operand is the sum of the *disp* value and the contents of the given register. The address is always relative to the segment pointed to by the **ds** register.

Indexed operands have a variety of alternate forms. The following illustrate a few of these forms.

```
[disp][di]
[disp +di]
disp.[di]
[di]+disp
```

**Examples**

```
mov     ax, [ si ]
mov     ax, [ di ]
mov     ax, 12[ di ]
mov     ax, fred[ si ]
```

### 7.8.6 Based Indexed Operands

**Syntax**

```
disp[ bp ][ si ]
disp[ bp ][ di ]
disp[ bx ][ si ]
disp[ bx ][ di ]
```

A based indexed operand represents a memory address that is relative to a combination of base and index registers. The *disp* can be any immediate or direct memory operand. It must resolve to an absolute number or memory address. If no *disp* is given, 0 is assumed.

The effective address of a based indexed operand is the sum of the *disp* value and the contents of the given registers. If the **bp** register is used, the address is relative to the segment pointed to by the **ss** register. Otherwise, the address is relative to the segment pointed to by the **ds** register.

Based indexed operands have a variety of alternate forms. The following illustrate a few of these forms.

```
[disp][bp][di]
[disp + bp + di]
disp.[bp][di]
[di]+disp+[bp]
```

**Examples**

```
mov     ax, [ bp ][ si ]
mov     ax, [ bx ][ di ]
mov     ax, 12[ bp ][ di ]
mov     ax, fred[ bx ][ si ]
```

## 7.9 Expressions

An expression is a combination of operands and operators that resolves to a single value. Operands in expressions can be absolute values, memory operands, and labels. The result of an expression is also an absolute value, memory operand, or label, depending on the types of operands and operators used.

As provides a variety of operators. Arithmetic, shift, relational, and logical operators manipulate and compare the values of operands. Attribute operators manipulate the attributes of operands, such as their type, address, and size.

The following sections describe the operators in detail.

### 7.9.1 Arithmetic Operators

**Syntax**

*exp1* * *exp2*
*exp1* / *exp2*
*exp1* MOD *exp2*
*exp1* + *exp2*
*exp1* − *exp2*
+ *exp*
− *exp*

Arithmetic operators provide the common mathematical operations. The operators have the following meanings:

| Operator | Meaning |
|---|---|
| * | Multiplication. |
| / | Integer division. |
| MOD | Remainder after division (modulus). |
| + | Addition. |
| − | Subtraction. |
| + | Positive (unary). |
| − | Negative (unary). |

For most operators, the expressions *exp1* and *exp2* must be integer numbers. Labels and variable names can be given with the + and − (subtraction) operators only. With +, at least one must be an integer number. With −, *exp1* can be an integer number, label, or variable name; *exp2* can only be a label or variable name if *exp1* is also one and in the same segment.

**Examples**

```
14 * 4          ; equals 56
14 / 4          ; equals 3
14 MOD 4        ; equals 2
14 + 4          ; equals 18
14 − 4          ; equals 10
14 −  +4        ; equals 10
14 −  −4        ; equals 18
```

### 7.9.2 SHR and SHL Operators

**Syntax**

*expression* SHR *count*
*expression* SHL *count*

The SHR and SHL operators shift the given *expression* right or left by *count* number of bits. Bits shifted off the end of the expression are lost.

**Examples**

```
01110111B SHL 3        ; equals 10111000B
01110111B SHR 3        ; equals 00001110B
```

### 7.9.3 Relational Operators

**Syntax**

*exp1* EQ *exp2*
*exp1* NE *exp2*
*exp1* LT *exp2*
*exp1* LE *exp2*
*exp1* GT *exp2*
*exp1* GE *exp2*

The relational operators compare the expressions *exp1* and *exp2* and return true (0FFFFH) if the given condition is satisfied or false (0000H) if it is not. The expressions must resolve to absolute values. The operators have the following meanings:

| Operator | Condition is satisfied when: |
|---|---|
| EQ | Operands are equal. |
| NE | Operands are not equal. |
| LT | Left operand is less than right. |
| LE | Left operand is less than or equal to right. |
| GT | Left operand is greater than right. |
| GE | Left operand is greater than or equal to right. |

Relational operators are typically used with conditional directives and conditional instructions to direct program control.

**Examples**

```
1  EQ  0                ; false
1  NE  0                ; true
1  LT  0                ; false
1  LE  0                ; false
1  GT  0                ; true
1  GE  0                ; true
```

### 7.9.4 Logical Operators

**Syntax**

NOT *exp*
*exp1* AND *exp2*
*exp1* OR *exp2*
*exp1* XOR *exp2*

The logical operators perform bitwise operations on the given expressions. In a bitwise operation, the operation is performed on each bit in an expression rather than on the expression as a whole. The expressions must resolve to absolute values.

The operators have the following meanings:

| Operator | Meaning |
|---|---|
| NOT | Inverse. |
| AND | Boolean AND. |
| OR | Boolean OR. |
| XOR | Boolean exclusive OR. |

**Examples**

```
NOT   11110000B                ; equals 00001111B
01010101B   AND   11110000B    ; equals 01010000B
01010101B   OR   11110000B     ; equals 11110101B
01010101B   XOR   11110000B    ; equals 10100101B
```

## 7.9.5 Attribute Operators

The attribute operators modify or return the values and types associated with labels, variables, and symbols.

**PTR Operator Syntax**

*type* PTR *expression*

The PTR operator assigns a new *type* to the variable or label given by the *expression*. The *type* must be one of the following size or distance values:

BYTE
WORD
DWORD
QWORD
TBYTE
NEAR
FAR

The *operand* can be any memory operand or label. The BYTE, WORD, and DWORD types can be used with memory operands only. The NEAR and FAR types can be used with labels only.

The PTR operator is typically used with forward references to explicitly define what size or distance a reference has. If not used, as assumes a default size or distance for the reference. The PTR operator is also used to give instructions access to variables in ways that would otherwise generate errors. For example, accessing the high-order byte of a WORD size variable.

**Examples**

```
call     far ptr subrout3
mov      byte ptr [array], 1
add      al, byte ptr [full_word]
```

**Segment Override Operator Syntax**

*segment-register* : *expression*
*segment-name* : *expression*
*group-name* : *expression*

The segment override operator (:) forces the address of a given variable or label to be computed using the beginning of the given *segment−register*, *segment−name*, or *group−name*. If a *segment−name* or *group−name* is given, the name must have been assigned to a segment register with a previous ASSUME directive and defined using a SEGMENT or GROUP directive. The *expression* can be an absolute address or any memory operand. The *segment−register* must be one of CS, DS, SS, or ES.

By default, the effective address of a memory operand is computed relative to the DS or ES register, depending on the instruction and operand type. Similarly, all labels are assumed to be NEAR. These default types can be overridden using the segment override operator.

**Examples**

```
mov     ax, es:[bx][si]
mov     _TEXT:far_label, ax
mov     ax, DGROUP:variable
mov     al, cs:0001H
```

**SHORT Operator Syntax**

SHORT *label*

The SHORT operator sets the type of the given *label* to SHORT. Short labels can be used in "jump" instructions whenever the distance from the label to the instruction is not more than 127 bytes. Instructions using short labels are one byte smaller than identical instructions using near labels.

**Example**

```
1.      jmp     short repeat
```

**THIS Operator Syntax**

THIS *type*

The THIS operator creates an operand whose offset and segment value are equal to the current location counter value and whose type is given by *type*. The *type* can be any one of the following:

BYTE
WORD
DWORD
QWORD
TBYTE
NEAR
FAR

The THIS operator is typically used with the EQU or = directive. It is similar to creating operands with the LABEL directive.

**Examples**

```
tag             equu    this byte
spot_check      =       this near
```

Example 1 is equivalent to the statement "TAG LABEL BYTE".

Example 2 is equivalent to the statement "SPOT_CHECK LABEL NEAR".

**HIGH and LOW Operators Syntax**

HIGH *expression*
LOW *expression*

The HIGH and LOW operators return the high and low 8-bits of the given *expression*. The HIGH operator returns the high 8 bits of the *expression*; the LOW operator returns the low-order 8-bits. The *expression* can be any absolute value.

**Examples**

```
mov     ah, high word_value
mov     al, low 0FFFFH
```

**SEG Operator Syntax**

SEG *expression*

The SEG operator returns the segment value of the given *expression*. The *expression* can be any label, variable, or symbol.

**Example**

```
1.      mov     ax, seg variable_name
2.      mov     ax, seg label_name
```

**OFFSET Operator Syntax**

OFFSET *expression*

The OFFSET operator returns the offset of the given *expression*. The *expression* can be any label, variable, or symbol. The returned value is the number of bytes between item and the beginning of the segment in which it is defined.

The segment override operator (:) can be used to force OFFSET to return the number of bytes between the item in the *expression* and beginning of a named segment or group. This is method used to generate valid offsets for items in a group. See example 2.

**Examples**

```
mov     bx, offset array
mov     bx, offset DGROUP:global
```

The returned value is always a relative value that is subject to change by the linker when the program is actually linked.

**TYPE Operator Syntax**

TYPE *expression*

The TYPE operator returns a number representing the the type of the given *expression*. If the *expression* is a label, variable, or symbol, the operator returns the size of the operand in bytes. If the *expression* is a label, the operator returns 0FFFFH if the label is NEAR, and 0FFFEH if the label is FAR.

**Examples**

```
mov     ax, type array
jmp     (type get_loc) ptr destiny
```

**.TYPE Operator Syntax**

.TYPE *variable*

The .TYPE operator returns a byte that defines the mode and scope of the given *variable*. The *variable* can be any label, variable, or symbol. If the *expression* is not valid, .TYPE returns zero.

The variable's attributes are returned in bits 0, 1, 5, and 7 as follows:

| Bit Position | If Bit=0 | If Bit=1 |
|---|---|---|
| 0 | Absolute value | Program related |
| 1 | -- | Data Related |
| 5 | Not defined | Locally defined |
| 7 | Local scope | External scope |

If both the scope bit and defined bit are 0, the *expression* is not valid.

The .TYPE operator is typically used with conditional directives, where an argument may need to be tested to make a decision regarding program flow.

**Example**

```
x       db      12
z       =       .type x
```

This example sets z to 34.

**LENGTH Operator Syntax**

LENGTH *variable*

The LENGTH operator returns the size of the given *variable* in units of BYTE, WORD, DWORD, QWORD, or TBYTE. The units selected depends on the *variable*'s defined type.

Only variables that have been defined using the DUP operator return values greater than 1. The return value is always the number that precedes the first DUP operator.

In the following examples, assume the definitions:

```
array   dw      100  dup(1)
table   dw      100  dup(1,10 dup(?))
```

**Examples**

```
mov     cx, length array
mov     cx, length table
```

In example 1, LENGTH returns 100.

In example 2, LENGTH returns 100. The return value does not depend on any nested DUP operators.

**SIZE Operator Syntax**

SIZE *variable*

The SIZE operator returns the total number of bytes allocated for the given *variable*. The return value is equal to return value of LENGTH times the return value of TYPE.

In the following example, assume the definition:

```
    array     dw        100   dup(1)
```

**Example**

```
    mov       bx, size array
```

In this example, SIZE returns 200.

### 7.9.6 Expression Evaluation

Expressions are evaluated according to the rules of operator precedence and order. Operations of highest precedence are performed first. Operations of equal precedence are performed from left to right. This default order of evaluation can be overridden using enclosing parentheses. Operations in parentheses are always performed before any other operations. The following table list the precedence of all operators. Operators on the same line have equal precedence.

| Precedence | Operators |
|---|---|
| Highest | |
| 1 | LENGTH, SIZE |
| 2 | ( ) |
| 3 | [ ] |
| 4 | : |
| 5 | PTR, OFFSET, SEG, TYPE, THIS |
| 6 | HIGH, LOW |
| 7 | *, /, MOD, SHL, SHR |
| 8 | +, − |
| 9 | EQ, NE, LT, LE, GT, GE |
| 10 | NOT |
| 11 | AND |
| 12 | OR, XOR |
| 13 | SHORT, .TYPE |
| Lowest | |

**Examples**

```
    8 / 4 * 2                  ; equals 4
    8 / (4 * 2)                ; equals 1
    8 + 4 * 2                  ; equals 16
    (8 + 4) * 2                ; equals 24
    8 EQ 4 AND 2 LT 3          ; equals 0000H (false)
    8 EQ 4 OR 2 LT 3           ; equals 0FFFFH (true)
```

## 7.10 Instruction Mnemonics

As supports the complete instruction sets of the 8086 family of microprocessors. This includes the instruction sets for the 8086, 8087, 186, 286, and 287 microprocessors. The following sections list the instruction mnemonics of all instructions supported by the assembler. Instructions are listed by microprocessor.

The 8086 instructions apply to all microprocessors.

---

*Note*

The .8086, .186, .286c, .286p, .8087, and .287 directives define which instruction sets are recognized by the assembler. By default, as recognizes and assembles all 286 non−privileged and 287 instructions. This set can be limited to 8086 and 8087 instructions by using the .8086 and .8087 directives in the source file. It can be expanded to include 286 privileged instructions by using .286p. For a complete description of these directives, see the section, "Instruction Set Directives," given later in this chapter.

---

## 7.10.1 8086 Instruction Mnemonics

The following is a complete list of the 8086 instructions. As assembles all 8086 instructions by default. It also assembles 286 non−privileged instructions. The .8086 directive can be used to limit assembly to 8086 instructions only.

| 8086 Mnemonic | Full Name |
|---|---|
| AAA | ASCII adjust for addition |
| AAD | ASCII adjust for division |
| AAM | ASCII adjust for multiplication |
| AAS | ASCII adjust for subtraction |
| ADC | Add with carry |
| ADD | Add |
| AND | And |
| CALL | Call |
| CBW | Convert byte to word |
| CLC | Clear carry flag |
| CLD | Clear direction flag |
| CLI | Clear interrupt flag |
| CMC | Complement carry flag |
| CMP | Compare |
| CMPS | Compare byte or word (of string) |
| CMPSB | Compare byte string |
| CMPSW | Compare word string |
| CWD | Convert word to double word |
| DAA | Decimal adjust for addition |
| DAS | Decimal adjust for subtraction |
| DEC | Decrement |
| DIV | Divide |
| ESC | Escape |
| HLT | Halt |
| IDIV | Integer divide |
| IMUL | Integer multiply |
| IN | Input byte or word |
| INC | Increment |

| 8086 Mnemonic | Full Name |
|---|---|
| INT | Interrupt |
| INTO | Interrupt on overflow |
| IRET | Interrupt return |
| JA | Jump on above |
| JAE | Jump on above or equal |
| JB | Jump on below |
| JBE | Jump on below or equal |
| JC | Jump on carry |
| JCXZ | Jump on CX zero |
| JE | Jump on equal |
| JG | Jump on greater |
| JGE | Jump on greater or equal |
| JL | Jump on less than |
| JLE | Jump on less than or equal |
| JMP | Jump |
| JNA | Jump on not above |
| JNAE | Jump on not above or equal |
| JNB | Jump on not below |
| JNBE | Jump on not below or equal |
| JNC | Jump on no carry |
| JNE | Jump on not equal |
| JNG | Jump on not greater |
| JNGE | Jump on not greater or equal |
| JNL | Jump on not less than |
| JNLE | Jump on not less than or equal |
| JNO | Jump on not overflow |
| JNP | Jump on not parity |
| JNS | Jump on not sign |
| JNZ | Jump on not zero |
| JO | Jump on overflow |
| JP | Jump on parity |
| JPE | Jump on parity even |
| JPO | Jump on parity odd |
| JS | Jump on sign |
| JZ | Jump on zero |
| LAHF | Load AH with flags |
| LDS | Load pointer into DS |
| LEA | Load effective address |
| LES | Load pointer into ES |
| LOCK | Lock bus |
| LODS | Load byte or word (of string) |
| LODSB | Load byte (string) |
| LODSW | Load word (string) |
| LOOP | Loop |
| LOOPE | Loop while equal |
| LOOPNE | Loop while not equal |
| LOOPNZ | Loop while not zero |
| LOOPZ | Loop while zero |
| MOV | Move |
| MOVS | Move byte or word (of string) |
| MOVSB | Move byte (string) |

| 8086 Mnemonic | Full Name |
|---|---|
| MOVSW | Move word (string) |
| MUL | Multiply |
| NEG | Negate |
| NOP | No operation |
| NOT | Not |
| OR | Or |
| OUT | Output byte or word |
| POP | Pop |
| POPF | Pop flags |
| PUSH | Push |
| PUSHF | Push flags |
| RCL | Rotate through carry left |
| RCR | Rotate through carry right |
| REP | Repeat |
| RET | Return |
| ROL | Rotate left |
| ROR | Rotate right |
| SAHF | Store AH into flags |
| SAL | Shift arithmetic left |
| SAR | Shift arithmetic right |
| SBB | Subtract with borrow |
| SCAS | Scan byte or word (of string) |
| SCASB | Scan byte (string) |
| SCASW | Scan word (string) |
| SHL | Shift left |
| SHR | Shift right |
| STC | Set carry flag |
| STD | Set direction flag |
| STI | Set interrupt flag |
| STOS | Store byte or word (of string) |
| STOSB | Store byte (string) |
| STOSW | Store word (string) |
| SUB | Subtract |
| TEST | Test |
| WAIT | Wait |
| XCHG | Exchange |
| XLAT | Translate |
| XOR | Exclusive OR |

### 7.10.2 8087 Instruction Mnemonics

The 8087 is a coprocessor that operates in conjunction with the 8086 microprocessor. As assembles all 8087 instructions by default. It also assembles 287 instructions. The .8087 directive can be used to limit assembly to 8087 instructions only.

The following is a list of the 8087 instructions.

| 8087 Mnemonic | Full Name |
|---|---|
| F2XM1 | Calculate $2^x-1$ |

| 8087 Mnemonic | Full Name |
|---|---|
| FABS | Take absolute value of top of stack |
| FADD | Add real |
| FADDP | Add real and pop stack |
| FBLD | Load packed decimal onto top of stack |
| FBSTP | Store packed decimal and pop stack |
| FCHS | Change sign on the top stack element |
| FCLEX | Clear exceptions after WAIT |
| FCOM | Compare real |
| FCOMP | Compare real and pop stack |
| FCOMPP | Compare real and pop stack twice |
| FDECSTP | Decrement stack pointer |
| FDISI | Disable interrupts after WAIT |
| FDIV | Divide real |
| FDIVP | Divide real and pop stack |
| FDIVR | Reversed real divide |
| FDIVRP | Reversed real divide and pop stack twice |
| FENI | Enable interrupts after WAIT |
| FFREE | Free stack element |
| FIADD | Add integer |
| FICOM | Integer compare |
| FICOMP | Integer compare and pop stack |
| FIDIV | Integer divide |
| FIDIVR | Reversed integer divide |
| FILD | Load integer onto top of stack |
| FIMUL | Integer multiply |
| FINCSTP | Increment stack pointer |
| FINIT | Initialize processor after WAIT |
| FIST | Store integer |
| FISTP | Store integer and pop stack |
| FISUB | Integer subtract |
| FISUBR | Reversed integer subtract |
| FLD | Load real onto top of stack |
| FLD1 | Load +1.0 onto top of stack |
| FLDCW | Load control word |
| FLDENV | Load 8087 environment |
| FLDL2E | Load $\log_2 e$ onto top of stack |
| FLDL2T | Load $\log_2 10$ onto top of stack |
| FLDLG2 | Load $\log_{10} 2$ onto top of stack |
| FLDLN2 | Load $\log_e 2$ onto top of stack |
| FLDPI | Load pi onto top of stack |
| FLDZ | Load +0.0 onto top of stack |
| FMUL | Multiply real |
| FMULP | Multiply real and pop stack |
| FNCLEX | Clear exceptions with no WAIT |
| FNDISI | Disable interrupts with no WAIT |
| FNENI | Enable interrupts with no WAIT |
| FNINIT | Initialize processor, with no WAIT |
| FNOP | No operation |
| FNSAVE | Save 8087 state with no WAIT |
| FNSTCW | Store control word without WAIT |
| FNSTENV | Store 8087 environment with no WAIT |

| 8087 Mnemonic | Full Name |
|---|---|
| FNSTSW | Store 8087 status word with on WAIT |
| FPATAN | Partial arctangent function |
| FPREM | Partial remainder |
| FPTAN | Partial tangent function |
| FRNDINT | Round to integer |
| FRSTOR | Restore state |
| FSAVE | Save 8087 state after WAIT |
| FSCALE | Scale |
| FSQRT | Square root |
| FST | Store real |
| FSTCW | Store control word with WAIT |
| FSTENV | Store 8087 environment after WAIT |
| FSTP | Store real and pop stack |
| FSTSW | Store 8087 status word after WAIT |
| FSUB | Subtract real |
| FSUBP | Subtract real and pop stack |
| FSUBR | Reversed real subtract |
| FSUBRP | Reversed real subtract and pop stack |
| FTST | Test top of stack |
| FWAIT | Wait for last 8087 operation to complete |
| FXAM | Examine top of stack element |
| FXCH | Exchange contents of stack element and stack top |
| FXTRACT | Extract exponent and significand from number in top of stack |
| FYL2X | Calculate Y $\log_2 x$ |
| FYL2P1 | Calculate Y $\log_2(x+1)$ |

The 8087 instructions can be freely combined with 8086 and 286 instructions. During normal operation, the 8086 or 286 passes all 8087 instructions to the 8087 coprocessor.

### 7.10.3 186 Instruction Mnemonics

The 186 instruction set consists of all 8086 instructions plus the following instructions.

| 186 Mnemonic | Full Name |
|---|---|
| BOUND | Detect value out of range |
| ENTER | Enter Procedure |
| INS | Input byte/word/string from port DX |
| LEAVE | Leave Procedure |
| OUTS | Output byte/word/string to port DX |
| PUSHA | Push all registers |
| POPA | Pop all registers |

As assembles these instructions by default. The .186 directive can be used to enable instructions if they have been disabled by the .8086 directive.

### 7.10.4 286 Non–Privileged Instruction Mnemonics

The 286 non–privileged instruction set consists of all 8086 instructions plus the following instructions.

| 286 Mnemonic | Full Name |
|---|---|
| BOUND | Detect value out of range |
| ENTER | Enter Procedure |
| INS | Input byte/word/string from port DX |
| LEAVE | Leave Procedure |
| OUTS | Output byte/word/string to port DX |
| PUSHA | Push all registers |
| POPA | Pop all registers |

As assembles these instructions by default. The .286c directive can be used to enable instructions if they have been disabled by the .8086 directive.

### 7.10.5 286 Privileged Instruction Mnemonics

The 286 privileged instruction set consists of all 8086 and 286 non–privileged instructions plus the following.

| 286p Mnemonic | Full Name |
|---|---|
| ARPL | Adjust requested privilege level |
| CLTS | Clear task switched flag |
| LAR | Load access rights |
| LGDT | Load global descriptor table |
| LIDT | Load interrupt descriptor table |
| LLDT | Load local descriptor table |
| LMSW | Load machine status word |
| LSL | Load segment limit |
| LTR | Load task register |
| SGDT | Store global descriptor table |
| SIDT | Store interrupt descriptor table |
| SLDT | Store local descriptor table |
| SMSW | Store machine status word |
| STR | Store task register |
| VERR | Verify read access |
| VERW | Verify write access |

As assembles these instructions only if the .286p directive has been given in the source file.

### 7.10.6 287 Instruction Mnemonics

The 287 instruction set consists of all 8087 instructions plus the following additional instructions.

| 287 Mnemonic | Full Name |
|---|---|
| FSETPM | Set Protected Mode |
| FSTSW AX | Store Status Word in AX (wait) |
| FNSTSW AX | Store Status Word in AX (no-wait) |

As assembles these instructions by default. The .287 directive can be used to enable the instructions if they have been disabled by the .8087 directive.

## 7.11 Directives

Directives give the assembler directions and information about input and output, memory organization, conditional assembly, listing and cross-reference control, and definitions. There are the following directives:

| | | | |
|---|---|---|---|
| .186 | DT | IF1 | %OUT |
| .286c | DW | IF2 | PAGE |
| .286p | ELSE | IFDEF | PROC |
| .287 | END | IFE | PUBLIC |
| .8086 | ENDIF | IFNDEF | .RADIX |
| .8087 | ENDP | INCLUDE | SEGMENT |
| = | ENDS | LABEL | .SFCOND |
| ASSUME | EQU | .LFCOND | SUBTTL |
| COMMENT | EVEN | .LIST | .TFCOND |
| DB | EXTRN | NAME | TITLE |
| DD | GROUP | ORG | .XLIST |
| DQ | IF | | |

Any combination of upper and lowercase letters can be used when giving directives names in a source file.

### 7.11.1 ASSUME Directive

**Syntax**

ASSUME *seg-reg* : *seg-name* ,,,
ASSUME NOTHING

The ASSUME directive selects the given segment register *seg-reg* to be the default segment register for all labels and variables defined in the segment or group given by *seg-name*. Subsequent references to the label or variable will automatically assume the selected register when the effective address is computed. The segment override operator (:) can be used to override the default segment register.

The ASSUME directive can define up to 4 selections: one selection for each of the four segment registers. The *seg-reg* can be any one of the segment register names: CS, DS, ES, or SS. The *seg-name* must be the one of the following:

— The name of a segment defined with the SEGMENT directive.

— The name of a group defined with the GROUP directive.

— The keyword NOTHING.

The keyword NOTHING cancels the current segment selection. The directive "ASSUME NOTHING" cancels all register selections made by a previous ASSUME statement.

**Examples**

```
ASSUME cs:_TEXT
ASSUME ds:DGROUP,ss:DGROUP,cs:IGROUP,es:NOTHING
ASSUME NOTHING
```

### 7.11.2 COMMENT Directive

**Syntax**

COMMENT *delim text delim*

The COMMENT directive causes as to treat all *text* between the given pair of delimiters *delim* as a comment. The delimiter character must be the first non-blank character after the COMMENT keyword. The *text* is all remaining characters up to the next occurrence of the delimiter. The *text* must not contain the delimiter.

The COMMENT directive is typically used for multiple line comments.

**Example**

```
COMMENT *
This comment continues until the
next asterisk.
*
```

### 7.11.3 DB Directive

**Syntax**

*name* DB *expression* ,,,

The DB directive allocates and initializes a byte (8 bits) of storage for each given *expression*. An *expression* can be an integer, a character string constant, a DUP operation, or a constant expression. If two or more expressions are given, they must be separated by commas (,).

The *name* is optional. If a *name* is given, the directive creates a variable of type BYTE whose offset value is the current location counter value.

**Examples**

```
integer         DB      16
string          DB      'ab'
message         DB      "Enter your name: "
constantexp     DB      4 * 3
empty           DB      ?
multiple        DB      1, 2, 3, '$'
duplicate DB    10 dup(?)
high_byte DB    255
```

## 7.11.4 DW Directive

**Syntax**

*name* DW *expression* ,,,

The DW directive allocates and initializes a word (2 bytes) of storage for each given *expression*. An *expression* can be an integer, a 1 or 2 character string constant, a DUP operation, a constant expression, or an address expression. If two or more expressions are given, they must be separated by commas (,).

The *name* is optional. If a *name* is given, the directive creates a variable of type WORD whose offset value is the current location counter value.

**Examples**

```
integer         DW      16728
string          DW      'ab'
constantexp     DW      4 * 3
addressexp      DW      string
empty           DW      ?
multiple        DW      1, 2, 3, '$'
duplicate DW    10 dup(?)
high_word       DW      65535
arrayptr  DW    offset array
array2ptr DW    offset DGROUP:array
```

## 7.11.5 DD Directive

**Syntax**

*name* DD *expression* ,,,

The DD directive allocates and initializes a doubleword (4 bytes) of storage for each given *expression*. An *expression* can be an integer, a real number, a 1 or 2 character string constant, an encoded real number, a DUP operation, a constant expression, or an address expression. If two or more expressions are given, they must be separated by commas (,).

The *name* is optional. If a *name* is given, the directive creates a variable of type DWORD whose offset value is the current location counter value.

**Examples**

```
integer         DD      16728
string          DD      'ab'
real            DD      1.5
encodedreal     DD      3f000000R
constantexp     DD      4 * 3
addressptr      DD      real
empty           DD      ?
multiple        DD      1, 2, 3, '$'
duplicate       DD      10 dup(?)
high_double     DD      4294967295
```

## 7.11.6 DQ Directive

**Syntax**

*name* DQ *expression* ,,,

The DQ directive allocates and initializes a quadword (8 bytes) of storage for each given *expression*. An *expression* can be an integer, a real number, a 1 or 2 character string constant, an encoded real number, a DUP operation, or a constant expression. If two or more expressions are given, they must be separated by commas (,).

The *name* is optional. If a *name* is given, the directive creates a variable of type QWORD whose offset value is the current location counter value.

**Examples**

```
integer         DQ      16728
string          DQ      'ab'
real            DQ      1.5
encodedreal     DQ      3f00000000000000R
constantexp     DQ      4 * 3
empty           DQ      ?
multiple        DQ      1, 2, 3, '$'
duplicate       DQ      10 dup(?)
high_quad       DQ      18446744073709551615
```

## 7.11.7 DT Directive

**Syntax**

*name* DT *expression* ,,,

The DT directive allocates and initializes 10 bytes of storage for each given *expression*. An *expression* can be an integer expression, a packed decimal, a 1 or 2 character string constant, an encoded real number, or a DUP operation. If two or more expressions are given, they must be separated by commas (,).

The *name* is optional. If a *name* is given, the directive creates a variable of type TBYTE whose offset value is the current location counter value.

---

*Note*

The DT directive assumes that constants with decimal digits are packed decimals, not integers.

---

**Examples**

```
packeddecimal    DT    1234567890
integer          DT    16728D
string           DT    'ab'
real             DT    1.5
encodedreal      DT    3f000000000000000000R
empty            DT    ?
multiple         DT    1, 2, 3, '$'
duplicate        DT    10 dup(?)
high_tbyte       DT    1208925819614629174706175D
```

## 7.11.8 END Directive

**Syntax**

END *expression*

The END directive marks the end of the module. The assembler ignores any statements following this directive.

The optional *expression* defines the program entry point. The entry point defines the address at which program execution is to start. If the program has more than one module, only one of these modules can define an entry point. This must be the main module, i.e., the module containing the starting instruction. If no entry point is given, none is assumed.

---

*Note*

If the XENIX cc command is used to link assembly language programs, the command automatically determines its own entry point. In this case, an explicit entry point should not be given.

---

**Example**

```
            public table

  _DATA     segment word public 'DATA'
  table     db        100  dup(1,2,3,4,5)
  _DATA     ends

            END
```

## 7.11.9 EQU Directive

**Syntax**

*name* EQU *expression*

The EQU directive assigns the *expression* to the given *name*. The assembler replaces each occurrence of the *name* with either the text of the *expression*, or with the value of the *expression*, depending on the type of expression given.

The *name* must be a unique name, not previously defined. The *expression* can be an integer, a string constant, a real number, an encoded real number, an instruction mnemonic, a constant expression, or an address expression. Expressions that resolve to integer values in the range 0 to 65,535 cause the assembler to replace the name with a value. All other expressions cause the assembler the replace the name with text.

The EQU directive is typically used as a simple macro facility. Note that the assembler replaces text names before attempting to assemble the statements containing them.

**Examples**

```
  integer        EQU    16728            ; replace with value
  real           EQU    3.14159          ; replace with text
  constantexp    EQU    3 * 4            ; replace with value
  memoryop       EQU    [bp]             ; replace with text
  mnemonic       EQU    mov              ; replace with text
  addressexp     EQU    real             ; replace with text
  string         EQU    'Press Return'   ; replace with text
```

## 7.11.10 = Directive

**Syntax**

*name* = *expression*

The = directive creates an absolute symbol by assigning the numeric value of *expression* to *name*. No storage is allocated. Instead, the assembler replaces each occurrence of the *name* with the value of the given *expression*.

The *expression* can be an integer, a 1 or 2 character string constant, a constant expression, or an address expression. Its value must not exceed 65536. The *name* must be either a unique name, or a name that was previously defined using the = directive.

**Examples**

```
integer         =       16728
string          =       'ab'
constantexp     =       3 * 4
addressexp      =       string
```

Unlike the EQU directive, the = directive can be used to redefine symbols that have been previously defined.

## 7.11.11 EVEN Directive

**Syntax**

EVEN

The EVEN directive increments the location counter to an even value and generates one NOP instruction (90h). If the location counter is already even, the directive is ignored.

The EVEN directive must not be used in byte−aligned segments.

**Example**

```
        org     0
test1   db      1
        EVEN
test2   dw      513
```

In this example, EVEN increments the location counter and generates a NOP instruction (90h). This means the offset of "test2" is 2, not 1.

## 7.11.12 EXTRN Directive

**Syntax**

EXTRN *name*:*type* ...

The EXTRN directive defines an external variable, label, or symbol named *name* and whose type is *type*. An external item is any variable, label, or symbol that has been publicly declared in another module of the program.

The *name* must be the name of a variable, label, or symbol defined in another module of the program and listed in a PUBLIC directive of that module. The *type* must match the type given to the item in its actual definition. It can be any one of the following:

BYTE
WORD
DWORD
QWORD
TBYTE
NEAR
FAR
ABS

The ABS type is reserved for symbols that represent absolute numbers.

Although the actual address is not determined until link time, the assembler assumes a default segment register for the external item based on where the EXTRN directive is placed in the module. If it is outside all segments, the default segment register is DS or CS, depending on whether or not the item is a variable or label. If placed inside a segment, the default register is the same as for other variables defined in that segment. The segment override operator (:) can be used to override an external variable's or label's default segment register.

**Example**

```
EXTRN  tagn:near
EXTRN  var1:word, var2:dword
```

### 7.11.13 GROUP Directive

**Syntax**

*name* GROUP *seg−name*,,,

The GROUP directive associates a group name *name* with one or more segments, and directs the linker to load the named segments into the same physical segment. This means all addresses in the named segments are relative to a single segment value.

The order in which segments of a group are named does not influence the order in which they are loaded. Loading order depends on each segment's class, or on the order the object modules are given to the linker.

All segments in a group must fit within one 64 Kbyte block of memory. This means the total size of a group made up of contiguous segments must not exceed 64 Kbytes.

The *seg−name* must be the name of a segment defined using the SEGMENT directive, or a SEG expression. The *name* must be unique.

Group names can be used with the ASSUME directive and as an operand prefix with the segment override operator (:).

---

*Note*

A group name must not be used in more than one GROUP directive in any source file. If several segments within the source file belong to the same group, all segment names must be given in the same GROUP directive.

---

**Example**

```
DGROUP          GROUP _DATA, _BSS
        assume  ds:DGROUP

_DATA   segment word public 'DATA'
          .
          .
          .
_DATA   ends
_BSS    segment word public 'BSS'
          .
          .
          .
_BSS    ends
      end
```

### 7.11.14 INCLUDE Directive

**Syntax**

INCLUDE *filename*

The INCLUDE directive inserts source code from the source file given by *filename* into the current source file during assembly. The *filename* must name an existing file. A pathname must be given if the file is not in the current working directory. If the named file is not found, as displays an error message and stops.

When as encounters an INCLUDE directive, it opens the named file and begins to assemble its source statements immediately. When all statements have been read, as resumes with the next statement following the directive.

Nested INCLUDE directives are allowed. This means a file named by an INCLUDE directive can contain its own INCLUDE directives.

When a program listing is created, as marks included statements with the letter C.

**Examples**

```
INCLUDE entry
INCLUDE include/record
INCLUDE /usr/include/as/stdio
```

### 7.11.15 LABEL Directive

**Syntax**

*name* LABEL *type*

The LABEL directive creates a new variable or label by assigning the current location counter value and the given *type* to *name*.

The *name* must be unique and not previously defined. The *type* can be any one of the following:

```
BYTE
WORD
DWORD
QWORD
TBYTE
NEAR
FAR
```

The *type* can also be the name of a valid structure type.

**Examples**

```
subroutine          LABEL  far
barray              LABEL  byte
```

### 7.11.16 NAME Directive

**Syntax**

NAME *module−name*

The NAME directive sets the name of the current module to *module−name*. A module name is used by the linker when displaying error messages.

The *module−name* can be any combination of letters and digits. Although the name can be any length, only the first six characters are used. The name must be unique and must not be a reserved word.

**Example**

```
NAME main
```

If the NAME directive is not used, as creates a default module name using the first six characters of a TITLE directive. If no TITLE directive is found, the default name "A" is used.

### 7.11.17 ORG Directive

**Syntax**

ORG *expression*

The ORG directive sets the location counter to *expression*. Subsequent instruction and data addresses begin at the new value.

The *expression* must resolve to an absolute number, i.e., all symbols used in the expression must be known on the first pass of the assembler. The location counter symbol ($) can also be used.

**Examples**

```
ORG     120H
ORG     $+2
```

### 7.11.18 PROC and ENDP Directives

**Syntax**

*name* PROC *type*
    *statements*
*name* ENDP

The PROC and ENDP mark the beginning and end of a procedure. A procedure is a block of instructions that forms a program subroutine. Every procedure has a *name* with which it can be called.

The *name* must be a unique name, not previously defined in the program. The optional *type* can be either NEAR or FAR. NEAR is assumed if no *type* is given. The *name* has the same attributes as a label and can be used as an operand in a **jump**, **call**, or **loop** instruction.

Any number of *statements* can appear between the PROC and ENDP statements. The procedure should contain at least one **ret** statement to return control to the point of call. Nested procedures are allowed.

**Example**

```
_main   PROC    NEAR
        push    bp
        mov     bp, sp
        push    si
        push    di

        mov     ax, offset string
        push    ax
        call    _printf
        add     sp, 2

        pop     di
        pop     si
        mov     sp, bp
        pop     bp
    ret
_main   ENDP
```

### 7.11.19 PUBLIC Directive

**Syntax**

PUBLIC *name*,,,

The PUBLIC directive makes the variable, label, or absolute symbol given by *name* available to all other modules in the program. The *name* must be the name of a variable, label, or absolute symbol defined within the current module. Absolute symbols, if given, can only represent 1 or 2 byte integer or string values.

If the **−Mu** or **−Mx** option is used, **as** converts all lowercase letters in the given names to uppercase before passing the name to the object file. Otherwise, **as** copies the names exactly as spelled.

**Example**

```
          PUBLIC  true, test, start
    true    =       0FFFFH
    test    db      1
    start   label   far
```

### 7.11.20 .RADIX Directive

**Syntax**

.RADIX *expression*

The .RADIX directive sets the default input radix for numbers in the source file. The *expression* defines whether the numbers are binary, octal, decimal, hexadecimal, or numbers of some other base. It must be within the range 2 to 16. The following lists some common values:

| | |
|---|---|
| 2 | – binary |
| 8 | – octal |
| 10 | – decimal |
| 16 | – hexadecimal |

The *expression* is always considered a decimal number regardless of the current default radix.

**Examples**

```
.RADIX 16
.RADIX 2
```

The .RADIX directive does not affect the DD, DQ, or DT directives. Numbers entered in the expression of these directives are always evaluated as decimal unless a numeric suffix is appended to the value.

### 7.11.21 SEGMENT and ENDS Directives

**Syntax**

*name* SEGMENT *align combine 'class'*
*name* ENDS

The SEGMENT and ENDS directives mark the beginning and end of a program segment. A program segment is a collection of instructions and/or data whose addresses are all relative to the same segment register.

The *name* defines the name of the segment. This name can be unique or can be the same name given to other segments in the program. Segments with identical names are treated as the same segment.

The optional *align*, *combine*, and *class* define program loading instructions that are to be used by the linker when forming the executable program. These options are described later.

Segments can be nested. When as encounters a nested segment, it temporarily suspends assembly of the enclosing segment, and begins assembly of the nested segment. When the nested segment has been assembled, as continues assembly of the enclosing segment. Overlapping segments are not permitted.

**Example**

```
SAMPLE_TEXT   SEGMENT WORD PUBLIC 'CODE'
_main     proc far
          .
          .
          .

CONST   SEGMENT WORD PUBLIC 'CONST'           ; nested segment
seg1    dw       ARRAY_DATA
CONST   ENDS                                ; end nesting

          mov      es, seg1
          push     es
          mov      ax, es:pointer
          push     ax
          call     _printf
          add      sp, 4
          .
          .
          .
          ret
_main     endp
SAMPLE_TEXT   ENDS
```

This example contains two segments: "SAMPLE_TEXT" and "CONST". The "CONST" segment is nested within the "SAMPLE_TEXT" segment.

---

*Note*

> Although a given segment name can be used more than once in a source file, each segment definition using that name must have either exactly the same attributes, or attributes that do not conflict with previously defined attributes.

---

**Program Loading Options**

The optional *align* defines where to place the start of the segment when loading into memory. It can be any one of the following:

| | |
|---|---|
| BYTE | segment starts on a byte boundary. |
| WORD | segment starts on a word boundary. |
| PARA | segment starts on a paragraph boundary (16 bytes/paragraph) |
| PAGE | segment starts on a page boundary (256 bytes/page) |

If no *align* is given, PARA is used by default. The actual start address is computed when the program is loaded. The linker, however, guarantees that the address will be on the given boundary. A BYTE boundary address will be any address in memory. A WORD boundary address will be a multiple of 2; a PARAGRAPH boundary a multiple of 16 (10 hexadecimal); and a PAGE boundary a multiple of 256 (100 hexadecimal).

The optional *combine* defines how to combine segments having the same name. It can be any one of the following:

PUBLIC
: Concatenates all segments having the same name and forms a single, contiguous segment. All instruction and data addresses in the new segment are relative to a single segment register, and all offsets are adjusted to represent the distance from the beginning of the new segment.

STACK
: Concatenates all segments as with PUBLIC segments. All addresses in the new segment are relative to the SS segment register. The Stack Pointer (SP) register is set to the first address of the first stack segment.

COMMON
: Creates overlapping segments by placing the start of all segments having the same name at the same address. The length of the resulting area is the length of the longest segment. All addresses in the segments are relative to the same base address.

MEMORY
: Places all segments having the same name in the highest physical segment in memory. If more than one MEMORY segment is given, the segments are overlap as with COMMON segments.

AT *address*
: Causes all label and variable addresses defined in the segment to be relative to the given *address*. The *address* can be any valid expression, but must not contain a forward reference. AT segments typically contain no code or initialized data. Instead, they represent address templates that can be placed over code or data already in memory, such code and data as found in ROM devices. The labels and variables in the AT segments can then be used to access the fixed instructions and data.

If no *combine* is given, the segment is not combined. Instead, it receives its own physical segment when loaded into memory.

---

*Note*

The linker requires at least one stack segment in a program. This segment is typically provided by the C program startup module linked with all programs.

---

The optional *class* defines which segments are to be loaded in contiguous memory. Segments having the same class name are loaded into memory one after another. All segments of a given class are loaded before segments of any other class. The *class* name must be enclosed in single quotation marks.

**Example**

```
        assume  cs:_TEXT
_TEXT   segment word public 'CODE'
        .
        .
        .
_TEXT   ends
```

This example illustrates the general form of a text segment for a small module program. The segment name is "_TEXT". The segment alignment and combine type are "word" and "public," respectively. The class is "CODE." These segment attributes are the required attributes for assembly language programs to be run under XENIX.

## 7.11.22 IF Directives (Conditionals)

The IF directives, or conditional directives, allow conditional assembly of blocks of statements. There are the following conditional directives:

```
IF
IFE
IF1
IF2
IFDEF
IFNDEF
ELSE
ENDIF
```

The six IF directives and the ENDIF and ELSE directives can be used to enclose the statements to be considered for conditional assembly. The conditional block takes the form:

```
IF
        statements
ELSE
        statements
ENDIF
```

where the *statements* can be any valid statements, including other conditional blocks. The ELSE directive is optional.

As assembles the statements in the conditional block only if the condition that satisfies the corresponding IF directive is met. If the conditional block contains an ELSE directive, however, as will assemble only the statements up to the ELSE directive. The statements following the ELSE directive are assembled only if the IF condition is not met. An ENDIF directive must mark the end of the conditional block. No more than one ELSE for each IF directive is allowed.

IF directives can be nested up to 255 levels. To avoid ambiguity, a nested ELSE directive always belongs to the nearest, preceding IF directive.

**IF and IFE Directives Syntax**

```
IF expression
IFE expression
```

The IF and IFE directives test the value of an *expression*. The IF directive grants assembly if the *expression* is non−zero (true). The IFE directive grants assembly if the *expression* is 0 (false). The *expression* must resolve to an absolute value and must not contain forward references.

**Example**

```
IF    debug
            extrn dump:far
            extrn trace:far
            extrn breakpoint:far
ENDIF
```

**IF1 and IF2 Directives Syntax**

```
IF1
IF2
```

The IF1 and IF2 directives test the current assembly pass. The IF1 directive grants assembly on pass 1 only. IF2 grants assembly on pass 2. The directives take no arguments.

**Example**

```
IF1
            %out Pass 1 Starting
ENDIF
```

**IFDEF and IFNDEF Directives Syntax**

```
IFDEF    name
IFNDEF   name
```

The IFDEF and IFNDEF directives test whether or not the given *name* has been defined. The IFDEF directive grants assembly if *name* is a label, variable, or symbol. The IFNDEF directive grants assembly if *name* has not yet been defined.

The *name* can be any valid name. Note that if *name* is a forward reference, it is considered undefined on pass 1, but defined on pass 2. This is a frequent cause of phase errors.

**Example**

```
IFNDEF  buffer
buffer    db      10 dup(?)
ENDIF
```

## 7.11.23 PAGE Directive

**Syntax**

```
PAGE   length, width
PAGE +
PAGE
```

The PAGE directive sets the line length and character width of the program listing, increments section page numbering, or generates a page break in the listing.

If a *length* and *width* are given, PAGE sets the maximum number of lines per page to *length*, and the maximum number of characters per line to *width*. The *length* must be in the range 10 to 255. The default is 50. The *width* must be in the range 60 to 132. The default is 80.

If a plus sign (+) is given, PAGE increments the section number and resets the page number to 1. Program listing page numbers have the form:

*section—minor*

By default, page numbers start at 1—1.

If no argument is given, PAGE starts a new output page in the program listing. It copies a form feed character to the file and generates a title and subtitle line.

**Examples**

```
PAGE
PAGE  58,60
PAGE  ,132
PAGE +
```

Example 1 creates a page break.

Example 2 sets the maximum line length to 58, and the maximum width to 60 characters.

Example 3 sets the maximum width to 132 characters. The current line length remains unchanged.

Example 4 increments the current section number and sets the page number to 1.

### 7.11.24 TITLE Directive

**Syntax**

TITLE *text*

The TITLE directive defines the program listing title. It directs as to copy *text* to the first line of each new page in the program listing. The *text* can be any combination of characters up to 60 characters long.

No more than one TITLE directive per module is allowed.

**Example**

```
TITLE progl -- 1st Program
```

Note that the first six non—blank characters of the title will be used as the module name if the module does not contain a NAME directive.

### 7.11.25 SUBTITLE Directive

**Syntax**

SUBTTL *text*

The SUBTTL directive defines the listing subtitle. It directs as to copy *text* to the line immediately after the title on each new page in the program listing. The *text* can be any combination of characters. Only the first 60 characters are used. If no characters are given, the subtitle line is left blank.

Any number of SUBTTL directives can be given in a program. Each new directive replaces the current subtitle with the new *text*.

**Examples**

```
1.      SUBTTL   Special I/O Routine
2.      SUBTTL
```

Example 1 creates the subtitle "Special I/O Routine."

Example 2 creates a blank subtitle.

### 7.11.26 %OUT Directive

**Syntax**

%OUT *text*

The %OUT directive directs as to display the *text* at the user's terminal. The directive is useful for displaying messages during specific points of a long assembly.

The %OUT directive generates output for both assembly passes. The IF1 and IF2 directives can be used to control when the directive is processed.

**Example**

```
if1
        %OUT First Pass -- Okay
endif
```

### 7.11.27 .LIST and .XLIST Directives

**Syntax**

```
.LIST
.XLIST
```

The .LIST and .XLIST directives control which source program lines are copied to the program listing. The .XLIST directive suppresses copying of subsequent source lines to the program listing. The .LIST directive restores copying. The directives are typically used in pairs to prevent a section of a given source file from being copied to the program listing.

The .XLIST directive overrides all other listing directives.

**Example**

```
.XLIST
        ;listing suspended here
.LIST
        ;listing resumes here
```

### 7.11.28 .SFCOND, .LFCOND, and .TFCOND Directives

**Syntax**

```
.SFCOND
.LFCOND
.TFCOND
```

The .SFCOND and .LFCOND directives determine whether or not conditional blocks should be listed. The .SFCOND directive suppresses the listing of any subsequent conditional blocks whose IF condition is false. The .LFCOND directive restores the listing of these blocks. The directives can be used like .LIST and .XLIST to suppress listing of the conditional blocks in sections of a program.

The .TFCOND directive sets the default mode for listing of conditional blocks. This directive works in conjunction with the −X option of the assembler. If −X is not given in the as command line, .TFCOND causes false conditional blocks to be listed by default. If −X is given, .TFCOND causes false conditional blocks to be suppressed.

**Examples**

```
.SFCOND
IF 0
        ;This block will not be listed.
ENDIF
.LFCOND
IF 0
        ;This block will be listed.
ENDIF
```

### 7.11.29 Instruction Set Directives

**Syntax**

```
.8086
.8087
.186
.286c
.286p
.287
```

The instruction set directives enable/disable the instruction sets for the given microprocessors. When a directive is given, as will recognize and assemble any instruction mnemonics belonging to the given microprocessor.

The .8086 directive disables the .186 and .286 instruction sets. Any attempt to use these instructions results in an error.

The .8087 directive enables the 8087 instruction set; .287 enables the 287 set. If the −r option has been selected in the as command line, the assembler generates the actual instruction code for these instructions. If the option is not given, as replaces the code with the code for a software interrupt to the floating emulator.

The .186 directive enables the 186 instructions. The .286c directive enables both the 186 instructions and the 286 non−protected instructions. The .286p directive enables the 286 protected instructions.

## 7.12 Program Listing Format

As creates a program listing whenever the **−l** option is given in the command line. The program listing has two parts: the listing of the actual statements and code, and tables detailing the names and attributes of all labels, variables, and symbols in the module.

### 7.12.1 Code Listing

Lines in the code listing part of the program listing have the form:

*offset code source−statement*

The *offset* is from the start of the current segment. The *code* is the instruction code or data generated by the assembler for the given *source−statement*. The *source−statement* is as it appears in the original source file. As displays offsets, code, and data in a hexadecimal radix. It displays line numbers in decimal. If desired, you can change the output radix to the octal radix by using the **−O** option in the as command line. Error messages, if any, are printed directly below the statement containing the error.

A number of special characters are used in the code listing to indicate specific attributes of the given statement or generated code.

C Appears between the code and source statement in lines that have been included as part of an INCLUDE directive.

E Appears beside code that contains an address to an externally defined label or variable.

R Appears beside code containing an address that must be resolved by the linker.

= Appears before the offset on lines containing the = or EQU directive.

*nn*: Appears before instruction code in which the segment override operator have been used. The *nn* is the instruction code for the segment register named in the source statement.

*nn*/ Appears before instruction code in which a **rep** or **lock** instruction has been used. The *nn* is the instruction code for the given prefix instruction.

−−−−

Appears in instruction code that contains a segment reference to a named segment or group. The actual value is resolved by the linker.

*nn* [ *xx* ]

Appears in place of generated data whenever a DUP operator is used. The *nn* is the number of duplicated elements, and *xx* is the initial value(s) given to each element.

As generates code listings for both assembly passes whenever you give the −d option along with the −l option. Code generated for pass 1 usually contains error messages that do not appear in the pass 2 listing. Typically, these messages are caused by forward references that are resolved by the time the second listing is generated. Since phase errors during assembly are caused by the assembler misunderstanding the actual size or type of forward references, the different messages generated for pass 1 and pass 2 can be very useful in tracking down the actual cause of these phase errors.

### 7.12.2 Symbol Table

The second part contains tables that define the names and attributes of all labels, variables, symbols, segments, and groups. It also lists the number of warnings and severe error messages generated.

The program listing contains two symbol tables: a table for segments and groups, and a table for labels, variables, and symbols. If a program does not generate entries for a given table, that table is left out of the program listing.

**Segment and Group Tables** A segment and group table defines the name, size, alignment, combine type, and class of a segment, and the name of a group. The size of a segment is the number of bytes of instruction code and data it contains.

Lines that define segments have the form:

*name.....* *size* *alignment* *combine−type* *class*

The *alignment* is a segment alignment name: BYTE, WORD, PARA, or PAGE. The *combine−type* is a segment combine type: PUBLIC, STACK, COMMON, MEMORY, or AT. The keyword NONE is given if the segment has no combine type. The *class* is the class name given to the segment.

Lines that define groups have the form:

*name......* GROUP

Segments that belong to the named group appear indented immediately under the group line.

**Example**

```
              Name            Size     align    combine  class

 _TEXT  . . . . . .           0044     WORD     PUBLIC 'CODE'
 DGROUP . . . . .                      GROUP
   _DATA  . . . . . .         0200     WORD     PUBLIC 'DATA'
   _BSS   . . . . . .         0031     WORD     PUBLIC 'BSS'
```

**Symbol Tables** A symbol table defines the name, type, value, and other attributes of a label, variable, or symbol. Each line in the table has the form:

*name.....* *type* *value* *attribute*

The *type* defines what the symbol is or what it represents. It usually consists of one or more of the following:

| | |
|---|---|
| L | – a label or variable |
| F | – a far label |
| N | – a near label |
| PROC | – a procedure label |
| Number | – an absolute number |
| Alias | – another symbol |
| Opcode | – an instruction mnemonic |
| Text | – any item not covered by Number, Alias, or Opcode |

The types BYTE, WORD, DWORD, QWORD, TBYTE, NEAR, and FAR may also appear. The *value* is either the actual value of the symbol or its offset in hexadecimal. The *attribute* names the segment to which the symbol belongs. It can also show whether the symbol is external or global. External symbols are declared using the EXTRN directive; global symbols with PUBLIC. The length of each procedure is also given.

**Example**

```
          Name                  Type      Value    Attr

     CLS. . . . . . . . .       N PROC 0036     _TEXT    Length =000E
     MAXCHAR. . . .             Number  0019
     MESSG. . . . . . . .       L BYTE  001C     _BSS
     PARMS. . . . . . . .       L 001C  0000     _BSS
     RECEIVR. . . . . .         L FAR   0000              External
     START. . . . . . . .       F PROC  0000     _TEXT    Length =0036
```

Symbols that have Number, Opcode, Alias, or Text type have been created using an EQU directive or an = directive. All information that follows one of these entries is considered its value, even if the value is simple text.

# Chapter 8
# Lex: A Lexical Analyzer

## 8.1 Introduction

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 in the 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 lines

```
%%
[\t]+$    ;
```

are 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 end 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, in order to save space. The result is still a fast analyzer. In particular, the time taken by a lex program to recognize and partition an input stream is proportional to the length of the input. The number of lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by lex.

In the program written by lex, the 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 lookahead. 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.

## 8.2 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 it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour          printf("color");
mechanise       printf("mechanize");
petrol          printf("gas");
```

would be a start. These rules are not quite enough, since the word *petroleum* would become *gaseum*; a way of dealing with such problems is described in a later section.

## 8.3 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 (these 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 needed to be quoted individually with a backslash (\) or as a group within quotation marks (" ). The quotation mark operator (") indicates that 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, a \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

## 8.4 Invoking *lex*

There are two steps in compiling a lex source program. First, the lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of lex subroutines. The generated program is in a file named lex.yy.c. The I/O library is defined in terms of the C standard library.

The library is accessed by the loader flag *-ll*. So an appropriate set of commands is

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

The resulting program is placed on the usual file *a.out* for later execution. To use lex with yacc see the section "Lex and Yacc" in this chapter and Chapter 9, "Yacc: A 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 library can be avoided.

## 8.5 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: these are the backslash (\), the dash (-), and the caret (^). The dash character indicates ranges. For example

```
[a-z0-9<>_]
```

indicates the character class containing all the lowercase letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using the dash 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 it is desired to include the dash 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 which 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.

## 8.6 Specifying an Arbitrary Character

To match almost any character, the period (.) designates the class of all characters except a newline. 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).

## 8.7 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.

## 8.8 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.

## 8.9 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*.

## 8.10 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 only applies within brackets. If the very last character is dollar sign, the expression only matched at the end of a line (when immediately followed by 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.

## 8.11 Specifying Expression Repetition

The curly braces ({ and }) specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

```
{digit}
```

looks for a predefined string named *digit* and inserts it at that point in the expression.

## 8.12 Specifying Definitions

The definitions are given in the first part of the lex input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of the character *a*.

Finally, an initial percent sign ( % ) is special, since it is the separator for lex source segments.

## 8.13 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 which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the lex 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 which merely copies can be omitted.

One of the simplest things that can be done 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, and the ***s*** indicate string type, and the data are the characters in *yytext*. So this just places the matched string on the output. This action is so common that it may be written as ECHO. 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 there is a rule that 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.

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 variables ***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 lookahead 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 it is desired 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 =-.

Alternatively it might be desired to treat this as = -*a*. To do this, just return the minus sign as well as the letter to the input. 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:

1. *input*() which returns the next input character;

2. *output*(c) which writes the character *c* on the output; and

3. *unput*(c) which 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 lookahead 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 lookahead:

```
+ * ? $
```

Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by lex. The standard lex library imposes a 100 character limit on backup.

Another lex library routine that 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 it is not possible to 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.

## 8.14 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 8 characters while

```
integer
```

matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (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 once and only once. For example, suppose it is desired to count occurrences of both *she* and *he* in an input text. Some lex rules to do this might be

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

where the last two rules ignore everything besides *he* and *she*. Remember that 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 the user really wants to count the included instances of *he*:

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

These rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user 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, that is the word *the* is considered to contain both *th* and *he*. Assuming a two-dimensional array named *digram* to be incremented, the appropriate source is

```
%%
[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 to ability to manipulate the not-yet-processed input.

## 8.15 Specifying Left Context Sensitivity

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The 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:

1. The use of flags, when only a few rules change from one environment to another
2. The use of start conditions with rules
3. The use 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 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;
          }
          }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to lex in the definitions section with a line reading

```
%Start   name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with 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:

```
%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.

## 8.16 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 classes of such things:

1. Any line that is not part of a lex rule or action which begins with a blank or tab is copied into the lex generated program. Such source input prior to the first %% delimiter 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 which contains the actions. This material must look like program fragments, and should precede the first lex rule.

   As a side effect of the above, lines that begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the lex source or the generated code. The comments should follow the conventions of the C language.

2. 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.

3. 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 in the section "Source Format".

## 8.17 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 on 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 get access to yacc's names for tokens is to compile the lex output file as part of the yacc output file by placing the line

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

in the last section of yacc input. Supposing the grammar to be named *good* and the lexical rules to be named *better* the 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 which 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. It may be objected that this program will alter such input items as 49.63 or X7. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as 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 which 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. There follow some rules to change double precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+              |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+        |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+        {
    /* convert constants */
    for(p=yytext; *p != 0; p++)
        {
        if (*p == 'd' || *p == 'D')
            *p+= 'e'- 'd';
         ECHO;
        }
```

After the floating point constant is recognized, it is scanned by the **for** loop to find the letter "d" or "D". The program then adds "'e' –' d' " which converts it to the next letter of the alphabet. The modified constant, now single precision, is written out again. There follow a series of names which must be respelled to remove their initial "d". By using the array *yytext* the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}        |
{d}{c}{o}{s}        |
{d}{s}{q}{r}{t}     |
{d}{a}{t}{a}{n}     |
...
{d}{f}{l}{o}{a}{t}          printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to initial *a*:

```
    {d}{l}{o}{g}      |
    {d}{l}{o}{g}10    |
    {d}{m}{i}{n}1     |
    {d}{m}{a}{x}1     {
            yytext[0] += 'a' - 'd';
            ECHO;
            }
```

And one routine must have initial *d* changed to initial *r*:

```
    {d}1{m}{a}{c}{h}         {
            yytext[0] += 'r' - 'd';
            ECHO;
    }
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters:

```
    [A-Za-z][A-Za-z0-9]*         |
    [0-9]+                       |
    \n                           |
            ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in FORTRAN or with the use of keywords as identifiers.

## 8.18 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 which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter *a* is represented as the same form as the character constant:

```
    'a'
```

If this interpretation is changed, by providing I/O routines which translate the characters, lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by 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 that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a larger number than the size of the hardware character set.

## 8.19 Source Format

The general form of a lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1. Definitions, in the form "name space translation"

2. Included code, in the form "space code"

3. Included code, in the form

   ```
   %{
   code
   %}
   ```

4. Start conditions, given in the form

   ```
   %S name1 name2 ...
   ```

5. Character set tables, in the form

```
%T
number space character-string
%T
```

6. Changes to internal array sizes, in the form

```
%x   nnn
```

where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

| 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. |

# Chapter 9
# Yacc: A Compiler-Compiler

## 9.1 Introduction

Computer program input generally has some structure; every computer program that does input can be thought of as defining an input language which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often 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 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 reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid yacc specifications for their input revealed errors of conception or design early in the program development.

The next several sections describe:

- The 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.

- Some suggestions which should improve the style and efficiency of the specifications.

## 9.2 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

```
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 ;
```

*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. Thus

| | |
|---|---|
| '\n' | Newline |
| '\r' | Return |
| '\'' | Single quotation mark |
| '\\' | Backslash |
| '\t' | Tab |
| '\b' | Backspace |
| '\f' | Form feed |
| '\xxx' | "xxx" in octal |

For a number of technical reasons, the ASCII NUL 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 name1 name2 ...
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
    %start symbol
```

The end of the input to the parser is signaled by a special token, called the endmarker. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it accepts the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as the end of the file or end of the record.

## 9.3 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 get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

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

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
    $ACT : /* empty */
                { $$ = 1; }
        ;

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

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call

```
    node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then 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.

## 9.4 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 there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by yacc, or chosen by the 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:

```
yylex(){
        extern int yylval;
        int c;
        ...
        c = getchar();
        ...
        switch( c ) {
                ...
        case '0':
        case '1':
          ...
        case '9':
                yylval = c-'0';
                return( DIGIT );
                ...
                }
        ...
```

The intent is to return a token number of *DIGIT*, and a value equal to the numerical value of the digit. 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. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the 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 a previous section. 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) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

## 9.5 How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF    shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a .) is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
        reduce 18
```

refers to grammar rule 18, while the action

```
    IF      shift 34
```

refers to state 34.

Suppose the rule being reduced is

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

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a goto action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
    A       goto 20
```

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 lookahead token is the endmarker, and indicates that the parser has successfully done its job. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be in a later section.

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 in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is ***shift 3***, so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is ***shift 6***, so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound : DING  DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off 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 lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on ***place***, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on ***rhyme*** causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by ***$end*** in the ***y.output*** file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG*, *DING DONG*, *DING DONG DELL DELL*, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

## 9.6 Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

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

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr - expr - expr
```

the rule allows this input to be structured as either

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

or as

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

(The first is called left association, the second right association).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

```
expr - expr - expr
```

When the parser has read the second expr, the input that it has seen:

```
expr - expr
```

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule; after applying the rule; the input is reduced to *expr* (the left side of the rule). The parser would then read the final part of the input:

```
- expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

```
expr - expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen

```
expr - expr - expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

```
expr - expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
     expr - expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a shift/reduce conflict. It may also happen that the parser has a choice of two legal reductions; this is called a reduce/reduce conflict. Note that there are never any shift/shift conflicts.

When there are shift/reduce or reduce/reduce conflicts, yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a disambiguating rule.

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.

2. In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the 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 which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by "." , is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

```
stat : IF '(' cond ')' stat
```

Once again, notice that the numbers following shift commands refer to other states, while the numbers following reduce commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there will be at most 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. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references might be consulted; the services of a local guru might also be appropriate.

## 9.7 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 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:

1. The precedences and associativities are recorded for those tokens and literals that have them.

2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.

4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by **yacc**. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially cookbook fashion, until some experience has been gained. The ***y.output*** file is very useful in deciding whether the parser is actually doing what was intended.

## 9.8 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 lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
    stat : error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

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

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any cleanup action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

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

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

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

As mentioned above, the token seen immediately after the *error* symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the 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, but do allow for a simple, fairly effective recovery of the parser from many errors. Moreover, the user can get control to deal with the error actions required by other portions of the program.

## 9.9 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 in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using **yacc**, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a **-ly** argument to the loader. 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 lookahead 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. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

## 9.10 Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

## 9.11 Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file.

1. Use uppercase letters for token names, lowercase letters for nonterminal names. This rule helps you to know who to blame when things go wrong.

2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

3. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

4. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

5. 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.

## 9.12 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.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
     seq : /* empty */
         | seq item
         ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if **yacc** is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

## 9.13 Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
  int dflag;
%}
  ... other declarations ...

%%

prog    : decls  stats
        ;

decls   : /* empty */
              {       dflag = 1;  }
        | decls declaration
        ;

stats   : /* empty */
              {       dflag = 0;  }
        | stats statement
        ;

    ... other rules ...
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, except for the first token in the first statement. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of back door approach can be over done. Nevertheless, it represents a way of doing some things that are difficult to do otherwise.

## 9.14 Handling Reserved Words

Some programming languages permit the user to use words like *if*, which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of yacc; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable". The user can make a stab at it, but it is difficult. It is best that keywords be reserved; that is, be forbidden for use as variable names.

## 9.15 Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros *YYACCEPT* and *YYERROR*. *YYACCEPT* 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.

## 9.16 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 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.

## 9.17 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 **$n** 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(C) will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the 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 of union ...
      } YYSTYPE;
```

The header file must be included in the declarations section, by use of %{ and %}.

Once *YYSTYPE* is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, saying

```
    %left <optype> '+' '-'
```

will 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 – see the previous subsection ) leaves yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
    rule : aaa { $<intval>$ = 3; } bbb
                 { fun( $<intval>2, $<other>0 ); }
         ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in 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 $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the yacc value stack is used to hold *int*'s, as was true historically.

## 9.18 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.

```
%{
# 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 );
        }
```

## 9.19 Yacc Input Syntax

This section has a description of the yacc input syntax, as a yacc specification. Context dependencies, etc., are not considered. Ironically, the yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token *C_IDENTIFIER*. Otherwise, it returns *IDENTIFIER*. Literals (quoted strings) are also returned as *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  ';'
        ;
```

## 9.20 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 a 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 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.

```
%{

# include <stdio.h>
# include <ctype.h>

typedef struct interval {
        double lo, hi;
        } INTERVAL;

INTERVAL vmul(), vdiv();

double   atof();

double   dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start   lines

%union {
        int ival;
        double dval;
        INTERVAL vval;
        }

%token  <ival>  DREG  VREG   /* indices into dreg, vreg arrays */

%token  <dval>  CONST       /* floating point constant */

%type   <dval>  dexp        /* expression */

%type   <vval>  vexp        /* interval expression */

        /* precedence information about the operators */

%left   '+' '-'
```

```
%left  '*' '/'
%left  UMINUS       /* precedence for unary minus */

%%

lines   : /* empty */
        | lines line
        ;

line    : dexp '\n'
                { printf( "%15.8f\n", $1 ); }
        | vexp '\n'
                { printf( "(%15.8f, %15.8f )\n", $1.lo, $1.hi ); }
        | DREG '=' dexp '\n'
                { dreg[$1] = $3; }
        | VREG '=' vexp '\n'
                { vreg[$1] = $3; }
        | error '\n'
                { yyerrok; }
        ;

dexp    : CONST
        | DREG
                { $$ = dreg[$1]; }
        | dexp '+' dexp
                { $$ = $1 + $3; }
        | dexp '-' dexp
                { $$ = $1 - $3; }
        | dexp '*' dexp
                { $$ = $1 * $3; }
        | dexp '/' dexp
                { $$ = $1 / $3; }
        | '-' dexp %prec UMINUS
                { $$ = - $2; }
        | '(' dexp ')'
                { $$ = $2; }
        ;

vexp    : dexp
                { $$.hi = $$.lo = $1; }
        | '(' dexp ',' dexp ')'
                {
                 $$.lo  =  $2;
                 $$.hi  =  $4;
                 if( $$.lo > $$.hi ){
                        printf("interval out of order\n");
                        YYERROR;
                        }
                      }
        | VREG
```

```
                { $$ = vreg[$1]; }
        | vexp '+' vexp
                { $$.hi = $1.hi + $3.hi;
                  $$.lo = $1.lo + $3.lo; }
        | dexp '+' vexp
                { $$.hi = $1 + $3.hi;
                  $$.lo = $1 + $3.lo; }
        | vexp '-' vexp
                { $$.hi = $1.hi - $3.lo;
                  $$.lo = $1.lo - $3.hi; }
        | dexp '-' vexp
                { $$.hi = $1 - $3.lo;
                  $$.lo = $1 - $3.hi;}
        | vexp '*' vexp
                { $$ = vmul( $1.lo, $1.hi, $3 ); }
        | dexp '*' vexp
                { $$ = vmul( $1, $1, $3 ); }
        | vexp '/' vexp
                { if ( dcheck( $3 ) ) YYERROR;
                 $$ = vdiv( $1.lo, $1.hi, $3 ); }
        | dexp '/' vexp
                { if ( dcheck( $3 ) ) YYERROR;
                 $$ = vdiv( $1, $1, $3 ); }
        | '-' vexp  %prec UMINUS
                { $$.hi = -$2.lo; $$.lo = -$2.hi; }
        | '(' vexp ')'
                {     $$  =  $2;  }
        ;

%%

# define BSZ 50  /* buffer size for 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 = '\0';
                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 ) );
        }
```

## 9.21 Old Features

This section mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotation marks (").

2. 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.

3. Most places where '%' is legal, backslash (\) may be used. In particular, the double backslash (\\) is the same as %%, \*left* the same as %*left*, etc.

4. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec

5. Actions may also have the form

   ```
   ={ ... }
   ```

   and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

# Chapter 10
# XENIX to MS–DOS: A Cross Development System

## 10.1 Introduction

The XENIX system provides a variety of tools to create programs that can be executed under control of the MS-DOS operating system. The MS-DOS cross development system lets you create, compile, and link MS-DOS programs on the XENIX system and transfer these programs to an MS-DOS system for execution and debugging.

The complete development system consists of

— The C program compiler **cc**

— The 8086 assembler **as**

— The MS-DOS linker **dosld**

— The MS-DOS libraries (in */usr/lib/dos*)

— The MS-DOS include files (in */usr/include/dos*)

— The **dos**(C) commands

The heart of the cross development system is the **cc** command. The command provides a special **-dos** option which directs the compiler to create code for execution under MS-DOS. When **-dos** is given, **cc** uses the special MS-DOS include files and libraries to create a program. The resulting program file has the correct format for execution on any MS-DOS system.

The **cc** command uses the **dosld** commands to carry out the last part of the compilation process, the creation of the executable program file. **Cc** invokes the **as** command only when 8086 assembly language source files are given in the command line. In most cases, **cc** invokes **as** and **dosld** automatically. You can also invoke them directly when you need to perform special tasks.

The last important step in the cross development process is to transfer the executable program files to an MS-DOS system. Since MS-DOS programs cannot be executed or debugged on the XENIX system, you must copy the resulting programs to MS-DOS file systems before attempting execution. You can do this using the XENIX **dos**(C) commands. For example, the **doscp** command lets you copy files back and forth between XENIX and MS-DOS disks. This means you can transfer program files from the XENIX system to an MS-DOS system, or copy source files from an MS-DOS system to XENIX.

## 10.2 Creating Source Files

You can create program source files using either XENIX or MS-DOS text editors. The most convenient way is to use a XENIX editor, such as **vi**, since this means you do not have to transfer the source files from the MS-DOS system to XENIX each time you make changes to the files.

When creating source files, you should follow these simple rules:

- Use the standard C language format for your source files. MS−DOS source files have the same format as XENIX source files. In fact, many MS−DOS programs, if compiled without the **−dos** option, can be executed on the XENIX system.

- Use the MS−DOS naming conventions when giving file and directory names within a program, e.g., use "\" instead of "/" for the pathname separator. Since the compiler does not check names, failure to follow the conventions will cause errors when the program is executed.

- Use only the MS−DOS include files and library functions. Most MS−DOS include files and functions are identical to their XENIX counterparts. Others have only slight differences. For a complete list of the available MS−DOS include files and functions, and a description of the differences between them and the corresponding XENIX files and functions, see Appendix C of the XENIX *Programmer's Reference*.

If you use a function that does not exist, **dosld** displays an error message and leaves the linked output file incomplete.

## 10.3 Compiling an MS−DOS Source File

You can compile an MS−DOS source file by using the **−dos** option of the XENIX **cc** command. The command line has the form

```
cc −dos options filename ...
```

where *options* are other **cc** command options (as described in Chapter 2), and *filename* is the name of the source file you wish to compile. You can give more than one source file if desired. Each source filename must end with the ".c" extension.

The **cc** command compiles each source file separately, creating an object file for each file, then links all object files together with the appropriate C libraries. The object files created by the **cc** command have the same base name as the corresponding source file, but end with the ".o" extension instead of the ".c" extension. The resulting program file also has the name *a.out* if no name is explicitly given.

For example, the command

```
cc −dos test.c
```

compiles the source file *test.c* and creates the object file *test.o*. It then calls **dosld** which links the object file with functions from the MS−DOS libraries. The resulting program file is named, *a.out*.

You can use any number of **cc** options in the command line. The options work as described in the Chapter 2 of this guide. For example, you can use the **−o** option to explicitly name the resulting program file, or the **−c** option to create object files without creating a program file. In some cases, the default values for an option are different than when compiling for XENIX. In particular, the default directory for library files given with the **−l** option is */usr/lib/dos*. Note that the **−p** (for "profiling") option cannot be used.

## 10.4 Using Assembly Language Source Files

You can direct **cc** to assemble 8086 assembly language source files by including the files in the **cc** command line. Like C source files, assembly language source files may contain only calls to functions in the MS-DOS libraries. Furthermore, the source files must follow the C calling conventions described in Appendix A of the XENIX *Programmer's Reference*. The filename of an assembly language source file must end with the ".s" extension.

When an assembly language source file is given, **cc** automatically invokes **as**, the 8086 assembler. The assembler creates an object file that can be linked with any other object file created by **cc**.

You can invoke the assembler directly by using the **as** command. The command creates an object file just as the **cc** command, but does not create an executable file. For a description of the command and its options, see **as(CP)** in the XENIX *Reference Manual*.

## 10.5 Creating Linking Object Files

You can link MS-DOS object files previously created by **cc** or **as** by giving the names of the files in the **cc** command line. The object files must have been created using **as** or with **cc** using the **-dos** option. Object files created without using the **-dos** option cannot be linked to MS-DOS programs. The object filenames must end with the ".o" extension.

When an object file is given, **cc** automatically invokes **dosld** the MS-DOS linker, which links the given object files with the appropriate C libraries. If there are no errors, **dosld** creates an executable program file named *a.out*.

You can invoke the linker directly by using the **dosld** command. The command creates an MS-DOS program file just as the **cc** command, but does not accept source files. For a description of the command and its options, see **dosld(CP)** is the XENIX *Reference Manual*.

---

*Note*

MS-DOS programs created by **cc** and **dosld** are completely compatible with the MS-DOS system and can be executed on any such system. MS-DOS programs cannot be executed on the XENIX system.

---

## 10.6 Running and Debugging an MS-DOS Program

You can debug an MS-DOS program by transferring the program file to an MS-DOS system and using the MS-DOS debugger, **Debug**, to load and execute the program. The following section explains how to transfer program files between systems. For a description of the **Debug** program, see the appropriate MS-DOS manual.

## 10.7 Transferring Programs Between Systems

You can transfer programs between XENIX and MS-DOS systems by using MS-DOS floppy disks and the XENIX **doscp** command. The **doscp** command lets you copy files to an MS-DOS floppy disk. The command has the form

doscp -r *file-1 dev:file-2*

where -r is the required "raw" option, *file-1* is the name of the MS-DOS program file you wish to transfer, *dev* is the full pathname of a XENIX system floppy disk drive, and *file-2* is the full pathname of the new program file on the MS-DOS disk. The new filename must have the ".exe" extension. The -r option ensures that the program file is copied byte for byte.

To transfer a program file to a MS-DOS system, follow these steps:

1. Insert a formatted MS-DOS diskette into a XENIX system floppy disk drive.

2. Use the **doscp** command to copy the program file to the disk. For example, to copy the program file *a.out* to the file *test.exe* on the MS-DOS disk in the floppy drive */dev/fd0*, type

   ```
   doscp -r a.out  /dev/fd0:/test.exe
   ```

3. Remove the floppy disk from the drive.

You can now insert the floppy disk into the floppy disk drive of the MS-DOS system and invoke the program just as you would any other MS-DOS program.

---

*Note*

> MS-DOS program files that do not end with the ".exe" or ".com" extension cannot be loaded for execution under MS-DOS. When transferring program files from XENIX to MS-DOS, you must make sure you rename *a.out* files to an appropriate ".exe" or ".com" file.

---

On some XENIX systems, you may be able to create an MS-DOS partition on the system hard disk and copy MS-DOS program files to this partition instead of to floppy disks. To execute the program, you must reboot the system, loading the MS-DOS operating system from the MS-DOS partition.

## 10.8 Creating MS-DOS Libraries

You can create a library of your own MS-DOS object files by using the XENIX **ar** command. The command copies object files created by the compiler to a given archive file. The command has the form

ar *archive filename* ...

where *archive* is the name of an archive file, and *filename* is the name of the

MS-DOS object file you wish to add to the library.

---

*Note*

MS-DOS libraries created on the XENIX system are not compatible with libraries created on the MS-DOS system. This means you cannot copy the libraries to the MS-DOS system and expect them to work with the MS-DOS **Link** command.

---

# Chapter 11

# Writing Device Drivers

## 11.1 Introduction

This chapter, along with Chapter 12, "Sample Device Drivers," explains how to write and install device drivers in a XENIX environment. It describes the role of device drivers in a XENIX-based system, and discusses other special considerations involved in writing a device driver. It describes the XENIX model of devices in terms of files, tasks to be performed, and interrupts to be processed.

### 11.1.1 What is a XENIX Device Driver?

For each peripheral device in a XENIX system, there must be a "device driver" to provide the software interface between the device and the system. A XENIX device driver is a set of routines that communicates with a hardware device, and provides a uniform interface to the kernel. This interface allows the kernel to interpret user I/O requests into operating system tasks to be performed.

### 11.1.2 Relationship to XENIX Operating System

The XENIX device driver manages the flow of data and control between the user program and the peripheral devices. The path of an I/O request is shown below, starting with a system call from a user program, and ending at the device driver:

```
+-------------+
| User Program |
+-------------+
        |
        |               User Space
---------------|------------------------
        |               Kernel Space
        |
+------|---------+-------+
|      |         |       |
|      +---->--------->---------> Peripheral
|  Kernel        |Device |        Devices
|                |Drivers|
+------------------------+
```

**User Program Requesting I/O**

### 11.1.3 Device Models Supported by XENIX

The XENIX operating system supports two device models: character devices and block devices. This chapter describes how to write device drivers for both device models.

In general, any device that appears to be a randomly addressable set of fixed-size records is a block device; any other type of device is a character device. For example, disk drives and tape drives are block devices, while terminals and line printers are character devices. The XENIX operating system presents a uniform interface to user programs by providing blocking and unblocking in the kernel. Thus, character and block devices look alike to the user program.

Character device drivers communicate directly with the user program. The process begins when a user program requests a data transfer of some number of bytes between a section of its memory and a specific device. The operating system transfers control to the appropriate device driver. The user program supplies the parameters for the request to the device driver, which, in turn, performs the work. Thus, the operating system has minimal involvement in the request; the data transfer is a private transaction between the user process and the device driver.

Block device drivers require more involvement from the operating system to perform their tasks. Block devices transfer data in fixed-size blocks, and are usually capable of random access. (The device does not need to be capable of random access; magnetic tapes are often read or written using block I/O.) The two factors that distinguish block I/O from character I/O are:

— The size of data transfer requests from the kernel to the device is always a multiple of the system block size (called BSIZE) regardless of the size of the user process' original request. A single user process request can generate many system requests to the driver. BSIZE is 1024 bytes in the 286 version of XENIX. The device's physical block size may be smaller than BSIZE, in which case the device driver initiates multiple physical transfers to transfer a single logical block.

— Transfers are never done directly into a user process' memory area. They are always staged through a pool of BSIZE buffers. Program I/O requests are satisfied directly from the buffers. XENIX commands the device driver to read and write from the buffers as necessary. It manages these buffers to perform services such as blocking and unblocking of data and disk caching.

### 11.1.4 Using Sample Device Drivers

Chapter 12, "Sample Device Drivers," discusses sample device driver source code for a line printer, a terminal, a hard disk drive, and a memory-mapped screen. These source code samples are intended as prototypes from which the experienced programmer can begin writing a device driver for a particular device.

### 11.1.5 Special Device Files

To a XENIX user, a device will usually appear to act like a "file." A file consists of an ordered sequence of bytes. Files that contain data are called "regular files," and files that represent devices are called "special device files." Each file has at least one name; the names of special device files are, by convention, placed in the directory named */dev*.

Each special device file has a "device number," that uniquely identifies the device. The device number consists of two parts, the "major number" and the "minor number." The major number tells the kernel which device driver will handle requests for this special file. The minor number can be used by the driver to provide more information about a particular unit of the devices that it controls (such as the unit number).

Before the user process can request I/O, it must first have opened a "special device file." A special device file looks like an ordinary disk file except that it was created by a utility program called **mknod**(C) described in the XENIX *Reference Manual*. The file appears in a directory and has owner and permission fields, as does any disk file, but it contains no data. Instead, it has a pair of 8-bit numbers, called the "major" and "minor" numbers, associated with it. The command *ls -l* displays these numbers:

```
crw--w--w- 1 davewo    4,  3 Sep 21 09:49 /dev/tty03
brw------- 1 sysinfo   3,  2 Sep 21 09:49 /dev/hd01
```

Here the file */dev/tty03* has a major device number of 4 and a minor device number of 3. The */dev/hd01* file has a major device number of 3 and a minor device number of 2.

When a user process opens the special device file, XENIX recognizes that it is a special device file and uses the major number to index a table of entry points. If the special device file designates a character device, the table used is *cdevsw*; if it designates a block device, the table used is *bdevsw*. These two tables are defined in the */usr/sys/conf/c.c* file generated by the **make** program when the kernel is built. XENIX calls the device driver's open entry-point through this table, supplying as an argument the minor device number. The minor device number usually encodes the unit number, although often a device driver dedicates some of the bits in the minor number to indicating special options, such as "use double density" in the case of a floppy disk.

The convention is for these special device files to have meaningful names and reside in the */dev* directory For example,

/dev/tty03

would normally be associated with the major device number of the serial device driver; its minor number would indicate the fourth port. It is important to note that this is just a *convention*; the system administrator could just as well assign the same major/minor numbers to either of the files

/usr/ellen/magtape
/usr/ellen/tty91

with identical results. The name is for user convenience; XENIX keys solely on the major and minor device numbers.

## 11.2 Kernel Environment

This section briefly discusses a few functional aspects of the XENIX operating system: modes of operation, context switching, system mode stack use, task time processing, and interrupt time processing. It also describes the services provided to device drivers by the XENIX kernel, and the rules that device drivers are required to obey.

### 11.2.1 Modes of Operation

When a process is executing instructions in the user program, it is said to be in "user mode." When it is executing instructions in the XENIX kernel, it is said to be in "system mode." When the kernel receives an interrupt from an external device, it switches to system mode if it was in user mode, and control is passed to the interrupt routine of the appropriate device driver. When the driver is done, it returns, and the processing that was interrupted is resumed. The processing that was interrupted is referred to as "task time processing" and the processing that took place as a result of the interrupt is called "interrupt time processing."

Although all processes originate as user programs, a given process may run in either user or system mode. In system mode, it executes XENIX kernel code and has privileged access to I/O devices and other services. In user mode, it executes the user's program code, and has no special privileges. In fact, XENIX provides a high level of protection around processes in user mode to prevent a user program from inadvertently damaging the system or other user programs. A process voluntarily enters system mode when it makes a system call. When an interrupt or trap is received while a process is executing in user mode, the process will switch into system mode to handle the interrupt. At this time it may lose the CPU and the kernel may decide to switch control or "context" to a different process.

### 11.2.2 Context Switching

Context switching occurs when the kernel decides to transfer control of the CPU from the currently executing process to a different process.

In user mode, the kernel switches context whenever:

— The process' time slice has expired.

— The process makes a system call that cannot be completed immediately; for example, a read from a slow input device.

— An interrupt is received that allows a blocked process to proceed. This case will occur when the process has been sleeping at high priority, waiting for the interrupt handler to call *wakeup*() to indicate a completed I/O request. If the priority at which the process is sleeping is higher than that of the currently running process, a context switch will occur.

In system mode, switching contexts is always voluntary. A process voluntarily gives up the processor when the routine *sleep*() is called. Interrupts can still arrive (they can be locked out for short periods of time, if necessary) but when the interrupt service routine returns, control always passes back to the interrupted process.

### 11.2.3 System Mode Stack

Each process has a special area of memory associated with it, called the "u" area. The u area is not directly accessible to the user (that is, it is not in the process' normal address space). It contains the information the kernel needs to manage the process, and contains space for a system mode stack. When any process makes a system call, its registers are preserved in its u area, and the stack pointer is moved to the beginning of its system mode stack area. When the system call has completed, the registers are restored from the u area, the stack pointer is restored to the process' stack, and control is returned to the process. Since each process in the system has its own u area, a system running N processes has N user stacks and N system stacks.

The XENIX operating system, and therefore the task time portions of the device drivers, uses a fixed-size system mode stack in the u area. In XENIX, the size of this per-process stack is 1024 bytes. It is critical, then, that device driver procedures not create local (frame) buffers of any significant size. The following declaration will cause trouble, since as soon as the routine is called it requires at least 1024 bytes of stack space:

```
open()
{
        char buf [512];
        char buf2[512];
```

Further, interrupt service routines make use of whatever system stack was set up at the time of the interrupt. If the interrupt occurs while the currently running process is in user mode, the interrupt service routine will have the entire u stack area for its use. However, if the interrupt takes place while the process is in system mode, the interrupt routine will be sharing the u stack area. For this reason, interrupt service routines must minimize their frame variable declarations, keeping their frame requirements below 512 bytes.

### 11.2.4 Task Time Processing

The operating system manages a number of processes, each corresponding to a user program. Any particular process may be running in system mode or user mode at any given time. When a process makes a system call to request kernel service, the process switches to system mode, and starts running kernel code. When the kernel is executing code at the request of a user program, it is doing "task time processing."

If there are 50 processes running, there may be as many as 50 simultaneous processes in system mode, each with its own local variables. This requires that all kernel code be re-entrant, but it otherwise greatly simplifies things. Each system process instance has to deal only with servicing the specific system call that the user program requested. The active process' u area is always mapped into the kernel's address space, so when kernel code is executing it has information about the request and process it is serving.

Often the kernel cannot service a request immediately. The request may require doing some I/O, or it could even be a request to wait awhile. When a process in system mode blocks, awaiting some event, the system scheduler schedules some other process, which may be in either user or system mode.

I/O requests from the user process are passed via system calls to the device driver. Some parameters of the request, such as byte count and transfer address, are kept in the u area; these task time portions of the driver can reference and perhaps modify the u area cells, since we know that the currently running process' u area is always mapped into the kernel address space.

### 11.2.5 Interrupt Time Processing

When a device interrupt is received, the tasks performed as a result of the interrupt are referred to as "interrupt time processing." When an interrupt arrives, any of the active processes on the system may be executing. Even if this interrupt signals the completion of a user process' request, the interrupt service routine can take no direct action: the process that was interrupted is almost certainly not the process that initiated the request. Instead, all interrupt time portions of device driver routines must store, in static memory locations, information for the task time portion of the device driver routines to figure out the result of the interrupt service. Any data or status that the interrupt service routine wants to return to the task time portion of the driver (and hence, perhaps to the requesting user program) must also be passed via static memory.

The local (frame) variables of the task portion of the device driver are kept in its system mode stack, which is in the u area. This u area is not mapped into the kernel address space at interrupt time; the u area there belongs to some other process. The correct u area might even be out on the swap disk. Thus, the interrupt service routine must never attempt to store data in the u area or in user memory; and the I/O device itself, via DMA or whatever, must not attempt to transfer directly into the user's memory area.

Usually, this is not a problem. Character devices typically make use of small system supplied buffers called character lists (clists). Block devices use BSIZE buffers in the system buffer pool. The task time portion of the driver transfers the data from the buffers into the user's memory. It may be important that the transfer take place directly into user memory. In such cases, it is necessary to lock the user program into physical memory so that it is not swapped.

Typically, the task time portion of the device driver issues a *sleep*() call when it makes the initial I/O request. The interrupt service routine decides what to do and, if it needs to notify the task time portion (as opposed to issuing another I/O command), it puts any status information and data into static cells and issues a *wakeup*() call to the task portion. The interrupt service routine then exits to the operating system, and the operating system exits the interrupt. The system scheduler soon reschedules the running process so that the one that has just been awakened is executed. The task time portion of the device driver finds that it has returned from the *sleep*() call, and that there are status and data bytes waiting in static memory locations.

Access to static variables that can be modified at interrupt time is interlocked with the *spl5*() routine. The *spl5*() routine raises the interrupt priority of the CPU so that interrupts that might cause a value change are locked out until the *splx*() routine is called. This period must be kept as short as possible. Refer to Section 11.3, "Kernel Support Routines," for a more detailed description of the routines mentioned here.

Device drivers that use the standard interfaces to the kernel are provided with a method for passing information between the interrupt time portion of a driver and the task time portion. Standard buffered I/O device drivers note the outcome of the data transfer in the buffer headers associated with the transfer. The header for the list of transfers the driver is working on is defined in */usr/sys/h/iobuf.h*; the header for the buffer associated with the current transfer is defined in */usr/sys/h/buf.h*. Standard character I/O device drivers use the per device "tty" structure (defined in */usr/sys/h/tty.h)* to pass information about the I/O request.

### 11.2.6 Interrupt Routine Rules

An interrupt routine operates in a more restricted environment than a task time routine, since it cannot make any assumptions about the state of the system or about the presence of particular user processes or user data in system memory. The relationship between the scope of task time and interrupt time routines is illustrated in the figure below:

```
                 TASK              |      INTERRUPT
                 TIME              |        TIME
+-------------+                    |
| User Program |                   |
+-------------+                    |
                                   |
                                   |
                                   |
+------------------------+         |    +---------+
|        u area          |         |    |         |
|------------------------|         |    | Driver  |
|  Kernel        |Drivers|         |    |Interrupt|
|                |       |         |    |Routines |
+------------------------+         |    +---------+
                                   |
                                   |
```

**Task and Interrupt Time**

The key things to remember are that the user process is mapped into memory, and its u space is mapped into the kernel's address space only at task time. Task time processing occurs whenever the user program code itself is executing (user mode) or the operating system is executing and performing services for the program (system mode).

It cannot be assumed that the u area is mapped into memory during the execution of an interrupt routine. No interrupt routine, nor any routine that may be called at interrupt time, may make any reference to user memory, the u area, or nonstatic memory locations. This means that the task time portion of the driver must not try to pass addresses of its frame variables and buffers to devices and interrupt service routines. Those locations are valid only when that individual user process is executing.

## 11.3 Kernel Support Routines

This section describes the routines that the kernel provides for device driver use.

### 11.3.1 in(), out(), inb(), and outb()

This section describes the routines used to interface to the registers that access and control a particular device. These registers may reside either in main memory (memory mapped) or in I/O space. There are four routines that provide a portable interface to the registers. These routines are described as follows:

**in (port) word**

**Purpose:** This routine returns the value of the word specified by the given port or register address.

**Parameters:** *port* is an integer value that specifies the address of the desired word.

*word* is an integer specifying the value of the returned word.

**Result:** The value of *word* is returned.

**Example:** To read the status of a word register at address 20 (hex), you may use the following lines of code:

```
int	val;
val = in(0x20);
```

**inb (port) byte**

**Purpose:** This routine returns the value of the byte specified by the given port or register address.

**Parameters:** *port* is an integer value that specifies the address of the desired byte.

*byte* is a byte specifying the value of the returned byte.

**Result:** The value of *byte* is returned.

**out (port, value)**

**Purpose:** This routine sets the word at the specified address to the specified value.

**Parameters:** *port* is an integer value that specifies the address of the word.

*value* is the integer value that the word will be set to.

**Result:** The word at the specified address is set to the specified value.

**outb (port, value)**

**Purpose:** This routine sets the byte at the specified address to the specified value.

**Parameters:** *port* is an integer value that specifies the address of the byte.

*value* is the byte value that the byte will be set to.

**Result:** The byte at the specified address is set to the specified value.

### 11.3.2 spl5() and splx()

This section describes the routines used to enable and disable interrupts during task time processing.

**spl5 () level**

**Purpose:** This routine may be called if interrupts should not be acknowledged during task time processing. It disables all disk and character I/O interrupts, and returns the pre-empted interrupted level. This value is used when restoring interrupts with the *splx()* routine.

**Parameters:** *level* is an integer value that specifies the interrupt level pre-empted by this routine.

**Result:** The value of the pre-empted interrupt level is returned.

**splx (oldspl)**

**Purpose:** This routine takes the return value of the *spl5()* routine and enables the interrupt levels that were accepted before the call to *spl5()*. Calls to *spl5()* and *splx()* nest correctly.

**Parameters:** *oldspl* is an integer value specifying the level of interrupts that were disabled by *spl5*.

**Example:** To restrict interrupts during critical device driver processing, you may use the following lines of code:

```
int x;
x = spl5();
/* do uninterruptable work */
splx(x);
```

### 11.3.3 sleep() and wakeup()

This section describes the routines used to suspend and reawaken requests that cannot be serviced immediately. For example, a device driver may receive a write request when the output buffer is full. In this case, the requesting process can suspend itself by calling *sleep()*. When the condition is alleviated, the suspended process is awakened in either of two ways: some other process may awaken the suspended process by calling *wakeup()* or it can be awakened by a signal.

**sleep (chan, pri)**

**Purpose:** This routine suspends a requesting process when one of the conditions required to execute the process cannot be met. This routine should never be called at interrupt time.

**Parameters:** *chan* is a unique number that identifies the sleeping process. The convention for generating this unique number is to use the address of some data structure the device drivers uses. Since no data structures will have the same address, uniqueness is guaranteed.

*pri* is an integer value that determines the priority of the process when it awakens. If a process goes to sleep at a priority lower than manifest constant PZERO, the sleep will not be broken by a signal. Typically, the priority is below PZERO if the condition is likely to disappear almost immediately, and it is above PZERO otherwise.

**wakeup (chan)**

**Purpose:** This routine wakes up a process(es) that has (have) been suspended by the *sleep()* routine. All the processes that have called *sleep()* with the unique number specified are awakened. When a process is awakened, the call to *sleep()* returns, and the process should check that the reason for going to sleep has disappeared.

**Parameters:** *chan* is a unique number that identifies the sleeping process to be awakened. The convention for generating this unique number is to use the address of some data structure the device drivers uses. Since no data structures will have the same address, uniqueness is guaranteed.

## 11.3.4 timeout()

This section describes the routine used to schedule a call to a routine at some later time.

**timeout (function, arg, tim)**

**Purpose:** This routine allows a function to be called at a scheduled time in the future.

**Parameters:** *function* is an integer value specifying the function to be called.

*arg* is the argument to the function being called.

*tim* is an integer value specifying the number of clock ticks that should elapse before the call.

**Example:** This routine can be used, along with *sleep()* and *wakeup()* to provide "busy waiting." The following code sample illustrates this:

```
#define PERIOD          HZ/10 /* 1/10 second */
#define BUSYPRI         (PZERO -1)   /* somewhat arbitrary */
int stopwait();
int status;

int busywait()  /* wait until status is non-zero */
{
        while (status == 0) {
                timeout(stopwait, 0, PERIOD);
                sleep(&status, BUSYPRI);
        }
}

int stopwait()
{
        wakeup(&status);
}
```

---

*WARNING*

A driver should never loop while waiting for a status change unless the delay involved is shorter than 100 microseconds.

---

### 11.3.5 copyio ()

This section describes the routine used to copy bytes to and from specific locations in the kernel.

**copyio (addr, faddr, cnt, mapping) error**

**Purpose:** This routine can be used to copy bytes to/from a physical address (i.e., buffer address) in the kernel or to/from a long address (i.e., user data pointer) in the kernel.

**Parameters:** *addr* is a long value that specifies the physical kernel address to which or from which the data is to be transferred.

*faddr* is a character value that specifies the segment and offset of the user address to which or from which the data is to be transferred.

*cnt* is an integer value that specifies the number of bytes of data to transfer.

*mapping* is an integer that designates the direction of the transfer. The possible mapping values are defined in *user.h* and listed below:

| | |
|---|---|
| U_WUD | transfer from user data to a kernel data (buffer) |
| U_RUD | transfer from kernel data (buffer) to user data |
| U_WUI | transfer from user text to a kernel data (buffer) |
| U_RUI | transfer from kernel data (buffer) to user text |
| U_WKD | transfer from kernel data to file (buffer) |
| U_RKD | transfer from file (buffer) to kernel data |

| | |
|---|---|
| U_READ | means copy from addr to faddr |
| U_WRITE | means copy from faddr to addr |

**Result:** If successful, this routine performs the specified data transfer; otherwise, it returns -1.

**Example:** The *ioctl* interface to a driver actually has two calling sequences:

```
1)        ioctl (fd, cmd, arg)
    int fd, cmd, arg;

2)        ioctl (fd, cmd, arg)
    int fd, cmd;
    int *arg;
```

In the kernel, the *ioctl* interface is translated into the device specific call shown below:

```
xxioctl (dev, cmd, arg)
```

```
int dev, cmd;
faddr_t arg;
If "arg" is a pointer to a data structure, you may copy
your data in/out using the copyio routine as shown
below:

struct foo dst;
/* copy from arg to dst */
if ( copyio ((caddr_t) &dst, arg, sizeof(foo), U_WUD) = = -1 ) {
        u.uerror = EFAULT;
        return;
}
```

**Note:** The file named */usr/sys/h/param.h* defines several macros that are useful for converting addresses from one type to another. These macros include:

| | |
|---|---|
| ftoseg(x) - | converts x from an faddr_t to a segment (selector number |
| ftoof(x) - | converts x from an faddr_t to an offset |
| sotofar(seg,off) - | converts a segment, offset pair into an faddr_t |
| ptok(x) - | converts a physical address to a kernel logical address |
| ktop(x) - | converts a kernel logical address to a physical address |

## 11.3.6 Version 7/System 5 Compatibility Issues

This section describes some of the changes between Version 7 UNIX and System 5 of UNIX that affect the device driver interface.

### Device Numbers

In Version 7 of UNIX, the *dev* parameter passed to the *open(), close(), read(), write(), and ioctl()* driver routines included the major and minor device numbers. In System 3 and System 5, only the minor device number is passed in the *dev* parameter. This means it is no longer necessary for all device drivers to mask out the major device number before checking the minor device number.

### iomove ()

Some Version 7 device drivers used a routine called *iomove*() to copy to or from user space. The *iomove*() routine does not exist in System 3 and System 5; however, adding the code shown below will provide most of the same capability:

```
#include "../h/param.h"
#include "../h/dir.h"
#include "../h/user.h"
/*
 * iomove - equivalent to the V7 version except we don't provide
 *                 any of the standard segflg machinations for writing
 *                 to instruction space
 *         NOTE:  u.ubase is an faddr_t
 */

iomove(cp, cnt, flag)
caddr_t cp;
register int cnt;
int flag;
{

        register int ret_val;

        if (cnt == 0)
                return;                         /* Nothing to do! */

        if( flag == B_WRITE )
                ret_val = copyio((caddr_t)cp, u.ubase, cnt, U_WUD);
        else
                ret_val = copyio((caddr_t)cp, u.ubase, cnt, U_RUD);
        if( ret_val == -1 ) {
                u.uerror = EFAULT;
                return;
        }
        u.ubase += cnt;
        u.ucount -= cnt;
        u.uoffset += cnt;
}
```

## 11.4 Parameter Passing to Device Drivers

The task time portion of the device driver has access to the user's u area, since this is mapped into kernel address space. The kernel routines that process the user process' I/O request place information describing the request into the process' u area. The parameters passed in the u area are:

| | |
|---|---|
| u.ubase - | address in user data to read/write data for transfer |
| u.ucount - | the number of bytes to transfer |
| u.uoffset - | the start address within the file for transfer |
| u.usegflg - | indicates the direction of the transfer |

Refer to the */usr/sys/h/user.h* file for the values to use for u.usegflg. In addition to the parameters passed in the u area, the kernel I/O routines pass the major and minor device numbers as a parameter to the driver when it is called. Thus the driver has all the information it needs to perform the request: the target device, the size of the data transfer, the starting address on the device, and the address in the process' data.

Only device drivers that do not use standard character and block I/O interfaces in the kernel need examine the parameters in the u area. Kernel routines that provide these standard interfaces have done the work of converting the values passed in the u area into values that the driver expects. In the case of the standard block I/O interface, these parameters are set in the buffer header describing the data transfer. Refer to Section 11.7, "Device Drivers for Block Devices," for more information on using the buffer header information to set up a block data transfer.

Device drivers using the standard character I/O interface use the clist buffering scheme and the routines that manipulate the clist to effect the data transfers. Refer to Section 11.6, "Device Drivers for Character Devices," for more information on using clists and the character I/O interface routines.

## 11.5 Naming Conventions

There is a naming convention for all driver routines called by the kernel, and for some driver variables. Each driver uses a unique two-to-four character prefix to identify its routines. For example, a hard disk driver might use the prefix "hd" In the following sections, the prefix used is "xx"

## 11.6 Device Drivers for Character Devices

This section describes XENIX character device drivers. Character devices conform to the XENIX file model; their data consists of a stream of bytes delimited only by the beginning and end of file. The XENIX system provides programs with direct access to devices through the special device files described in Section 5.1.5, "Special Device Files."

Most character device drivers in XENIX should be designed around the special requirements of terminal devices. There are facilities provided for programming functions on input and output (character erase, line kill, tab functions, etc.), and for setting line options such as speed. Other character-oriented devices such as lineprinters use the same program interface as terminals, but with a different driver.

The character device drivers for slow devices use a data buffering mechanism known as a character list or "clist." Clists are used for transferring relatively small amounts of data between the driver and the user program, and are described in more detail in Section 5.6.3, "Character List Architecture."

### 11.6.1 Character Device Driver Routines

The task time portion of the character device driver is called when a user process requests a data transfer to or from a device under the control of the driver. The system determines this from the major device number of the device with which the user wishes to do I/O. The driver's job is to take the user process' requests, check the parameters supplied, and set up the necessary information for the device interrupt routine to perform the I/O.

In the case of a write to a slow device (that is, one using *clists),* the driver copies the data from user space into the output *clist* for the device. In the case of direct I/O between the device and user memory (for example, magnetic tapes), the driver simply sets up the I/O request. The routines that provide the interface between the kernel and character device drivers are described as follows (xx is a mnemonic that refers to the device type).

**xxinit ()**

> **Purpose:** This routine is called to initialize the device when XENIX is first booted. If present, it is called indirectly through the *dinitsw* table defined in the kernel configuration file *(/usr/sys/conf/c.c)*.

**xxopen (dev, flag)**

> **Purpose:** This routine is called each time the device is opened. It is the responsibility of this routine to prepare the device for the I/O transfers, and perform any error or protection checking.
>
> **Parameters:** *dev* is an integer that specifies the minor number of the device.
>
> *flag* is the *oflag* argument that was passed to the open system call.

**xxclose (dev, flag)**

**Purpose:** This routine is called on the last close on a device. It is responsible for any cleanup that may be required, such as disabling interrupts, clearing device registers, and so on.

**Parameters:** *dev* is an integer that specifies the minor number of the device.

*flag* is the *oflag* argument passed to the last open system call.

**xxstart ()**

**Purpose:** If the task time portion of the driver detects that the device is idle, this routine may be called to start it. This routine is often called by both task time and interrupt time parts of the driver. It checks whether the device is ready to accept another transfer request, and if so, starts it up, usually by sending it a control word. *xxstart*() is not used by device drivers that control tty devices.

**xxintr (vec_num)**

**Purpose:** This routine is called by the kernel when the device issues an interrupt. Since the interrupt typically signals completion of a data transfer, the interrupt routine must determine the appropriate action; perhaps taking the received character and placing it in the input buffer, or removing the next character from the output buffer and starting the transmission.

**Parameters:** *vec_num* is an integer that specifies the interrupt vector number.

**xxread (dev)**

**Purpose:** This routine is called when a program makes a read system call. Its responsibility is to transfer data to the user's address space. A subroutine is available to transfer one character at a time to the user: *cpass ()*. This subroutine returns a -1 when there are no more characters to be transferred.

**Parameters:** *dev* is an integer that specifies the minor number of the device.

**xxwrite (dev)**

**Purpose:** This routine is called when a program makes a write system call. Its responsibility is to transfer data from the user's address space. A subroutine is available to transfer one character at a time from the user: *passc()*. This subroutine returns a -1 when there are no more characters to be transferred.

**Parameters:** *dev* is an integer that specifies the minor number of the device.

**xxproc (tp, cmd)**

**Purpose:** This routine is called to perform output character expansion, output characters, halt or restart character output and, in general, effect the desired change in the output.

**Parameters:** *tp* specifies the tty value of the device.

*cmd* specifies the process to be performed. The sample tty driver in Chapter 12, "Sample Device Drivers," documents the list of "cmd" argument values that *xxproc()* can expect.

**xxioctl (dev, cmd, arg, mode)**

**Purpose:** This routine is called by the kernel when a user process makes an *ioctl()* system call for the specified device. It performs hardware dependent functions such as setting the data rate on a character device.

**Parameters:** *dev* is an integer that speifies the minor device number of the device.

*cmd* is an integer that specifies the command passed to the system call.

*arg* specifies the argument passed to the system call.

*mode* specifies the flags passed on the open system call for the device.

## Character List Routines

There is a pool of small buffers called character lists, "clists" in the kernel. A clist structure is the head of a linked list queue of characters. The elements in the linked list are called "cblocks"; each cblock can hold a small number of characters. These are used for buffering low-speed character devices. The primary use of the clist buffers is for terminal devices that must interface with the common terminal interface. Refer to Section 11.6.3, "Character List Architecture," for further information on clists.

A driver that wishes to use the clist buffer mechanism must declare a queue header of type clist. If both input and output are buffered, the driver will need two headers. There are six routines that the driver can use to manipulate clist buffers. These routines are described below:

**getc (cp)**

**Purpose:** This routine moves one character from the clist buffer for each call.

**Parameters:** *cp* specifies the clist buffer from which characters are moved.

**Result:** This routine returns the next character in the buffer, or -1 if the buffer is empty.

**putc (c, cp)**

**Purpose:** This routine moves one character to the clist buffer for each call.

**Parameters:** *c* is an integer that specifies the character to be moved.

*cp* specifies the clist buffer to which the character is moved.

**Result:** This routine places the specified character in the buffer, or returns -1 if there is no free space.

**getcb (cp) cbp**

**Purpose:** This routine moves one cblock from the clist buffer for each call.

**Parameters:** *cp* specifies the clist buffer from which the cblocks are moved.

*cbp* is a pointer to a cblock.

**Result:** This routine returns the next cblock in the buffer, or -1 if the buffer is empty.

**putcb (cbp, cp)**

**Purpose:** This routine moves one cblock to the clist buffer for each call.

**Parameters:** *cbp* is a pointer that specifies the cblock to be moved.

*cp* is a pointer that specifies the clist buffer to which the cblock is moved.

**Result:** This routine places the specified cblock in the buffer, or returns -1 if there is no free space.

**getcf () cbp**

**Purpose:** This routine takes a cblock from the freelist, and returns a pointer to it.

**Parameters:** *cbp* is a pointer to a cblock.

**putcf cbp**

**Purpose:** This routine puts the specified cblock onto the freelist.

**Parameters:** *cbp* is a pointer to a cblock.

**Notes:** All the cblocks not currently used are kept on a list of free memory blocks. Since there are a limited number of cblocks in the system, each driver must be judicious in determining how many cblocks are used for buffering input and output.

For output buffering, the driver usually follows a "high and low water mark" convention. The driver accepts and queues requests from the user process until the buffer has reached its high water mark. At that point, the requesting processes are suspended via *sleep*(). When the buffer has drained below the low-water mark, the suspended processes are awakened, and can fill the buffer again.

For input buffering, the driver usually buffers the data up to some limit. When this limit is reached, data is discarded to make room for the more recent data.

**putchar (c)**

**Purpose:** This routine is used for printing error and system crash messages when the device driver is used to handle the console. It puts one character on the console, doing a "busy wait" rather than depending on interrupts.

**Parameters:** *c* is the character to be printed on the console.

### Line Discipline Routines

If a serial device is to be used as an interactive terminal, it must support various functions such as character and line erase, echoing, and buffered input. The code needed to perform each of these functions has been abstracted into a set of routines that roughly corresponds to the character device function. Each of these sets is called a "line discipline". One standard line discipline is provided by default. Each of the routines is called through the *linesw* table initialized in */usr/sys/conf/c.c*; each entry in this table represents one line discipline, and has entries for eight functions.

The *l_open*() routine should be called on the first open of a device. The *l_close*() routine should be called on the last close of the device. The *l_read*() and *l_write*() routines are called by the drivers read and write routines, to pass characters to and from the calling process. The *l_input*() routine is called to buffer an incoming character. The *l_output*(), *l_ioctl*(), and *l_mdmint*() routines are currently unused.

## 11.6.2 Interrupt Routines for Character Device Drivers

The device interrupt routine is entered whenever one of its devices raises an interrupt. Note that in general one driver may control several devices, but that all interrupts are vectored through a single function entry point, usually called *xxintr*(), where *xx* is a mnemonic that refers to the device type (see Section 5.5, "Naming Conventions"). It is the driver's responsibility to decide which device caused the interrupt.

When a device raises an interrupt, it generally makes available some status information to indicate the reason for the interrupt. The driver interrupt routine decodes this information. If it indicates a transfer just completed, the *wakeup*() routine will alert any process waiting for the transfer to complete. It then makes a check to see if the device is idle, and if so looks for more work to start up. Thus in the case of output to a terminal, the interrupt routine looks for more work in the clists each time a transfer completes.

## 11.6.3 Character List Architecture

The character lists (clists) provide a general character buffering system for use by character device drivers. The mechanism is designed for buffering small amounts of data from relatively slow devices, particularly terminals.

The XENIX kernel has a pool of character lists. Each driver that wishes to use the clist mechanism declares a static buffer header that points to the first clist buffer. Each buffer contains a pointer to the next buffer, forming a singly linked list.

The kernel provides the *getc*() and *putc*() routines (described above) for putting characters into a clist, and removing characters from a clist. These routines should be used by all drivers using clists. Note that the routines are not the same as the Standard I/O Library routines of the same name.

The static buffer header for each clist contains three fields: a count of the number of characters in the list, a pointer to the first character in the list, and a pointer to the last character. The clist buffers form a single linked list as shown below:

```
                     +------+    +------+     +-----+
struct {             | next |--->| next |---->|  0  |
int c_cc;            +------+    +------+     +-----+
char *c_cf; ---->|      |    |      |     |     |
char *c_cl; --+      |      |    |      |     |     |
} clist;      |      |chars |    |chars |     |chars|
              |      |      |    |      |     |     |
              |      |      |    |      | +-->|     |
              |      |      |    |      | |   |     |
              |      +------+    +------+ |   +-----+
              |                           |
              +---------------------------+
```

**Character List Buffers**

There is a protocol defined for use of the clists to prevent a particular process or driver from consuming all available resources. Two constants for the clist high and low water marks are defined in the file named *tty.h*. A process is allowed to issue write requests until the corresponding clist hits the high water mark. The process is then suspended and I/O performed. When the list reaches the low water mark, the process is awakened. A similar protocol is used for read requests.

## 11.6.4 Terminal Device Drivers

The terminal device drivers use clists extensively. For each terminal line (each minor device number), the driver declares static clist headers for three clists. These clists are the "raw queue," the "canonical queue," and the "output queue."

When a process writes data to a terminal device, the task time part of the driver puts the data into the output queue, and the interrupt routine transfers it from the queue to the device.

When a process requests a read of data from the terminal, the situation is slightly more complicated. This is because XENIX provides for some processing of characters on input, at the option of the requesting process.

For example, in normal input the backspace key is interpreted as "delete the last character input," and the line kill character means "forget the whole current line." Certain special characters (such as backspace) have to be treated in context; that is, they depend upon surrounding characters. To handle this, XENIX drivers use two queues for incoming data.

The two queues are the raw queue and the canonical queue. Data received by the interrupt routine is placed in the raw queue with no data processing. At task time, the driver decides how much processing to do. The user process has the option of requesting raw input, where it receives data directly from the raw queue. Cooked (the opposite of raw) input refers to the input after processing for erase, line kill, delete, and other special treatment. In this case, a task time routine, *canon*(), is used to transfer data from the raw queue to the canonical queue. This performs backspace and line kill functions, according to the options set by the process using the *ioctl*(2) system call.

The basic flow of data through the system during terminal I/O is shown in the diagram below:

```
XENIX  | DRIVER    |                          TASK TIME #INTERRUPT
KERNEL |           |                                    # TIME
       |           |                                    #
   <---|-<--    <- |---------< ttread()                 #
Read() | xxread()  |                |                   #
system |           |   +---->----+----<----+            #
call   |           |   |                   |            #
       |           | +-----+            +-----+         #
       |           | |     |            |     |         #
       |           | |canon|            | raw |         #
       |           | |queue|<- canon() <-|queue|<-#receive
       |           | |     |            |     |   #routine
       |           | |     |            |     |   #
       |           | +-----+            +-----+   #
       |           |                              #
Write()|           |                              #
system |           |                 +------+     #
call   |           |                 |      |     #
   --->|->--    ->-|---> ttwrite() -->|output|--->#transmit
       | xxwrite() |                 |queue |     #routine
       |           |                 |      |     #
       |           |                 |      |     #
       |           |                 +------+     #
```

**Data Flow For Terminal Device Drivers**

There are two slight complications to the scheme presented in the diagram above. These are output character expansion, and input character echo. Output expansion occurs for a few special characters. In cooked mode, tabs may be expanded into spaces, and the newline character is mapped into carriage return plus line feed. There is a facility for producing escape sequences for uppercase terminals, and delay periods for certain characters on slow terminals. Note that all these are simple expansions, or mapping single characters, and so do not require a second list, as is the case for input. Instead all the expansion is performed by the *xxproc*() routine before placing the characters in the output clist.

Character echo is a user process option required by most processes. With this, all input characters are immediately echoed to the output stream, without waiting for the user process to be scheduled. Character expansion is performed for echoed characters, as for regular output. Character echo takes place at interrupt time, so that a user typing at a terminal gets fast echo, regardless of whether his program is in memory and running, or swapped out on disk.

### 11.6.5 Other Character Devices

There are three character devices commonly found on XENIX systems: terminals, lineprinters, and magnetic tape drivers. Terminals receive a lot of special attention in the XENIX system. Lineprinters and magnetic tape tend to use existing kernel facilities, with little special handling.

#### Lineprinters

These are usually relatively slow character-oriented devices. The drivers use the clist mechanism for buffering data. However a lineprinter driver is generally simpler than a terminal driver because there is less processing of output characters to do, and no input.

#### Magnetic Tape Drivers

Magnetic tape is a special case. The data is arranged on the physical medium in blocks, as on a disk. However, it is almost always accessed serially. Furthermore, there is generally only one program accessing a tape drive at a time. Thus, the elaborate kernel buffer management scheme in XENIX (which is designed to optimize disk access when several processes are making simultaneous requests to different parts of the same disk) is not applicable to tapes. Neither is the clist mechanism appropriate, because of the large amount of data involved.

Usually tape drivers provide two interfaces, a block and a character interface. The character interface is used for raw, or physical, I/O directly between the device and the user process' address space. The block interface makes use of the XENIX kernel buffer pool and buffer manipulation routines to store data in transit between device and process. Refer to Section 11.7.1, "Character Interface to Block Devices," for information on providing the facility for raw I/O.

## 11.7 Device Drivers for Block Devices

Block devices are those that must be addressed in terms of large blocks of data, rather than individual bytes. Disks fall into this category, as do some magnetic tape systems. XENIX file systems always reside on block devices. However, block devices do not have to be used in this way.

Unlike the case with character devices, a block I/O transfer request is not a private transaction between a driver and a user process. The XENIX kernel provides a comprehensive buffer management scheme which is used by block device drivers.

The XENIX kernel maintains a pool of buffers, and keeps track of what data is in them, and whether the block is dirty (i.e., has been modified and therefore needs to be written out to disk). When a user process issues a transfer request to a block device, the kernel buffer routines check the buffer pool to see if the data is already in memory. If not, a request is passed to the driver to get the data. All the driver ever sees are fixed size requests (BSIZE bytes long) coming in from one source. This is regardless of the size of the process' I/O request. Large requests are broken down into BSIZE blocks, and handled individually, since some may be in memory, and some not.

When a process issues a read request, this generally translates into one or more disk blocks. The kernel checks which of these is already in memory, and requests that the driver get the rest. The data from each buffer filled by the driver is copied into the process' memory by the kernel. In the case of a write request, the kernel copies the data from the user process' memory into the buffer pool. If there are insufficient free buffers, the kernel will have the driver write some out to disk, using a selection algorithm designed to reduce disk traffic. When all the data is copied out of user space, the kernel can reschedule the process. Note that all the data may not yet be out on disk; some may be in memory buffers and marked as needing to be written out at some later time.

### 11.7.1 Character Interface to Block Devices

Sometimes block device drivers provide a character I/O interface as well as one for block I/O. In this case, a separate special device file can be created to access the device through the character interface. To construct a character I/O interface to a block device, use the utility **mknod**(C) described in the XENIX *Reference Manual* to create a character special device file that has the same major and minor number as the block special file for this device. The block device driver must provide the routines *xxread*() and *xxwrite*() described below to implement character I/O.

When a block device is accessed through a character interface, data transfer takes place directly between the device and the process' memory space. There is no intermediate buffering in the kernel buffer pool or the clists. The driver receives the request exactly as the process sent it, for whatever size was specified. There is no kernel support to break the job into BSIZE blocks. This type of data transfer is referred to as physical (or raw) I/O. It has some advantages for certain types of programs.

Programs that need to read or write an entire device can usually do this more efficiently through the character interface since the device can be accessed sequentially, and large transfers can be used. There is also less copying of data between buffers than is used in the block interface. Thus disk backup programs, or utilities that copy entire volumes, typically operate through this interface.

The cost of this extra efficiency is that the process has to be locked in memory during the transfer, since the driver has to know where to buffer the data. The routine *physio()* called by the *xxread* and *xxwrite* driver routines handles locking the process in core for the duration of the data transfer.

## 11.7.2 Block Device Driver Routines

A block device appears to the kernel as a randomly addressable set of records of size BSIZE, where BSIZE is a manifest constant defined in the *param.h* file. The XENIX kernel inserts a layer of buffering software between user requests for block devices and the device driver. This buffering improves system performance by acting as a cache, allowing read ahead and write behind on block devices.

Each buffer in the cache contains an area for BSIZE bytes of data and has associated with it a header of type struct *buf* which contains information about the data in the buffer. When an I/O request is passed to the task time portion of the block device driver, all of the information needed to handle the data transfer request has been stored in the buffer header. This information includes the disk address, and whether a read or a write is to be done. The file */usr/sys/h/buf.h* describes the fields in the buffer header. The fields most relevant to the device driver are:

| | |
|---|---|
| b_dev | - the major and minor numbers of the device |
| b_bcount | - the number of bytes to transfer |
| b_paddr | - the physical address of the buffer |
| b_blkno | - the block number on the device |
| b_error | - set if an error occurred during the transfer |

The driver validates the transfer parameters in the buffer header, and then queues the buffer on a doubly linked list of pending requests. In each block device driver, this chain of requests is pointed to by a header of type struct *iobuf* named *xxtab*. The file */usr/sys/h/iobuf.h* describes the fields in the request queue header. The requests in the list are kept sorted using the *disksort*() routine. The device interrupt routine takes its work from this list.

When a transfer request is placed in the list, the process making the request sleeps until the transfer is completed. When the process is awakened, the driver checks the status information from the device interrupt routine, and if the transfer completed successfully, returns a success code to the kernel. The kernel buffer routines are responsible for correlating the completion of an individual buffer transfer with particular user process requests.

The interface between the kernel and the block device driver consists of the routines described in the following paragraphs.

**xxinit ()**

**Purpose:** This routine is called to initalize the device when XENIX is first booted. If present, it is called indirectly through the *dinitsw* table defined in the kernel configuration file *(/usr/sys/conf/c.c)*.

**xxopen (dev, flag)**

**Purpose:** This routine is called each time the device is opened. It is the responsibility of this routine to intialize the device, and perform any error or protection checking.

**Parameters:** *dev* is an integer that specifies the device number.

*flag* is the *oflag* argument that was passed to the open system call.

**xxclose (dev, flag)**

**Purpose:** This routine is called on the last close on a device. It is responsible for any cleanup that may be required, such as disabling interrupts, clearing device registers, ejecting media, and so on.

**Parameters:** *dev* is specifies the device number of the device being closed.

*flag* is the *oflag* argument that was passed to the last open system call.

**xxstrategy (bp)**

**Purpose:** This routine is called by the kernel to queue an I/O request. It must make sure the request is for a valid block, and then insert the request into the queue. Usually the driver will call *disksort*() to insert the request into the queue. The *disksort*() routine takes two arguments: a pointer to the head of the queue, and a pointer to the buffer header to be inserted.

**Parameters:** *bp* is a pointer to a buffer header.

**xxstart ()**

**Purpose:** If the task time portion of the driver detects that the device is idle, this routine may start it. It is often called by both task time and interrupt time parts of the driver. It checks whether the device is ready to accept another transfer request, and if so, starts it up, usually by sending it a control word.

**xxintr (vec_num)**

**Purpose:** This routine is called whenever the device issues an interrupt. Depending on the meaning of the interrupt, it may mark the current request as complete, start the next request, continue the current request, or retry a failed operation. The routine examines the device status information, and determines whether the request was successful. The block buffer header is updated to reflect this. The interrupt routine checks to see if the device is idle, and if so, starts it up before exiting.

**Parameters:** *vec_num* is an integer that specifies the interrupt vector number.

## xxread (dev)

**Purpose:** The only action taken by this routine is to call the *physio()* routine with the appropriate arguments.

**Parameters:** *dev* specifies the device number of the device.

**Note:** Often a block device driver will provide a character device driver interface so that the device can be accessed without going through the structuring and buffering imposed by the kernel's block device interface. For example, a program might wish to read magnetic tape records of arbitrary size, or read large portions of a disk directly. When a block device is referenced through the character device interface, it is called raw I/O to emphasize the unstructured nature of the action. Adding the character device interface to a block device requires the *xxread*() and *xxwrite*() routines.

## xxwrite (dev)

**Purpose:** The only action taken by this routine is to call the *physio()* routine with appropriate arguments.

**Parameters:** *dev* specifies the device number of the device.

**Note:** See Note for *xxread()* routine.

**physio (bs, bp, dev, flag)**

**Purpose:** This routine provides the raw I/O interface for block device drivers. It validates the request, builds a buffer header, locks the process in core, and calls the strategy routine to queue the request.

**Parameters:** *bs* is a pointer to the strategy routine for the block device.

*bp* is a pointer to the buffer header describing the request to be filled.

*dev* is the device number of the device.

*flag* specifies the call is a read or write operation.

**xxioctl (dev, cmd, arg, mode)**

**Purpose:** This routine is called by the kernel when a user process makes an *ioctl*() system call for the specified device. It performs hardware dependent functions such as parking the heads of a hard disk, setting a variable to indicate that the driver is to format the disk, or telling the driver to eject the media when the close routine is called.

**Parameters:** *dev* specifies the minor number of the device.

*cmd* specifies the command that was passed to the *ioctl()* system call.

*arg* specifies the argument that was passed to the *ioctl()* system call.

*mode* specifies the flags that were set on the *open()* system call for the specified device.

## 11.8 Sharing Interrupt Vectors

I/O devices may only share interrupt vectors if there is a way to poll each device using the shared vector to determine whether that device has posted an interrupt.

If there are two devices "aa" and "bb" that share interrupt level 3, the code in the *c.c* file should be as follows:

```
vector3(level)
int level;
{
        aaintr(level);
        bbintr(level);
}


int (*vecintsw[])() =
{
        clock,
        consintr,
        novec,
        vector3,
        novec,
        etc
        .
        .
        .
}
```

The interrupt routines *aaintr()* and *bbintr()* should have the following format:

```
xxintr(level)
int level;
{

        IF NOT MY INTERRUPT
                return;

        NORMAL INTERRUPT PROCESSING

}
```

## 11.9 Warnings

The following warnings will help you avoid problems when writing a device driver:

— Don't defer interrupts with *spl5*() calls any longer than necessary.

— Don't change the per process data in the u structure at interrupt time.

- Don't call *seterror*() or *sleep*() at interrupt time.
- Don't call *spl5*() at interrupt time.
- Make interrupt time processing as short as possible.
- Protect buffer and clist processing with *spl5*() calls.
- Avoid "busy waiting" whenever possible.
- Never use floating point arithmetic operations in device driver code.
- If any assembly language device driver sets the direction flag (using **std**), it must clear it (using **cld**) before returning.
- Keep the local (stack) data requirements for your driver very small.

# Chapter 12

# Sample Device Drivers

## 12.1 Introduction

This chapter provides sample device driver code for line printer, terminal, and hard disk drives. Each 50-line segment of code is followed by some general comments, which describe the routines used and explain key lines in the program. These key lines are identified by line number.

## 12.2 Sample Device Driver for Line Printer

```
/*
** lp- prototype line printer driver
*/
#include "../h/param.h"
#include "../h/dir.h"
#include "../h/a.out.h"
#include "../h/user.h"
#include "../h/file.h"
#include "../h/tty.h"

#define LPPRI       PZERO+5
#define LOWAT       50
#define HIWAT       150

/* register definitions */

#define RBASE       0x00            /* base address of registers */
#define RDATA       (RBASE + 0)     /* place character here */
#define RSTATUS     (RBASE + 1)     /* non zero means busy */
#define RCNTRL      (RBASE + 2)     /* write control here */

/* control definitions */
#define CRESET      0x01    /* initialize the interface */
#define CIENABL     0x02    /* +Interrupt enable */

/* flags definitions */
#define FIRST       0x01
#define ASLEEP      0x02
#define ACTIVE      0x04

struct clist lp_queue;
unsigned lp_flags = 0;

lpopen(dev)
int dev;
{
   if ( (lp_flags & FIRST) == 0 ) {
            lp_flags |= FIRST;
            outb(RCNTRL, CRESET);
   }
   outb(RCNTRL, CIENABL);
}

lpclose(dev)
int dev;
{
```

```
47   }
```

## Description of Device Driver for Line Printer

The device driver presented here is for a single parallel interface to a printer. It transfers characters one at a time, buffering the output from the user process through the use of character blocks (cblocks).

11: LPPRI is the priority at which a process sleeps when it needs to stop. Since the priority is greater than PZERO, a signal sent to the suspended process will awaken it.

12: LOWAT is the minimum number of characters in the buffer. If there are fewer than LOWAT characters in the buffer, a process that was suspended (because the buffer was full) can be restarted.

13: HIWAT is the maximum number of characters in the queue. If a process fills the buffer up to this point, it will be suspended via *sleep*() until the buffer has drained below LOWAT.

17-20: The device registers in this interface occupy a contiguous block of address, starting at RBASE, and running through RBASE+2. The data to be printed is placed in RDATA, one character at a time. Printer status can be read from RSTATUS, and the interface can be configured by writing into RCNTRL.

27-29: The flags defined in these lines are kept in the variable *lp_flags*. FIRST is set if the interface has been initialized. ASLEEP is set if a process is asleep waiting for the buffer to drain below LOWAT. ACTIVE is set if the printer is active.

31: *lp_queue* is the head of the linked list of cblocks that forms the output buffer.

32: *lp_flags* is the variable in which the flags mentioned above are kept.

## lpopen() - lines 34 to 42

The *lpopen*() routine is called when some process makes an *open*() system call on the special file that represents this driver. Its single argument, *dev* represents the minor number of the device. Since this driver supports only one device, the minor number is ignored.

37-39: If this is the first time (since XENIX was booted) that the device has been touched, the interface is initialized by setting the CRESET bit in the control register.

41: Interrupts from this device are enabled by setting the IENABL bit in the control register.

## lpclose() - lines 44 to 47

The *lpclose*() routine is called on the last close of the device; that is, when the current *close*() system call results in zero processes referencing the device. No action is taken.

```
lpwrite(dev)
int dev;
{
    register int c;
    int x;

    while ( (c = cpass()) >= 0 ) {
        x = spl5();
        while ( lp_queue.c_cc > HIWAT ) {
                lpstart();
                lp_flags |= ASLEEP;
                sleep(&lp_queue, LPPRI);
        }
        splx(x);
        putc(c, &lp_queue);
    }
    x = spl5();
    lpstart();
    splx(x);
}

lpstart()
{
    if ( lp_flags & ACTIVE )
        return; /* interrupt chain is keeping printer going */
    lp_flags |= ACTIVE;
    lpintr(0);
}


lpintr(vec)
int vec;
{
    int tmp;

    if ( (lp_flags & ACTIVE) == 0 )
        return;          /* ignore spurious interrupt */

    /* pass chars until busy */
    while ( inb(RSTATUS) == 0 && (tmp = getc(&lp_queue)) >= 0)
        outb(RDATA, tmp);

    /* wakeup the writer if necessary */
    if ( lp_queue.c_cc < LOWAT && lp_flags & ASLEEP ) {
        lp_flags &= ~ASLEEP;
        wakeup(&lp_queue);
    }

    /* wakeup writer if waiting for drain */
98      if ( lp_queue.c_cc <= 0 )
99              lp_flags &= ~ACTIVE;
100   }
```

## lpwrite() - lines 49 to 66

The *lpwrite*() routine is called to move the data from the user process to the output buffer. Code is defined as follows:

55: While there are still characters to be transferred, do what follows.

56-63: Raise the processor priority so the interrupt routine can't change the buffer. If the buffer is full, make sure the printer is running, note that the process is waiting, and put it to sleep. When the process wakes up, check to make sure the buffer has enough space, then go back to the old priority and put the character in the buffer.

65-66: Make sure the printer is running, by locking out interrupts and calling *lpstart*().

## lpstart() - lines 70 to 76

The *lpstart*() routine ensures that the printer is running. It's called twice from *lpwrite*(), and serves simply to avoid duplicate code. Code is defined as follows:

72-75: If the printer is running, just return; otherwise, mark it ACTIVE, and call *lpintr*() to start the transfer of characters.

## lpintr() - lines 79 to 100

The *lpintr*() routine is called from two places: *lpstart*(), and from the kernel interrupt handling sequence when a device interrupt occurs. Code is defined as follows:

84-85: If *lpintr*() is called unexpectedly, or the driver doesn't have anything to do, it just returns.

88-89: While the printer indicates it can take more characters and the driver has characters to give it, the characters come from the buffer through *getc*(), and pass to the interface by writing to the data register.

92-94: If the buffer has fewer than LOWAT characters in it, and some process is asleep waiting for room, wake it up.

98-99: If the queue is empty, turn off the ACTIVE flag. Note that the interrupt that completes the transfer and empties the buffer is in some sense "spurious", since it will occur with the ACTIVE flag reset.

## 12.3 Sample Device Driver for Terminal

```
/*
** td- terminal device driver
*/
#include "../h/param.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/file.h"
#include "../h/tty.h"
#include "../h/conf.h"

/* registers */
#define RRDATA   0x01     /* received data */
#define RTDATA   0x02     /* transmitted data */
#define RSTATUS  0x03     /* status */
#define RCNTRL   0x04     /* control */
#define RIENABL  0x05     /* interrupt enable */
#define RSPEED   0x06     /* data rate */
#define RIIR     0x07     /* interrupt identification */

/* status register bits */
#define SRRDY    0x01     /* received data ready */
#define STRDY    0x02     /* transmitter ready */
#define SOERR    0x04     /* received data overrun */
#define SPERR    0x08     /* received data parity error */
#define SFERR    0x10     /* received data framing error */
#define SDSR     0x20     /* status of dsr (cd)*/
#define SCTS     0x40     /* status of clear to send */

/* control register */
#define CBITS5   0x00     /* five bit chars */
#define CBITS6   0x01     /* six bit chars */
#define CBITS7   0x02     /* seven bit chars */
#define CBITS8   0x03     /* eight bit chars */
#define CDTR     0x04     /* data terminal ready */
#define CRTS     0x08     /* request to send */
#define CSTOP2   0x10     /* two stop bits */
#define CPARITY  0x20     /* parity on */
#define CEVEN    0x40     /* even parity otherwise odd */
#define CBREAK   0x80     /* set xmitter to space */

/* interrupt enable */
#define EXMIT    0x01     /* transmitter ready */
#define ERECV    0x02     /* receiver ready */
#define EMS      0x04     /* modem status change */

/* interrupt ident */
```

```
47      #define  IRECV    0x01
48      #define  IXMIT    0x02
49      #define  IMS      0x04
50
50A     #define  NTDEVS    2
50B     #define  VECT0     3
50C     #define  VECT1     5
```

## Description of Device Driver for Terminal

This driver supports two serial terminals on a hypothetical UART type interface.

12-18: The interface for each line consists of seven registers. The values that would be defined here represent offsets from the base address, which is defined in line 72. The base address differs for each line. The data to be transmitted is placed one character at a time into the RTDATA register. Likewise, the received data is read one character at a time from the RRDATA register. The status of the UART can be determined by examining the contents of the RSTATUS register. The UART configuration is adjusted by changing the contents of the RCNTRL register. Interrupts are enabled or disabled by setting the bits in the RIENABL register. The data rate is set by changing the contents of the RSPEED register. Interrupts are identified by reading the bits in the RIIR register.

30-39: The two low order bits of the "control register" determine the length of the character sent. The next two bits control the data-terminal-ready and request-to-send lines of the interface. The next bit controls the number of stop bits, the next controls whether parity is generated, and the next controls whether generated parity is even or odd. Finally, the most significant bit forces the transmitter to continuous spacing if it is set.

42-44: The three low order bits of the "interrupt enable" register control whether the device generates interrupts under certain conditions. If bit 0 is set, an interrupt is generated every time the transmitter becomes ready for another character. If bit 1 is set, an interrupt is generated every time a character is received. If bit 2 is set, an interrupt is generated every time the data-set-ready line changes state.

47-49: After an interrupt, the value in the interrupt identification register will contain one of three values, indicating the reason for the interrupt.

```
/* data rates */
int td_speeds[] = {
        /* B0    */     0,
        /* B50   */     2304,
        /* B75   */     1536,
        /* B110  */     1047,
        /* B134  */     857,
        /* B150  */     768,
        /* B200  */     0,
        /* B300  */     384,
        /* B600  */     192,
        /* B1200 */     96,
        /* B1800 */     64,
        /* B2400 */     48,
        /* B4800 */     24,
        /* B9600 */     12,
        /* EXTA  */     6, /* 19.2k bps */
        /* EXTB  */     58 /* 2000 bps */
        };

struct tty td_tty[NTDEVS];
int td_addr[NTDEVS] = { 0x00, 0x10 };


tdopen(dev, flag)
int dev, flag;
{
        register struct tty *tp;
        int addr;
        extern tdproc();
        int     x;

        if ( dev >= NTDEVS ) {
                seterror(ENXIO);
                return;
        }
        tp = &td_tty[dev];
        addr = td_addr[dev];
        if( (tp->t_lflag & XCLUDE) && !suser() ) {
                seterror(EBUSY);
                return;
        }
        if ((tp->t_state&(ISOPEN|WOPEN)) == = 0) {
                ttinit(tp);
                tp->t_proc = tdproc;
                tdparam(dev);
        }
        x = spl5();
        if ( tp->t_cflag & CLOCAL || tdmodem(dev, TURNON))
```

```
100                 tp->t_state |= CARR_ON;
101         else
102                 tp->t_state &= ~CARR_ON;
```

52-69: The values to be loaded into the RSPEED register to get various data rates are defined here.

71: Each line must have a tty structure allocated for it.

72: Here, the base addresses of the registers are defined for each line.

## tdopen() - lines 75 to 110

The *tdopen*() routine is called whenever a process makes an *open*() system call on the special file corresponding to this driver. Code is defined as follows:

83-85: If the minor number indicates a device that doesn't exist, indicate the error, and return.

89-91: If the line is already open for exclusive use, and the current user is not the super-user, indicate the error and return.

93-96: If the line is not already open, initialize the tty structure via a call to *ttinit*(), set the value of the *proc* field in the tty structure, and configure the line by calling *tdparam*().

98: Defer interrupts so the interrupt routines cannot change the state while it is being examined.

99-102: If the line is not using modem control, or if it is not turning on the data-terminal-ready and request-to-send signals (which results in carrier-detect being asserted by the remote device), indicate that the carrier signal is present on this line. Otherwise, indicate that there is no carrier signal.

```
    if (!(flag&FNDELAY))
            while ((tp->t_state&CARR_ON)==0) {
                    tp->t_state |= WOPEN;
                    sleep((caddr_t)&tp->t_canq, TTIPRI);
            }
    (*linesw[tp->t_line].l_open)(tp);
    splx(x);
}

tdclose(dev)
{
    register struct tty *tp;

    tp = &td_tty[dev];
    (*linesw[tp->t_line].l_close)(tp);
    if (tp->t_cflag & HUPCL)
            tdmodem(dev, TURNOFF);
    tp->t_lflag &= ~XCLUDE;  /* turn off exclusive use bit */
    /* turn off interrupts */
    outb(td_addr[dev] + RIENABL, 0);
}

tdread(dev)
{
    (*linesw[tp->t_line].l_read)(&td_tty[dev]);
}

tdwrite(dev)
{

    (*linesw[tp->t_line].l_write)(&td_tty[dev]);
}

tdparam(dev)
{
    register int cflag;
    register int addr;
    register int temp, speed, x;

    addr = td_addr[dev];
    cflag = td_tty[dev].t_cflag;

    /* if speed is B0, turn line off */
    if ( (cflag & CBAUD) == B0){
            outb(addr + RCNTRL, inb(addr+RCNTRL) & ~CDTR & ~CRTS);
            return;
    }

```

103-106: If *open*() is supposed to wait for the carrier, wait until the carrier is present.

108: Call the *l_open* routine indirectly through the *linesw* table. This completes the work required for the current line discipline to open a line.

109: Allow further interrupts.

## tdclose() - lines 112 to 123

The *tdclose*() routine is called on the last close on a line.

117: Call the *close*() routine through the *linesw* table to do the work required by the current line discipline.

118-119: If the "hang up on last close" bit is set, drop the data-terminal-ready and request-to-send signals.

120: Reset the exclusive use bit.

122: To prevent spurious interrupts, disable all interrupts for this line.

## tdread() and tdwrite() - lines 125 to 134

Both of these routines simply call the relevant routine via the *linesw* table; the called routine performs the action appropriate for the current line discipline.

## tdparam() - lines 136 to 171

The *tdparam*() routine configures the line to the mode specified in the appropriate tty structure.

142-143: Get the base address and flags for the referenced line.

146-148: The speed B0 means "hang up the line."

```
    /* set up speed */
    outb( addr + RSPEED, td_speeds[ cflag & CBAUD ]);

    /* set up line control */
    temp = (cflag & CSIZE) >> 4; /* length */
    if ( cflag & CSTOPB )
            temp |= CSTOP2;
    if ( cflag & PARENB ) {
            temp |= CPARITY;
            if ( (cflag & PARODD) == 0)
                    temp |= CEVEN;
    }
    temp |= CDTR | CRTS;
    out( addr + RCNTRL, temp );

    /* setup interrupts */
    temp = EXMIT;
    if ( cflag & CREAD )
            temp |= ERECV;
    outb(addr + RIENABL, inb(RIENABL) | temp);
}

tdmodem(dev, cmd)
int dev, cmd;
{
    register int addr;

    addr = td_addr[dev];
    switch(cmd){
    case TURNON: /* enable modem interrupts, set DTR & RTS true */
            outb(addr + RIENABL, inb(addr+RIENABL) | EMS);
            outb(addr + RCNTRL, inb(addr+RCNTRL) | CDTR | CRTS );
            break;
    case TURNOFF: /* disable modem interrupts, reset DTR, RTS */
            outb(addr + RIENABL, inb(addr+RIENABL) & ~EMS);
            outb(addr + RCNTRL, inb(addr+RCNTRL) ~(CDTR | CRTS) );
            break;
    }
    return (inb(addr + RSTATUS) & SDSR);
}
#endif

tdintr(vec)
int vec;
{
    register int iir, dev, inter;

    switch( vec ) {
            case VECT0:
```

```
200                     dev = 0;
```

152: The remainder of the *tdparam*() routine simply loads the device registers with the correct values.

## tdmodem() - lines 173 to 190

The *tdmodem*() routine controls the data-terminal-ready and request-to-send line signals. Its return value indicates whether data-set-ready signal (carrier detect) is present for the line.

180-183: If *cmd* was TURNON, turn on modem interrupts, and assert data-terminal-ready and request-to-send.

184-187: If *cmd* was TURNOFF, disable modem interrupts, and drop data-terminal-ready and request-to-send.

189: Return a zero value if there is no data-set-ready on this line, otherwise return a non-zero value.

## tdintr() - lines 193 to 217

The *tdintr*() routine determines which line caused the interrupt and the reason for the interrupt, and calls the appropriate routine to handle the interrupt.

198-207: Different lines will result in different interrupt vectors being passed as the *tdintr*() routine's argument. Here, the minor number is determined from the interrupt vector that was passed to *tdintr*().

```
                        break;
                case VECT1:
                        dev = 1;
                        break;
                default:
                        printf("tdint: wrong level interrupt (%x)\n",vec);
                        return;
                }
        while( (iir = inb(td_addr[dev]+RIIR)) != 0) {
                if( (iir & IXMIT) != 0 )
                        tdxint(dev);
                if( (iir & IRECV) != 0 )
                        tdrint(dev);
                if( (iir & IMS) != 0 )
                        tdmint(dev);
        }
}

tdxint(dev)
{
        register struct tty *tp;
        register int addr;

        tp = &td_tty[dev];
        addr = td_addr[dev];
        if ( inb(addr + RSTATUS) & STRDY )
        {
                tp->t_state &= ~BUSY;
                if (tp->t_state & TTXON) {
                        outb(addr + RTDATA, CSTART);
                        tp->t_state &= ~TTXON;
                } else if (tp->t_state & TTXOFF) {
                        outb(addr + RTDATA, CSTOP);
                        tp->t_state &= ~TTXOFF;
                } else
                        tdproc(tp, T_OUTPUT);
        }
}

tdrint(dev)
{
        register int c, status;
        register int addr;
        register struct tty *tp;

        tp = &td_tty[dev];
        addr = td_addr[dev];

        /* get char and status */
```

```
250       c = inb( addr + RRDATA );
```

209-215: While the interrupt identification register indicates that there are more interrupts, call the appropriate routine. When the condition that caused the interrupt is resolved, the UART will reset the bit in the register by itself.

## tdxint() - lines 219 to 238

The *tdxint*() routine is called when a transmitter ready interrupt is received. It may issue a CSTOP character to indicate that the device on the other end must stop sending characters, — it may issue a CSTART character to indicate that the device on the other end may resume sending characters, or it may call *tdproc*() to send the next character in the queue.

226: If the transmitter is ready, reset the busy indicator.

229-231: If the line is to be restarted, send a CSTART, and reset the indicator.

232-234: If the line is to be stopped, send a CSTOP, and reset the character.

235-236: Otherwise, call *tdproc*() and ask it to send the next character in the queue.

## tdrint() - lines 240 to 263

The *tdrint*() routine is called when a receiver interrupt is received. All it has to do is pass the character, along with any errors, to the appropriate routine via the *linesw* table.

250-251: Get the character and status.

```
    status = inb(addr + RSTATUS);

    /*
     * Were there any errors on input?
     */
    if( status & SOERR )    /* overrun error */
            c |= OVERRUN;
    if( status & SPERR )    /* parity error */
            c |= PERROR;
    if( status & SFERR )    /* framing error */
            c |= FRERROR;
    (*linesw[tp->t_line].l_input)(tp, c, 0);
}

tdmint(dev)
{
    register struct tty *tp;
    register int addr,c;

    tp = &td_tty[dev];
    if ( tp->t_cflag & CLOCAL ) {
            return;
    }
    addr = td_addr[dev];

    if (inb(addr + RSTATUS) & SDSR) {
            if ((tp->t_state & CARR_ON)==0) {
                    tp->t_state |= CARR_ON;
                    wakeup(&tp->t_canq);
            }
    } else {
            if (tp->t_state & CARR_ON) {
                    if (tp->t_state & ISOPEN) {
                            signal(tp->t_pgrp, SIGHUP);
                            tdmodem(dev, TURNOFF);
                            ttyflush(tp, (FREAD|FWRITE));
                    }
                    tp->t_state &= ~CARR_ON;
            }
    }
}

tdioctl(dev, cmd, arg, mode)
int dev;
int cmd;
faddr_t arg;
int mode;
{
    if (ttiocom(&td_tty[dev], cmd, arg, mode))
```

```
300                 tdparam(dev);
301         }
```

256-261: If any errors were detected, set the appropriate bit in *c*.

262: And finally, pass the character and errors to the *l_input*() routine for the current line discipline.

## tdmint() - lines 265 to 291

The *tdmint*() routine is called whenever a modem interrupt is caught.

271-272: If there is no modem support for this line, just return.

276-279: If a data-set-ready is present for this line, and it wasn't before, mark the line as having carrier, and wake up any processes that are waiting for the carrier before their *tdopen*() call can be completed.

281-290: If no data-set-ready is present for this line, and one existed before, send a hangup signal to all of the processes associated with this line, call *tdmodem*() to hang up the line, flush the output queue for this line by calling *ttyflush*(), and finally, mark the line as having no carrier.

## tdioctl() - lines 293 to 301

The *tdioctl*() routine is called when some process makes an **ioctl** system call on a device associated with the driver. It just calls *ttiocom*() which returns a non-zero value if the hardware must be reconfigured.

```

tdproc(tp, cmd)
register struct tty *tp;
{
    register c;
    register int addr;

    extern ttrstrt();

    addr = td_addr[tp - td_tty];
    switch (cmd) {

    case T_TIME:
            tp->t_state &= ~TIMEOUT;
            outb(addr + RCNTRL, inb(addr + RCNTRL) & ~CBREAK);
            goto start;

    case T_WFLUSH:
    case T_RESUME:
            tp->t_state &= ~TTSTOP;
            goto start;

    case T_OUTPUT:
    start:
            if (tp->t_state&(TIMEOUT|TTSTOP|BUSY))
                    break;
            if ( (tp->t_state&TTIOW) && tp->t_outq.c_cc==0) {
                    tp->t_state &= ~TTIOW;
                    wakeup((caddr_t)&tp->t_oflag);
            }
            while ((c=getc(&tp->t_outq)) >= 0) {
                    if (tp->t_oflag&OPOST && c == 0200) {
                            if ((c = getc(&tp->t_outq)) < 0)
                                    break;
                            if (c > 0200) {
                                    tp->t_state |= TIMEOUT;
                                    timeout(ttrstrt, (caddr_t)tp,
                                    (c&0177)); break;
                            }
                    }
                    tp->t_state |= BUSY;
                    outb(addr + RTDATA, c);
                    break;
            }
            if ( (tp->t_state&OASLP) && tp->t_outq.c_cc<=
                          ttlowat[tp->t_cflag&CBAUD]) {
                       tp->t_state &= ~OASLP;
                    wakeup((caddr_t)&tp->t_outq);
            }
```

```
350             break;
```

## tdproc() - lines 303 to 382

The *tdproc*() routine is called to effect some change on the output, such as emitting the next character in the queue, or halting or restarting the output.

312: The *cmd* argument determines the action taken.

314-317: The time delay for outputting a break has finished. Reset the flag that indicates there is a delay in progress, and stop sending a continuous space. Then restart output by jumping to *start*.

321-322: Either a line on which output was stopped is restarting, or someone is waiting for the output queue to drain. Reset the flag indicating that output on this line is stopped, and start the output again by jumping to *start*() (line 325).

326-327: Try to put out another character. If some delay is in progress (TIMEOUT) or the line output has stopped (TTSTOP) or a character is in the process of being output (BUSY), just return.

328-330: If some process was waiting for the output queue to drain, reset the indicator, and wake the process.

332: While characters still exist in the output buffer do the following:

333-340: If output postprocessing is occurring on this line, and the current character is a delay marker (octal 200), get the next character, which specifies the delay in clock ticks, mark the line as waiting for a delay to expire, and schedule the line to be restarted via the *timeout*() routine.

342-344: Otherwise, output a character; mark the line BUSY, and pass the character to the controller.

346-348: If some process is waiting because the buffer went over the high water mark, and it is now below the low water mark, wake it up.

```

	case T_SUSPEND:
		tp->t_state |= TTSTOP;
		break;

	case T_BLOCK:
		tp->t_state &= ~TTXON;
		tp->t_state |= TBLOCK;
		if (tp->t_state&BUSY)
			tp->t_state |= TTXOFF;
		else
			outb(addr + RTDATA, CSTOP);
		break;

	case T_RFLUSH:
		if (!(tp->t_state&TBLOCK))
			break;
	case T_UNBLOCK:
		tp->t_state &= ~(TTXOFF|TBLOCK);
		if (tp->t_state&BUSY)
			tp->t_state |= TTXON;
		else
			outb(addr + RTDATA, CSTART);
		break;

	case T_BREAK:
		outb( addr + RCNTRL, inb( addr + RCNTRL ) | CBREAK );
		tp->t_state |= TIMEOUT;
		timeout(ttrstrt, tp, HZ/4);
		break;
	}
}
```

352-354: To stop the output on this line, since there is no way to stop the character we have already passed to the controller, just flag the line stopped, and drop through.

356-363: To tell the device on the other end to stop sending characters, reset the flag asking to stop the line, and mark the line stopped. If the line is already busy, set the flag; otherwise, output a CSTOP character.

365-367: A process is waiting to flush the input queue. If the device hasn't been blocked, just return. Otherwise, drop through and unblock the device.

368-374: To tell the device on the other end to resume sending characters, adjust the flags. If the controller is sending a character, set the flag to send a CSTART later; otherwise, send the CSTART now.

376-380: To send a break, set the transmitter to continuous space, mark the line as waiting for a delay, and schedule output to be restarted later.

## 12.4 Sample Device Driver for Disk Drive

```
  /*
  ** hd- prototype hard disk driver
  */

#include "../h/param.h"
#include "../h/buf.h"
#include "../h/iobuf.h"
#include "../h/dir.h"
#include "../h/conf.h"
#include "../h/user.h"

/* disk parameters */
#define NHD     4          /* number of drives */
#define NCPD    600        /* # cylinders/disk */
#define NTPC    4          /* # tracks/cylinder */
#define NSPT    10         /* # sectors/track */
#define NBPS    512        /* # bytes/sector */
 #define NSPB    (BSIZE/NBPS)                /* sectors/block */
#define NBPC    (NTPC*NSPT*NSPB)    /* blocks/cylinder */

/* addresses of controller registers */
#define RBASE   0x00        /* base of all registers */
#define RCMD    (RBASE+0)   /* command register */
#define RSTAT   (RBASE+1)   /* status - nonzero means error */
#define RCYL    (RBASE+2)   /* target cylinder */
#define RTRK    (RBASE+3)   /* target track */
#define RSEC    (RBASE+4)   /* target sector */
#define RADDRL  (RBASE+5)   /* target memory address lo 16 bits*/
#define RADDRH  (RBASE+6)   /* target memory address hi 8 bits*/
#define RCNT    (RBASE+7)   /* number of sectors to xfer */

/* bits in RCMD register */
#define CREAD   0x01        /* start a read */
#define CWRITE  0x02        /* start a write */
#define CRESET  0x03        /* reset the controller */

/*
** minor number layout is 000ddppp
**  where d is the drive number and ppp is the partition
*/
#define drive(d)    ( minor(d) >> 3)
#define part(d)     ( minor(d) & 0x07)

/* partition table */
struct partab {
    daddr_t len;            /* # of blocks in partition */
46        int      cyloff;         /* starting cylinder of partition */
47  };
48
```

## Description of Device Driver for Disk Drive

The device driver presented here is for an intelligent controller that is attached to one or more disk drives. The controller can handle multiple sector transfers that cross track and cylinder boundaries.

13: NHD defines the number of drives the controller can be attached to.

14-18: Each disk drive attached to the controller has NCPD cylinders; each cylinder has NTPC tracks, and each track has NSPT sectors. The sectors are NBPS bytes long and each cylinder has NBPC blocks.

21-29: The controller registers occupy a region of contiguous address space starting at RBASE and running through RBASE+7.

32-34: To make the controller perform some action, the registers that describe the transfer (RCYL, RTRK, RSEC, RADDRL, RADDRH, RCNT) are set to the appropriate values, and then the bit representing the desired action is written into the RCMD register.

40-41: The *drive*() and *part*() macros split out the two parts of the minor number. Bits 0 through 2 represent the partition on the disk, and the remaining bits specify the drive number. Thus, the minor number for drive 1, partition 2 would be 10 decimal.

44-46: Large disks are typically broken into several partitions of a more manageable size. The structure that specifies the size of the partitions specifies the length of the partition in blocks, and the starting cylinder of the partition.

```
struct partab hd_sizes[8] = {
        NCPD*NBPC,      0,              /* whole disk */
        ROOTSZ*NBPC,    0,              /* root area */
        SWAPSZ*NBPC,    ROOTSZ,         /* swap area */
        USERSZ*NBPC,    USROFS,         /* usr area */
        0,      0,                      /* spare */
        0,      0,                      /* spare */
        0,      0,                      /* spare */
        0,      0,                      /* spare */
    };

struct  iobuf   hdtab;                  /* start of request queue */
struct  buf     rhdbuf;                 /* header for raw i/o */
/*
**    Strategy Routine:
**    Arguments:
**      Pointer to buffer structure
**    Function:
**      Check validity of request
**      Queue the request
**      Start up the device if idle
*/
int hdstrategy(bp)
register struct buf *bp;
{
        register int dr, pa;    /* drive and partition numbers */
        daddr_t ss, bn;
        int x;
        dr = drive(bp->b_dev);
        pa = part(bp->b_dev);
         bn = bp->b_blkno * NSPB;
        ss = (bp->b_bcount  + BMASK) >> BSHIFT;
        if ( dr<NHD && pa<NPARTS && bn>=0 && bn<hd_sizes[pa].len &&
            ((bn + ss < hd_sizes[pa].len) || (bp->b_flags & B_READ)))
        {
            if ( bn + ss > hd_sizes[pa].len ) {
                ss = (hd_sizes[pa].len - bn) * NBPS;
                bp->b_resid = bp->b_bcount - (unsigned) ss;
                bp->b_bcount = (unsigned) ss;
            }
        } else {
            bp->b_flags |= B_ERROR;
            iodone(bp);
            return;
        }
        bp->b_cylin = (b_blkno / NBPC) + hd_sizes[pa].cyloff;
        x = spl5();
        disksort(&hdtab, bp);
        if (hdtab.b_active == = NULL)
99              hdstart();
100         splx(x);
101     }
```

49-53: This driver splits a disk into up to eight pieces, but at present, only four are used. The first partition covers the whole disk. The remaining three split the disk three ways, one partition for each of root, swap, and usr.

60: The buffer headers representing requests for this driver are linked into a queue, with *hdtab* forming the head of the queue. In addition, information regarding the state of the driver is kept in *hdtab*.

61: Each block driver that wants to allow raw I/O allocates one buffer header for this purpose.

## hdstrategy() - lines 72 to 101

The *hdstrategy*() routine is called by the kernel to queue a request for I/O. The single argument is a pointer to the buffer header which contains all of the data relevant to the request. The strategy routine is responsible for validating the request, and linking it into the queue of outstanding requests.

79-81: First, compute various useful numbers that will be used repeatedly during the validation process.

82-94: If the request is for a non-existent drive or a non-existent partition, if it lies completely outside the specified partition, or is a write, and ends outside the partition, the B_ERROR bit in the *b_flags* field of the header is set to indicate that the request has failed. The request is then marked as complete by calling *iodone*() with the pointer to the header as an argument. If the request is a read, and ends outside the partition, it is truncated to lie completely within the partition.

95: Compute the target cylinder of the request for the benefit of the *disksort*() routine.

96: Block interrupts, to prevent the interrupt routine from changing the queue of outstanding requests.

97: Sort the request into the queue by passing it and the head of the queue to *disksort*().

98: If the controller is not already active, start it up.

99: Re-enable interrupts and return to the user process.

```
102
103  /*
104   *      Startup Routine:
105   *      Arguments:
106   *        None
107   *      Function:
108   *        Compute device-dependent parameters
109   *        Start up device
110   *        Indicate request to I/O monitor routines
111   */
112  hdstart()
113  {
114      register struct buf *bp;      /* BUFFER POINTER */
115      register unsigned sec;
116
117      if ((bp = hdtab.b_actf) == NULL) {
118          hdtab.b_active = NULL;
119          return;
120      }
121      hdtab.b_active = 1;
122
123      sec = (unsigned)bp->blkno * NSPB);
124      out(RCYL, sec / NSPC);        /* cylinder */
125      sec %= NSPC;
126      out(RTRK, sec / NSPT);        /* track */
127      out(RSEC, sec % NSPT);        /* sector */
128      out(RCNT, bp->b_count / NBPS);  /* count */
129      out(RDRV, drive(bp->b_dev));    /* drive */
130      out(RADDRL, bp->b_paddr & 0xffff);  /* memory address lo */
131      out(RADDRH, bp->b_paddr >> 16);     /* memory address hi */
132      if ( bp->b_flags & B_READ )
133          out(RCMD, CREAD);
134      else
135          out(RCMD, CWRITE);
136  }
137
138  /*
139   *      Interrupt routine:
140   *        Check completion status
141   *        Indicate completion to i/o monitor routines
142   *        Log errors
143   *        Restart (on error) or start next request
144   */
145  hdintr()
146  {
```

```
147         register struct buf *bp;
148
149         if (hdtab.b_active == 0)
150             return;
```

### hdstart() - lines to 112 to 136

The *hdstart*() routine performs the calculation of the physical address on the disk, and starts the transfer.

117-119: If there are no active requests, mark the state of the driver as idle, and return.

121: Mark the state of the driver as active.

123-127: Calculate the starting cylinder, track, and sector of the request, and load the controller registers with these values.

129-131: Load the controller with the drive number, and the memory address of the data to be transferred.

132-135: If the request is a read request, issue a read command; otherwise, issue a write command.

### hdintr() - lines 145 to 171

The *hdintr*() routine is called by the kernel through the *vecintsw* table whenever the controller issues an interrupt.

149-150: If an unexpected call occurs, just return.

```

    bp = hdtab.b_actf;

    if ( in(RSTAT) != 0 )
        out(RCMD, CRESET);
        if (++hdtab.b_errcnt <= ERRLIM) {
            hdstart();
            return;
        }
        bp->b_flags |= B_ERROR;
        deverr(&hdtab, bp, in(RSTAT), 0);
    }
    /*
     *      Flag current request complete, start next one
     */
    hdtab.b_errcnt = 0;
    hdtab.b_actf = bp->av_forw;
    bp->b_resid = 0;
    iodone(bp);
    hdstart();
}

/*
 *  raw read routine:
 *    This routine calls "physio" which computes and validates
 *    a physical address from the current logical address.
 *
 *      Arguments
 *        Full device number
 *      Functions:
 *        Call physio which does the actual raw (physical) I/O
 *        The arguments to physio are:
 *          pointer to the strategy routine
 *          buffer for raw I/O
 *          device
 *          read/write flag
 */
hdread(dev)
{

    physio(hdstrategy, &rhdbuf, dev, B_READ);
}

/*
 *      Raw write routine:
 *      Arguments(to hdwrite):
 *        Full device number
 *      Functions:
 *        Call physio which does actual raw (physical) I/O
```

```
200     */
```

152: Get a pointer to the first buffer header in the chain; this is the request that is currently being serviced.

154-162: If the controller indicates an error, and the operation hasn't been retried ERRLIM times, try it again. If it has been retried ERRLIM times, assume it is a hard error, mark the request as failed, and call *deverror*() to print a console message about the failure.

166-171: Mark this request complete, take it out of the request queue, and call *hdstart*() to start on the next request.

### hdread() - lines 188 to 192

The *hdread*() routine is called by the kernel when a process requests raw read on the device. All it has to do is call *physio*(), passing the name of the strategy routine, a pointer to the raw buffer header, the device number, and a flag indicating a read request. The *physio*() routine does all the preliminary work, and queues the request by calling the device strategy routine.

### Last Five Lines of Sample Driver

```
201     hdwrite(dev)
202     {
203
204             physio(hdstrategy, &rhdbuf, dev, B_WRITE);
205     }
```

### hdwrite() - lines 201 to 205

The *hdwrite*() routine is called by the kernel when a process requests a raw write on the device. Its responsibilities and actions are the same as *hdread*(), except that it passes a flag indicating a write request.

# Appendix A
# C Language Portability

## A.1 Introduction

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) and 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 pathnames 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 mainframes. 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 of 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 is 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.

## A.2 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 pathnames unless a pathname is common to all systems (e.g.,

Files required at compiletime (i.e., include files) may also introduce nonportability if the pathnames 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.

## A.3 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.

### A.3.1 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.

### A.3.2 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 Beyond that no assumptions are portable. For example on some machines **short** is the same length as whereas on others **long** is the same length as

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 PDP11
#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.

### A.3.3 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 CPU's, 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 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.

### A.3.4 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, on some systems there is 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.

## A.3.5 Bitfields

Bitfields are not implemented in all C compilers. When they are, no field may be larger than an 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 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.

## A.3.6 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 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 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 attract a warning message from *lint*.

The C Reference manual 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 a **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.

## A.3.7 Address Space

The address space available to a program running under XENIX varies considerably from system to system. On a small PDP-11 there may be only 64K bytes 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 may 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.

## A.3.8 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 between zero and 127. Thus they can all be represented in 7 bits, and on an 8-bits-per-byte machine are that all positive, whether **char** is treated as signed or unsigned.

There is a set of macros defined under XENIX 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.

## A.4 Compiler Differences

There are a number of C compilers running under XENIX. On PDP-11 systems there is the so-called "Ritchie" compiler. Also on the 11, and on most other systems, there is the Portable C Compiler.

### A.4.1 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.

### A.4.2 Shift Operations

The left shift operator, "<<" shifts its operand a number of bits left, filling vacated bits with zero. This is a so-called 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 it is necessary to 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 using the divide operator:

```
        char c;

        c = c / 8;
```

### A.4.3 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 linker 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.

### A.4.4 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. On a PDP-11, up to three register declarations are significant, and they must be of type or pointer. While other machines and compilers may support declarations such as

```
          register unsigned short
```

this should not be relied upon.

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.

### A.4.5 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 which 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 Machines that sign extend **char** can give surprises. 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 passing to the function. The function itself has no knowledge of the original type of the argument, and is expecting an 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);
```

## A.4.6 Functions With a Variable Number of Arguments

Functions with a variable number of arguments present a particular portability problem if the type of the arguments is variable too. In such cases the code is dependent upon the size of various data types.

In XENIX there is 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 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 there is just one argument 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 and 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., and **short** on long word machines) must be declared as

### A.4.7 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. There are certain XENIX macros 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 here is a separator for two C statements. 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;
```

## A.5 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' environment.

Any program containing hard-coded pathnames to files or directories, or user IDs, login names, terminal lines or other system dependent parameters is nonportable. These types of constant should be in header files, passed as command line arguments, obtained from the environment, or obtained by using the XENIX default parameter library routines *dfopen*, and *dfread*.

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 family, and 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 upon 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. The reference manual page for contains the correct specification for these routines.

## A.6 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 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.

## A.7 Lint

*Lint* is a C program checker which 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 linker do this.

*Lint* will generate warning messages about nonportable pointer arithmetic, assignments, and type conversions. Passage unscathed through *lint* is not a guarantee that a program is completely portable.

## A.8 Byte Ordering Summary

The following conventions are used in the tables below:

| | |
|---|---|
| a0 | The lowest physically addressed byte of the data item. 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!

Byte Ordering for Short Types

| CPU | Byte Order | |
|---|---|---|
| | a0 | a1 |
| PDP-11 | b0 | b1 |
| VAX-11 | b0 | b1 |
| 8086 | b0 | b1 |
| 286 | b0 | b1 |
| M68000 | b1 | b0 |
| Z8000 | b1 | b0 |

Byte Ordering for Long Types

| CPU | Byte Order | | | |
|---|---|---|---|---|
| | a0 | a1 | a2 | a3 |
| PDP-11 | b2 | b3 | b0 | b1 |
| VAX-11 | b0 | b1 | b2 | b3 |
| 8086 * | b0 | b1 | b2 | b3 |
| 8086 ** | b2 | b3 | b0 | b1 |
| 286 | b0 | b1 | b2 | b3 |
| M68000 | b3 | b2 | b1 | b0 |
| Z8000 | b3 | b2 | b1 | b0 |

Note that byte ordering for long types is compiler dependent (not CPU dependent) on PDP-11 and 8086 based machines. This table is based on a PDP-11 using the Ritchie compiler. 8086 * shows byte ordering for compilers using little-endian word order. 8086 ** shows byte ordering for big-endian compilers. 8086 users can refer to the XENIX *Development System Release Notes* for the type of word order used of the compiler.

# Appendix B
# M4: A Macro Processor

## B.1 Introduction

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
- 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 twenty built−in macros. In addition, the user can define new macros. Built−ins 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.

## B.2 Invoking m4

The invocation syntax for *m4* is:

```
m4 [files]
```

Each file name argument is processed in order. If there are no arguments, or if an argument is a dash (−), then the standard 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*.

## B.3 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.

Things may be defined in terms of other things. 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 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 **define** are being collected, it is immediately replaced by 100; it's just as if you had said

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

in the first place.

If this isn't what you really want, there are two 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).

## B.4 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 of the same thing, which is a bit more surprising, consider redefining "N":

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

Perhaps regrettably, the "N" in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

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

This statement is ignored by *m4*, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really 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-ins related to **define**. The built-in **undefine** removes the definition of some macro or built-in:

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

removes the definition of "N". Built-ins can be removed with **undefine**, as in

```
undefine('define')
```

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to determine if a macro is currently defined. 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 say:

```
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)
```

## B.5 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 as many arguments as you want, but only the first nine are accessible, through $1 to $9. (The macro name itself is $0.) Arguments that are not supplied are replaced by null strings, so we can define a macro *cat* which simply concatenates its arguments, like this:

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

Thus

```
cat(x, y, z)
```

is equivalent to

```
xyz
```

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)". And of course a bare comma or parenthesis can be inserted by quoting it.

## B.6 Using Arithmetic Built-ins

*M4* provides two built-in functions 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)')
```

Then "N1" is defined as one more than the current value of "N".

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. 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 false is 0. The precision in **eval** is implementation dependent.

As a simple example, suppose we want "M" to be "2**N+1". Then

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

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

## B.7 Manipulating Files

You can include a new file in the input at any time by the built-in function **include**:

include(*filename*)

inserts the contents of *filename* in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** (for "silent include") says nothing and continues if it can't access the file.

It is also possible to divert the output of *m4* to temporary files during processing, and output the collected material upon command. *M4* maintains nine of these diversions, numbered 1 through 9. If you say

```
divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as "n". Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

```
undivert
```

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is not the diverted stuff. 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.

## B.8 Using System Commands

You can run any program in the local operating system with the **syscmd** built-in. For example,

```
syscmd(date)
```

runs the **date** command. Normally, **syscmd** would be used to create a file for a subsequent **include**.

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function **mktemp**: a string of "XXXXX" in the argument is replaced by the process id of the current process.

## B.9 Using Conditionals

There is a built-in called **ifelse** which enables you to perform 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** which 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 too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

**ifelse** can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input

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

if the string *a* matches the string *b*, the result is *c*. Otherwise, if *d* is the same as *e*, the result is *f*. Otherwise the result is *g*. If the final argument is omitted, the result is null, so

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

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

## B.10 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
```

If *i* or *n* are out of range, various sensible things happen.

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 found in *f* by the corresponding character of *t*. That is

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

replaces the vowels by 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".

There is also a built-in called **dnl** which deletes all characters that follow it up to and including the next newline. It is useful mainly for throwing away empty lines that otherwise tend to clutter up *m4* output. For example, if you say

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

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines will disappear.

Another way to achieve this, is

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

## B.11 Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus, you can say

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

**Dumpdef** is a debugging aid that dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget the quotation marks.

# Appendix C A Common Library For XENIX and MS-DOS

## C.1 Introduction

This appendix lists the XENIX library routines that form the Common C Library for the XENIX and MS–DOS versions of the Microsoft C compiler. These routines can be used by programmers who wish to develop C programs for both the XENIX and MS–DOS environments. The routines provide an identical interface to a set of operations that are useful on both XENIX and MS–DOS.

The following is a list of the common routines:

| | | | | | |
|---|---|---|---|---|---|
| abort* | ecvt | free | islower | putw | strncmp |
| abs | execl* | freopen* | isprint | rand | strncpy |
| access* | execle* | frexp | ispunct | read* | strpbrk |
| acos | execlp* | fscanf | isspace | realloc | strrchr |
| asctime* | execv* | fseek* | isupper | rewind | strspn |
| asin | execve* | fstat* | isxdigit | sbrk* | strtok |
| assert | execvp* | ftell | j0 | scanf | swab |
| atan | exit* | ftime | j1 | setbuf | system* |
| atan2 | exp | fwrite | jn | setjmp | tan |
| atof | fabs | gcvt | ldexp | signal* | tanh |
| atoi | fclose | getc | localtime* | sin | time |
| atol | fcvt | getchar | log | sinh | toascii |
| cabs | fdopen* | getcwd | log10 | sprintf | tolower |
| calloc | feof | getenv | longjmp | sqrt | toupper |
| ceil | ferror | getpid* | lseek* | srand | _tolower |
| chdir* | fflush | gets | malloc | sscanf | _toupper |
| chmod* | fgetc | getw | mktemp* | stat* | tzset |
| chsize* | fgets | gmtime* | modf | strcat | umask* |
| clearerr | fileno | hypot | open* | strchr | ungetc |
| close | floor | isalnum | perror | strcmp | unlink* |
| cos | fmod | isalpha | pow | strcpy | utime |
| cosh | fopen* | isascii | printf | strcspn | write* |
| creat* | fprintf | isatty* | putc | strdup | y0 |
| ctime* | fputc | iscntrl | putchar | strlen | y1 |
| dup | fputs | isdigit | puts | strncat | yn |
| dup2 | fread | isgraph | | | |

The following is a list of the variables used by the common routines and available in both environments:

daylight timezone tzname

Routines marked by an asterisk (*) have a slightly different operation or meaning in the MS–DOS environment than they do under XENIX. These differences are fully described in the following sections. Routines which are not marked function exactly the same in MS–DOS as they do in XENIX. Complete descriptions are given in section S of the XENIX *Reference Manual*.

In addition to common routines, this appendix describes the following MS–DOS specific routines:

| | | | |
|---|---|---|---|
| eof | itoa | spawnle | strnset |
| fcloseall | labs | spawnlp | strrev |
| fgetchar | ltoa | spawnv | strset |
| filelength | mkdir | spawnve | strupr |
| flushall | rmdir | spawnvp | tell |
| fputchar | spawnl | strlwr | ultoa |

## C.2 Common Include Files

Structure definitions, return value types, and manifest constants used in the descriptions of some of the common routines may vary from environment to environment and are therefore fully defined in a set of include files for each environment. There are the following include files:

assert.h
ctype.h
errno.h
fcntl.h
math.h
setjmp.h
signal.h
stdio.h
time.h
sys/timeb.h
sys/types.h
sys/stat.h
sys/utime.h

The *assert.h* file defines the statements used to implement the *assert* function.

The *ctype.h* file defines the values and macros used to support the character translation and testing functions.

The *errno.h* file contains definitions of the error values returned in the global variable, errno. Whenever a library routine or system call detects an error, a general error indicator is returned from the call. The indicator is defined to be some otherwise illegal return value, usually −1. This method is used to avoid possible conflicts between an error return and a legitimate return value. When an error return is detected, the actual error value can be determined by looking at the value of errno. The value of errno is undefined if the function returned a non−error result.

The *fcntl.h* file defines the values and macros used with the *open* and *creat* functions.

The *math.h* file defines some of the floating point math routine interfaces and some standard constants.

The *setjmp.h* function defines the structure used with the *setjmp* and *longjmp* functions.

The *signal.h* file defines the values and macros used with the *signal* function.

The *stdio.h* file contains the definitions of the basic system file structure, FILE, some of the basic operations available for files, such as the *putc*, *getc*, *putchar*, and *getchar* macros, as well as the standard pointer constant NULL.

The *time.h* file defines the structure **tm** returned with the *localtime* and *gmtime* functions, and used by the *asctime* function.

The *sys/timeb.h* file defines the structure **timeb** used with the *ftime* function.

The *sys/types.h* file defines some of the types used in defining system structures, such as the time, date, and file status structures.

The *sys/stat.h* file defines the format, fields, and constant values for the file status structure returned by the *stat* and *fstat* routines. It also defines the file permission modes that may be used in the *open*, *chmod*, and *creat* functions.

The *sys/utime.h* file defines the structure used with the time routines.

## C.3 Differences Between Common Routines

The following sections explain how the MS-DOS routines of the common library differ from their XENIX counterparts. These descriptions are intended to be used in conjunction with the more detailed descriptions of XENIX functions provided in section S of the XENIX *Reference Manual*.

### C.3.1 Abort

The *abort* routine terminates the process and returns control to the operating system *without* creating a core file. It also copies the message "Abnormal program termination" to the **stderr** file.

### C.3.2 Access

The *access* routine checks the access to a given file. Access does not depend on real and effective IDs as it does in the XENIX environment. Under MS-DOS, the real and effective IDs are ignored. *Amode* can be any combination of the values:

04 Read
02 Write
00 Check for existence

The "Execute" access mode (01) is not allowed.

The EROFS and ETXTBSY error values are not used.

### C.3.3 Chdir

The *chdir* routine causes the named directory to become the current working directory just as it does in the XENIX environment. The only difference is that the directory pathname under MS-DOS should have a backslash separator (\) instead of a slash separator (/).

## C.3.4 Chmod

The *chmod* routine can set the "owner" access permissions for a given file, but all other permissions settings are ignored. *Mode* can be:

| | |
|---|---|
| S_IREAD | Read by owner |
| S_IWRITE | Write by owner |
| S_IREAD\|S_IWRITE | Read and write by owner |

The S_IREAD and S_IWRITE constants are defined in the *sys/stat.h* include file. Note that the OR operator (|) is used to combine these constants to form read and write permission.

If write permission is not given, the file is treated as a read-only file. If read permission is not given, the file is assumed to be readable. MS-DOS does not support non-readable files.

The *chmod* routine under MS-DOS is not affected by real or effective IDs.

The EPERM and ETXTBSY error values are not used.

## C.3.5 Chsize

The *chsize* routine changes the size of the given file just as it does in the XENIX environment. However, the maximum size of a file is not affected by the limit defined by the *ulimit*(S) routine. There is no *ulimit* routine for the MS-DOS environment.

## C.3.6 Creat

The *creat* routine creates a new file or prepares an existing file for writing. If the file is created, the access permissions are set as defined by *mode*. Only "owner" permissions are allowed (see "Chmod" above). Ownership of the file is not affected by the real and effective user and group IDs. (These are ignored under MS-DOS).

The EROFS and ETXTBSY error values are not used by *creat* under MS-DOS.

As in the XENIX environment, use of the *open* routine is preferred over *creat* when creating or opening files in the MS-DOS environment.

## C.3.7 Ctime, Localtime, Gmtime, and Asctime

Although the *ctime*, *localtime*, *gmtime*, and *asctime* routines, which return a date and time, carry out the same time conversions as in the XENIX environment, the earliest possible date returned by these routines in the MS-DOS environment is January 1, 1980. The XENIX routines can return dates as early as January 1, 1970. If the routines are passed values representing dates earlier than January 1, 1980, they return this date.

## C.3.8 Exec

The *execl*, *execle*, *execlp*, *execv*, *execve*, and *execvp* routines overlay the calling process as in the XENIX environment. If there is not enough memory for the new process, the exec routine will fail and return to the calling process. Otherwise, the new process begins execution.

Under MS-DOS, the exec routines *do not*:

— Use the close-on-exec flag to determine open files for the new process.

— Use the set user and group ID access permissions of the new process file to determine effective user and group IDs.

— Set up signal processing for the new process.

— Disable profiling for the new process (profiling is not allowed under MS-DOS).

— Give the new process attributes inherited from the calling process.

— Use the ETXTBSY error value.

The combined size of all arguments in an exec routine under MS-DOS must not exceed 128 bytes.

## C.3.9 Exit

The *exit* function terminates the current process and makes the low order byte of *status* available to the parent process. When the process is terminated, all buffers are flushed and released. Also, all open files in the calling process are closed.

## C.3.10 Fopen, Fdopen, Freopen

The *fopen*, *fdopen*, and *freopen* routines open stream files just as they do in the XENIX environment. However, there are the following additional values for the *type* string:

| | |
|---|---|
| t | Opens the file in **text** mode. Opening a file in this mode causes the low-level I/O routines to translate carriage return/linefeed (CR-LF) character combinations into a single linefeed (LF) on input. Similarly on output, linefeeds are translated into CR-LF combinations. |
| b | Opens the file in **binary** mode. This mode suppresses translation. |

For example, the call

```
    fopen("test.dat", "rt");
```

opens a file for reading in **text** mode.

If "t" or "b" is not given in the *type* string, then the mode is defined by the default mode variable *_fmode*. If *_fmode* is 0, the default mode is **text**. If the higher order bit of *_fmode* is 1, the default mode is **binary**. The linker initially sets the *_fmode* variable to 0 unless you link your program with the object file */lib/dos/rawmode.o*. This file sets *_fmode* to 1.

### C.3.11 Fseek

The *fseek* routine moves the file pointer to the given position just as in the XENIX environment. However, since MS−DOS uses the carriage return/linefeed (CR−LF) character combinations for newline characters ( XENIX uses only an LF), an *fseek* call which moves the file pointer a specific number of bytes past newline characters will not move the pointer to same place in the MS−DOS file as it does in the XENIX file. For example, if a file contains the characters

```
abcdef\n09123
```

(where \n is the newline character) and the file pointer is currently at the letter "a", then the call

```
fseek(stream, 8, 1);
```

moves the pointer to the digit "0", if the file is an MS−DOS file, or to the digit "9", if the file is a XENIX file.

Note that some *fseek* calls treat MS−DOS and XENIX files identically. For example,

```
fseek(stream, 0, 2)
```

always moves the file pointer to the end of the file.

### C.3.12 Getpid

The *getpid* routine returns a unique number. Although the number may be used to uniquely identify the process, it does not have the same meaning as the process ID returned by *getpid* in the XENIX environment.

### C.3.13 Isatty

The *isatty* routine indicates whether or not the given file descriptor is associated with any character device, not just a terminal. A character device can be a console, printer, or serial port.

### C.3.14 Lseek

The *lseek* routine is similar to the *fseek* routine under MS−DOS whenever the given file descriptor has been opened in text mode. In other words, *lseek* must move the file pointer one additional byte for each newline character in the MS−DOS file in order to move the file pointer to the same position in the XENIX file. See "Fseek" for more details.

### C.3.15 Mktemp

The *mktemp* routine creates a temporary filename, using a unique number instead of a process ID. The number is the same as returned by *getpid* (see "Getpid" above).

### C.3.16 Open

The *open* routine opens a file descriptor for a named file, just as in the XENIX environment. However, there is one additional *oflag* value, O_BINARY, and two values, O_NDELAY and O_SYNCW, have been removed.

The O_BINARY flag causes the file to be opened in the opposite mode specified by the *_fmode* variable (see "Fopen" above). For example, if the default mode is text (*_fmode* is 0), then using O_BINARY opens the file in binary mode, but if the default mode is binary (*_fmode* is 1), then using O_BINARY opens the file in text mode.

The EISDIR, EROFS, ETXTBSY, and ENXIO error values are not used.

### C.3.17 Read

The *read* routine reads characters from the file given by a file descriptor just as in the XENIX environment. However, if the file has been opened in text mode (see "Open" above), *read* will replace each CR-LF pair read from the file with a single LF character. The number of bytes returned is the number of bytes remaining after the the CR-LF pairs have been replaced. Thus, the return value may not always correspond with the actual number of bytes read. This is considered normal and has no implications as far as detecting the end of the file.

### C.3.18 Sbrk

The *sbrk* routine performs the same task as in the XENIX environment. However, *sbrk* is not affected by the limits imposed by *ulimit*(S), since no *ulimit* routine exists for MS-DOS.

### C.3.19 Signal

The *signal* routine can only handle the SIGINT signal. In MS-DOS, SIGINT is defined to be INT 23H (the CNTRL-C signal).

### C.3.20 Stat, Fstat

The *stat* and *fstat* routines return a structure defining the current status of the given file. The structure members returned by *stat* have the following names and meaning:

| | |
|---|---|
| st_mode | User read/write/execute bits are set or cleared to reflect the file's permissions. The execute bit is inferred from the filename extension. These are copied into the group and other bits. The S_IFREG bit is set if the file is a regular file; S_IFDIR is set if it is a directory. See *stat.h* in "Common Include Files" above. |
| st_ino | Not used. |

| | |
|---|---|
| st_dev | Drive number of the disk containing the file. |
| st_rdev | Drive number of the disk containing the file. |
| st_nlink | Always 1. |
| st_uid | Not used. |
| st_gid | Not used. |
| st_size | Size of the file in bytes. |
| st_atime | Time of last modification of file. |
| st_mtime | Time of last modification of file. |
| st_ctime | Time of last modification of file. |

*Fstat* returns less useful information since MS-DOS does not make as much information available for file descriptors as it does full pathnames. *Fstat* can detect device files, but it must not be used with directories. The structure returned by *fstat* has the following members:

| | |
|---|---|
| st_mode | User read and write bits are set or cleared to reflect the file's permissions. The S_IFCHR flag is set if this is a device. Otherwise, the S_IFREG bit is set. See *stat.h* in "Common Include Files" above. |
| st_ino | Not used. |
| st_dev | Either drive number of the disk containing the file, or file descriptor if this file is a device. |
| st_rdev | Either drive number of the disk containing the file, or file descriptor if this file is a device. |
| st_nlink | Always 1. |
| st_uid | Not used. |
| st_gid | Not used. |
| st_size | Size of the file in bytes. |
| st_atime | Time of last modification of file. |
| st_mtime | Time of last modification of file. |
| st_ctime | Time of last modification of file. |

### C.3.21 System

The *system* routine passes the given string the the operating system for execution. In order to execute this string, the full pathname of the directory containing the MS-DOS "COMMAND.COM" program must be assigned to the COMSPC environment variable, or assigned to the PATH environment variable. The call will return an error if "COMMAND.COM" cannot be found using these variables.

### C.3.22 Umask

The *umask* routine can set a mask for "owner" read and write access permissions only. All other permissions are ignored.

### C.3.23 Unlink

The *unlink* routine always deletes the given file. Since MS-DOS does not allow multiple "links" to the same file, unlinking a file is the same as deleting it.

The EBUSY, ETXTBSY, and EROFS error values are not used.

### C.3.24 Write

The *write* routine writes a specified number of characters to the file named by the given file descriptor just as in the XENIX environment. However, if the file has been opened in text mode (see "Open" above), every LF character in the output is replaced by a CR-LF pair before being written. This does not affect the return value.

## C.4 Differences in Definitions

Many of the special definitions given in *intro*(S) in the XENIX *Reference Manual* do not apply to the common routines when used in the MS-DOS environment. The following is a list of the differences.

The *process* ID is still a unique integer, but does not have the same meaning as in the XENIX environment.

The *parent process*, *process group*, *tty group*, *real user*, *real group*, *effective user* and *effective group* IDs are not used by the common routines when run under MS-DOS. Furthermore, there is no *super-user* or *special processes* in the MS-DOS environment.

*Filenames* in MS-DOS have two parts: a filename and a filename extension. Filenames may be any combination of up to eight letters or digits. Filename extensions may be any combination of up to three letters or digits, preceded by a period (.).

*Pathnames* in MS-DOS may be any combination of directory names separated by a backslash (\). The slash (/) used in the XENIX environment is not allowed unless the user has redefined the leading character used with options in MS-DOS command lines (this character is initially the slash). Directory names may be any combination of up to eight letters or digits. The special names "." and ".." refer to the current directory and the parent directory, respectively.

*Drive names* may be used at the begin of a pathname to specify a specific disk drive or device. Drives names are generally a letter or combination of letters and digits followed by a colon (:).

*Access permissions* in MS−DOS are restricted to read and write by the owner of the file. Since all users own all files in MS−DOS, access permissions do little more than define whether or not the file is a read−only file or can be modified. Execution permission and other permissions defined for files in the XENIX environment do not apply the files in the MS−DOS environment.

## C.5 MS−DOS Specific Routines

The MS−DOS specific routines are intended for programs being compiled in the XENIX environment, but which are to be executed in the MS−DOS environment only. These routines are not available for use in the XENIX environment. The following sections describe the routines in detail.

### C.5.1 Eof

```
int eof(fildes)
int fildes;
```

The *eof* function returns the value 1 if the current position of the file associated with *fildes* is at the end−of−file, otherwise the function returns 0. The return value −1 indicates an error.

### C.5.2 Fcloseall

```
int fcloseall( )
```

The *fcloseall* function closes all currently open streams, except **stdin**, **stdout**, and **stderr**. The function flushes all file buffers before closing, and although it releases system−allocated buffers, it does not release buffers allocated using *setbuf*.

*Fcloseall* returns the total number of streams closed. The return value −1 indicates an error.

### C.5.3 Fgetchar

```
#include <stdio.h>

int fgetchar( )
```

The *fgetchar* function reads a single character from the standard input stream **stdin**. *Fgetchar* is the function version of the macro *getchar*.

*Fgetchar* returns the character read, or EOF when end−of−file is reached.

### C.5.4 Filelength

```
long filelength(fildes)
int fildes;
```

The *filelength* function returns the length, in bytes, of the file associated with *fildes*. The return value −1 indicates an error.

### C.5.5 Flushall

```
int flushall( )
```

The *flushall* function flushes the buffers of all currently open output streams. All streams remain open after the call.

*Flushall* returns the total number of open streams (both input and output streams). There is no error return.

Note that buffers are automatically flushed when they are full, when the associated files are closed, or when a program terminates without closing the files.

### C.5.6 Fputchar

```
#include <stdio.h>

int fputchar(c)
char c;
```

The *fputchar* function writes the single character *c* to the output stream stdout. *Fputchar* is the function version of the macro *putchar*.

*Fputchar* returns the character written. The return value EOF indicates an error.

### C.5.7 Itoa, Ltoa, and Ultoa

```
char *itoa(value, string, radix)
int value;
char *string;
int radix;

char *ltoa(value, string, radix)
long value;
char *string;
int radix;

char *ultoa(value, string, radix)
unsigned long value;
char *string;
int radix;
```

The *itoa*, *ltoa*, and *ultoa* functions convert the given *value* to a character string that represents that value. The resulting string is stored in *string*, and consists of one or more digits from the numeric base given by *radix*.

*Itoa* converts type **int** values into strings, *ltoa* converts type **long** values, and *ultoa* converts type **unsigned long** values. The *radix* can be any in the range 2−36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (−).

All functions return a pointer to the new string. There is no error return, and no overflow checking is performed.

## C.5.8 Labs

```
long labs(value)
long value;
```

The *labs* function returns the absolute value of the type **long** number given by *value*. There is no error return.

## C.5.9 Mkdir

```
int mkdir(pathname)
char *pathname;
```

The *mkdir* function creates a new directory with the specified *pathname*. The last component of *pathname* names the new directory; the preceding components must identify an existing directory.

*Mkdir* returns the value 0 if the new directory was created. The return value −1 indicates an error.

## C.5.10 Rmdir

```
int rmdir(pathname)
char *pathname;
```

The *rmdir* function deletes the directory specified by *pathname*. The directory must be empty, and it must not be the current working directory or the root directory.

*Rmdir* returns the value 0 if the directory is successfully deleted. The return value −1 indicates an error.

## C.5.11 Spawn

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

int spawnl(modeflag, pathname, arg0, ..., argn, NULL)
int modeflag;
char *pathname, *arg0, ..., *argn;

int spawnle(modeflag, pathname, arg0, ..., argn, NULL, envp)
int modeflag;
char *pathname, *arg0, ..., *argn, *envp[ ];

int spawnlp(modeflag, filename, arg0, ..., argn, NULL)
int modeflag;
char *filename, *arg0, ..., *argn;

int spawnv(modeflag, pathname, argv)
int modeflag;
char *pathname, *argv[ ];

int spawnve(modeflag, pathname, argv, envp)
int modeflag;
char *pathname, *argv[ ], *envp[ ];

int spawnvp(modeflag, filename, argv)
int modeflag;
char *filename, *arv[ ];
```

The *spawn* functions load and execute new child processes. The *pathname* or *filename* argument names the executable file to be loaded. The *arg* n or *argv* arguments contain pointers to character strings to be passed to the new process. The *modeflag* argument defines the execution of the parent process after placing a call to a *spawn* function. The *envp* argument allows the user to alter the environment for the child process by passing a list of environment settings. The *spawnl*, *spawnle* and *spawnlp* functions are typically used in cases where the number of arguments is known in advance. *Spawnv*, *spawnve*, and *spawnvp* are useful when the number of arguments to the new process is variable. Pointers to the arguments are passed as an array, *argv*, which accommodates any number of elements.

The *modeflag* values are defined in the include file *spawn.h*. The following lists the meaning of each value:

| Modeflag | Meaning |
|---|---|
| P_WAIT | Suspend parent process until execution of child process is complete. |
| P_NOWAIT | Continue to execute parent process concurrently with child process. |
| P_OVERLAY | Overlay parent process with child process. |

When P_WAIT or P_NOWAIT is specified, there must be sufficient memory available for loading and executing the child process. If P_OVERLAY is specified, the parent process is destroyed and control cannot be returned to it. This is similar to the effect of the *exec* routines. Only P_WAIT and P_OVERLAY may be used under MS-DOS 2.0. P_NOWAIT is reserved for future implementations, and use of this flag with MS-DOS 2.0 will produce an error.

The *pathname* argument must be the full directory pathname for the file to be loaded. The *filename* argument (in the *spawnlp* and *spawnvp* functions) may be just the filename or a partial pathname for the file; the current value of the environment variable PATH is used to determine which directories are searched for this file.

The *arg* n arguments in the *spawnl*, *spawnle*, and *spawnlp* functions must be pointers to null-terminated character strings. These strings form the argument list for the child process. Their combined length must not exceed 128 bytes. (Terminating null characters (\0) are not counted.) Thus, any number of *arg*n arguments may be given, as long as the character count of the corresponding strings does not exceed 128. The NULL pointer value must mark the end of the *arg*n argument list.

The *argv* arguments in the *spawnv*, *spawnve*, and *spawnvp* functions must be pointers to a single array of pointers to the character strings. The combined length of the strings must not exceed 128 bytes. The NULL pointer value must be placed in the array element immediately following the element containing the last character string.

By convention, the *arg0* and *argv*[0] arguments should be a copy of the *pathname* or *filename* argument. A different value will not produce an error.

The *envp* argument in the *spawnle* and *spawnve* functions must be an array of character pointers, each element of which points to a null-terminated string defining an environment variable. An environment setting has the following form:

name=value

where *name* is the name of an environment variable and *value* is the string value to which that variable is set. Notice that *value* is not enclosed in double quotes. When *envp* is NULL, the child process inherits the environment settings of the parent process.

Files that are open when a call to a *spawn* function is made remain open in the new process. In the *spawnl*, *spawnlp*, *spawnv*, and *spawnvp* functions, the child process inherits the environment of the parent.

## Return Values

If the P_WAIT is specified, the return value is the exit status of the child process. The exit status is 0 if the process terminated normally. A positive exit status indicates an abnormal exit through an *abort* function call or an interrupt. The exit status may also be set to a non-zero value if the child process specifically calls the *exit* function with a non-zero argument.

If P_OVERLAY is specified and the child is successfully loaded, the routine never returns a value.

The return value –1 indicates an error (the child process is not started). The value –1 is also returned when P_NOWAIT is specified under MS–DOS 2.0.

## C.5.12 Strlwr and Strupr

```
char *strlwr(string)
char *string;

char *strupr(string)
char *string;
```

The *strlwr* function converts any uppercase letters in the given *string* to lowercase.

The *strupr* function converts any lowercase letters in the given *string* to uppercase.

*Strlwr* and *strupr* return a pointer to the converted *string*. There is no error return.

## C.5.13 Strset and Strnset

```
char *strset(string, c)
char *string, c;

char *strnset(string, c, n)
char *string, c;
unsigned int n;
```

The *strset* function sets all characters in the given *string* (except the terminating null character) to the character *c* and returns a pointer to the altered string.

The *strnset* function sets the first *n* characters of *string* to the character *c* and returns a pointer to the altered *string*. If *n* is greater than the length of a given string, the string length is used instead.

## C.5.14 Strrev

```
char *strrev(string)
char *string;
```

The *strrev* function reverses the order of the characters in the given *string*. The terminating null character (\0) remains in place.

*Strrev* returns a pointer to the altered *string*. There is no error return.

## C.5.15 Tell

```
long tell(fildes)
int fildes;
```

The *tell* function returns the current position of the file associated with *fildes*. The position is the number of bytes from the beginning of the file. The return value of −1 indicates an error.

# Appendix D
# Compiler, Assembler, and Linker Messages

## D.1 Introduction

This appendix lists the messages displayed by the **cc**, **as**, and **ld** commands when errors are encountered during compilation of a program. It also lists the restrictions imposed by the C compiler on the size and complexity of program source files and statements within source files.

## D.2 Compiler Error Messages

The error messages produced by the C compiler fall into three categories: warnings, program errors, and fatal errors. Warnings alert you to problems that may cause errors during execution of the program, but do not prevent compilation of your program. Program errors identify problems that make successful compilation of your program impossible. Fatal errors identify problems that prevent **cc** from continuing execution. Whenever the compiler encounters program or fatal errors, it terminates operation before producing an object file.

The following sections explain the meaning of the compiler error messages, and provide clues on how to solve the problem indicated by these messages.

### D.2.1 Warning Messages

The following is a complete list of compiler warnings messages. The number in square brackets ([ ]) at the end of each message gives the minimum warning level that must be set for the message to appear. You can set the warning level by using the **−W** option described earlier in this chapter.

warning: Address of frame variable taken, DS != SS [1]

Taking the address of a frame variable in a small model program with separate data and stack segments results in an incorrect address. The address does not refer to the correct segment.

warning: '*identifier*': bad type (not integral) [1]

The given bitfield is converted to an **unsigned** integral type.

warning: '*identifier*': bad type (not unsigned) [1]

The given bitfield is converted to an **unsigned** integral type.

warning: cast of int expression to far pointer [1]

A **far** pointer represents a full segmented address. Casting an integer value to a **far** pointer produces an address with a meaningless segment value.

warning: Constant too big [1]

Information is lost because a constant value is too large to be represented in the type to which it is assigned.

warning: conversion lost segment [1]

The conversion of a **far** pointer (a full segmented address) to a **near** pointer (a segment offset) results in the loss of the segment address.

warning: Data conversion [3]
Two data items had different types, causing the type of one item to be converted.

warning: '*operator*': different types [1]
The values specified in the operation have different types.

warning: Float constant in a cross compilation [1]
Floating point constants are not portable because the representation of floating point values differs across machines.

warning: '*identifier*': formals ignored [1]
Formal arguments appeared in a function declaration (for example, "extern int *f(a,b,c);"). The formal arguments are ignored.

warning: '*identifier*': function as an argument [1]
A formal parameter to a function is declared to be a function, which is illegal. The formal parameter is converted to a function pointer.

warning: Function must return a value [2]
A function is expected to return a value unless it is declared as void.

warning: function *identifier* too large for post−optimizer [0]
The named function was not optimized because insufficient program space was available. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning: '*identifier*': has bad class [1]
The specified storage class cannot be used in this context (for example, function parameters cannot be given extern class). The default storage class for that context is used in place of the illegal class.

warning: −S has precedence over −L [1]
You cannot create both a disassembled listing (−S) and an assembled listing (−L) with the same command. The −L option is ignored and a disassembled listing is created.

warning: Id truncated to '*identifier*' [1]
Only the first 31 characters of an identifier are significant.

warning: −C ignored (must also specify −P or −E or −EP) [1]
The −C option preserves comments in a preprocessed listing and takes effect only when you create such a listing with the −P, −E or −EP option.

warning: Ignoring unknown flag *option* [1]
The compiler does not recognize the given *option* and ignores it.

warning: Illegal null char [1]
The single quotes delimiting a character constant must contain one character. For example, the declaration "char a = ''" is illegal.

warning : '*operator*' : illegal pointer combination [1]
A pointer to a given type is forced to point to an object with a different type.

warning : '*operator*' : illegal with enums [1]
You may not use the given operator with enum values. The enum values are converted to int type.

warning : missing close paren after 'defined(id' [1]
The closing parenthesis is missing from an #if defined directive.

warning : Mixed near/far pointers [1]
A pointer is assigned to a pointer with a different size, resulting in the loss of a segment address from a far pointer or the addition of a segment address to a near pointer.

warning : Newline in string constant [1]
A newline character is not preceded by an escape character (\) in a string constant.

warning : '*identifier*' : no function return type [2]
A function declared to have void type returns a value.

warning : No return value [2]
A function declared to return a value does not do so.

warning : Not enough parameters [1]
The number of actual arguments specified with an identifier is less than the number of formal parameters given in the macro definition of the identifier.

warning : '&' on function/array, ignored [1]
The address of (&) operator is used incorrectly on a function or array.

warning : Only one of −P/−E/−EP allowed, −P selected [1]
Each of the −P, −E and −EP options produces a different kind of preprocessed listing; only one option can be used at a time.

warning : overflow in constant arithmetic [1]
The result of an operation exceeds 0x7fffffff.

warning : overflow in constant multiplication [1]
The result of an operation exceeds 0x7fffffff.

warning : '*identifier*' : overflows array bounds [1]
Too many initializers are present for the given array. The excess initializers are ignored.

warning : Pointer mismatch [1]
Pointers to different types of variables are used interchangeably.

warning : Procedure too large, loop inversion optimization missed but continuing [0]
Some optimizations for a function are skipped because insufficient program space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning : Procedure too large, skipping branch sequence optimization and continuing [0]
Some optimizations are skipped because insufficient program space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning : Procedure too large, skipping cross jump optimization and continuing [0]
Some optimizations for a function are skipped because insufficient program space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

warning: Recoverable heap overflow in post optimizer – some optimizations may be missed [0]
Some optimizations are skipped because insufficient program space is available for optimization. To correct this program, reduce the size of the function by breaking it down into two or more smaller functions.

warning : '*identifier*' : redeclaration ignored [1]
The named formal parameter was previously defined.

warning : *identifier*: redefinition [1]
The given *identifier* is redefined.

warning : 'register' on '*identifier*' ignored [1]
Only integral and pointer type variables may be given **register** storage class.

warning : "–i" required on the command line,changing name segment or group requires separate i and d. Setting /–i and continuing. [1]
The text segment of a small model program can be renamed (using –NT) only if a separate text segment is created using the –i option.

warning : requires parameters [1]
Formal parameters are given in the macro definition of an identifier, but no argument list is given with the identifier.

warning : Storage class *class* on '*identifier*' changed to extern [1]
Items declared outside of functions must have **static** or **extern** storage class.

warning : String too big, leading chars truncated [1]
Strings may not exceed 512 bytes.

warning : Strong type mis−match [2]

Two different but compatible types are used: for example, a **typedef** type with a non−**typedef** type, or two different but equivalent **struct** or **union** types.

warning : Too many parameters [1]

The number of actual arguments specified with an identifier is greater than the number of formal parameters given in the macro definition of the identifier.

warning : Type following '*keyword*' is illegal, ignored [1]

An illegal combination occurs (for example, **unsigned float**.)

warning : *identifier*: undefined [1]

The given *identifier* is not defined.

warning : '*identifier*' : unknown array size [1]

The size of the named array is not specified.

warning : '*identifier*' : unknown size [1]

The size of the named variable is not specified.

warning : unmatched close comment '*/' [1]

A comment was started (with '/*') but was not closed (with '*/').

warning : '*identifier*' : void type changed to int [1]

Only functions may be declared to have **void** type.

## D.2.2 Program Error Messages

The following is a complete list of program error messages. After printing a program error message, the compiler typically continues to look for more errors, but will not create an object file.

'+' : 2 pointers

Two pointers may not be added.

'*identifier*' : aggregate inits require curly braces

An initializer for an aggregate type has a syntax error.

Array of functions

Arrays of functions are not allowed.

auto allocation exceeds 32K

The space allocated for the local variables of a function exceeds the limit of 32K bytes.

'*identifier*' : automatic struct/arrays

Structures, arrays, and unions with **auto** storage class cannot be initialized.

Bad call

The expression before the parentheses in a function call does not evaluate to a function pointer. For example,

```
int *p;
  .
  .
  .
(*p)();
```

'*class*' : bad class

The given storage *class* cannot be used in this context.

*operator* : bad left operand

The left-hand operand of the given *operator* is an illegal value.

Bad octal number '*n*'.

The character *n* is not a valid octal digit.

*operator* : bad right operand

The right-hand operand of the given *operator* is an illegal value.

'*identifier*' : base type with near/far not allowed

Declarations of structure and union members may not use the **near** and **far** keywords to override the addressing convention for a member.

can't cast objects as 'far'

The **near** and **far** keywords may not be used in type casts. For example, "(int far)foo" is illegal.

can't cast objects as 'near'

The **near** and **far** keywords may not be used in type casts. For example, "(int near)foo" is illegal.

Case expression not constant

Case expressions must be integral constants.

Case expression not integral

Case expressions must be integral constants.

Case value '*n*' already used

The case value *n* has already been used in this **switch** statement.

cast of 'void' term to non-void

The **void** type may not be cast to any other type.

cast to array type is illegal

An object cannot be cast to an array type.

cast to function returning . . . is illegal

An object cannot be cast to a function type.

Compiler error (assertion): file *filename*,

line *n* source=*filename*

The compiler consistency check failed. Try rearranging your code. In this message, the first *filename* identifies the compiler file producing the error; the line number *n* refers to that file. The second *filename* gives the name of the source file being compiled.

Compiler error (code generation)

The compiler could not generate code for this expression. Try rearranging the expression.

Compiler error (internal): ....

The compiler consistency check failed. Try rearranging your code.

Compiler limit : macro's actual parameter is too big

Arguments to preprocessor macros may not exceed 256 bytes.

Compiler limit : struct/union nesting

Nesting of structure and union definitions may not exceed 5 levels.

Compiler limit : Too many actual parameters for macro

A macro definition may not take more than 8 actual arguments.

compiler limitation : Initializers too deeply nested

The compiler limit on nesting of initializers has been exceeded. The limit ranges from 10 to 15 levels, depending on the combination of types being initialized. To correct this problem, simplify the data type being initialized to reduce the levels of nesting, or assign initial values in separate statements after the declaration.

Constant expression is not integral

The context requires an integral constant expression.

#define syntax

A **#define** directive has a syntax error.

'*identifier*' : definition too big

Macro definitions may not exceed 256 bytes.

'*operator*' : different aggregate types

Pointers to different structure or union types are not allowed with the given *operator*.

Divide by 0

The second operand in a division (/) operation evaluates to zero (0).

'*identifier*' : enum/struct/union type redefinition

The given *identifier* has already been used for an enumeration, structure, or union tag in the same scope.

expected '(' to follow '*identifier*'
: The context requires parentheses after the function *identifier*.

Expected constant expression
: The context requires a constant expression.

expected 'defined(id)'
: An **#if defined** directive has a syntax error.

Expected exponent value, not '*n*'
: The exponent of a floating point constant is not a valid number.

Expected preprocessor command, found '*c*'
: The character following a number sign (#) is not the first letter of a preprocessor directive.

'*identifier*' : field is an array/ptr
: Bitfield members must have **unsigned** integral type.

'*identifier*' : field type too small for number of bits
: The number of bits specified in the bitfield declaration exceeds the number of bits in an **unsigned** integer of the given size.

'*identifier*' : fields only in structs
: Only structure types may contain bitfields.

Function returns array
: A function may not return an array. (It may return a pointer to an array.)

Function returns function
: A function may not return a function. (It may return a pointer to a function.)

'*identifier*' : Functions are illegal members
: A function cannot be a member of a structure; use a pointer to a function instead.

'*string*' : ignored
: The given text appeared out of context and was ignored.

Illegal allocation of segment > 64K
: The space allocated for a single data item exceeds the limit of one segment (64K bytes).

Illegal break
: A **break** statement is legal only when it appears within a **do**, **for**, **while**, or **switch** statement.

Illegal case
: The **case** keyword may only appear within a **switch** statement.

illegal cast

A type used in a cast operation is not a legal type.

Illegal continue

A **continue** statement is legal only when it appears within a **do**, **for**, or **while** statement.

Illegal default

The **default** keyword may only appear within a **switch** statement.

Illegal escape sequence

The character(s) after the escape character (\) do not form a valid escape sequence.

Illegal expression

An expression is illegal because of a previous error. (The previous error may not have produced an error message.)

'*operator*': illegal for struct/union

Structure and union type values are not allowed with the given *operator*.

Illegal index, indirection not allowed

A subscript was applied to an expression that does not evaluate to a pointer.

Illegal indirection.

The indirection operator ('*') was applied to a non-pointer value.

Illegal initialization

An initialization is illegal because of a previous error. (The previous error may not have produced an error message).

'*operator*': illegal pointer combination

Pointers that point to different types cannot be used with the given *operator*.

Illegal pointer subtraction.

Only pointers that point to the same type may be subtracted.

#include expected a file name

An **#include** directive lacks the mandatory filename specification.

'*identifier*': init of a function

Functions may not be initialized.

'*identifier*' is an undefined struct/union

The structure or union type of the given *identifier* is not defined.

keyword 'enum' illegal

The **enum** keyword appears in a structure or union declaration, or an **enum** type definition is not formed correctly.

Label '*identifier*' was undefined.
: The function does not contain a statement labeled with the given *identifier*.

left of '−>*identifier*' must have a struct/union type
: The expression before the member selection operator '−>' does not point to a structure or union type.

left of '.*identifier*' must have a struct/union type
: The expression before the member selection operator '.' does not have a structure or union type.

left of '−>' specifies undefined struct/union '*identifier*'
: The expression before the member selection operator '−>' points to a structure or union type that is not defined.

left of '.' specifies undefined struct/union '*identifier*'
: The expression before the member selection operator '.' has a structure or union type that is not defined.

*operator*: Left operand must be lval.
: The left operand of the given *operator* must be an lvalue.

#line expected a line number
: A **#line** directive lacks the mandatory line number specification.

'*identifier*': member of enum redefinition
: The given *identifier* has already been used for an enumeration constant, either within the same enumeration type or within another enumeration type in the same scope.

Missing '>'
: The closing angle bracket ('>') is missing from an **#include** directive.

Missing name following '<'
: An **#include** directive lacks the mandatory filename specification.

missing open paren after keyword 'defined'
: Parentheses must surround the identifier to be checked in an **#if defined** directive.

'*identifier*': Missing subscript
: To reference an element of an array you must use a subscript (for example, "A[6]").

Mod by 0
: The second operand in a remainder (%) operation evaluates to zero (0).

More than one default
: A **switch** statement contains too many **default** labels (only one is allowed).

'*operator*' needs lvalue.
The given *operator* must have an lvalue operand.

negative subscript
A value defining an array size was negative.

Newline in constant
A newline character in a character or string constant must be preceded by the backslash escape character (\).

No closing single quote
A newline character in a character constant must be preceded by the backslash escape character (\).

No struct definition
A structure or union type is used in a declaration without being defined.

Non-address expression
An attempt was made to initialize an item that is not an lvalue. For example, the declaration "int i, j = 1;" in the following example is illegal.

```
int i, j = i;
main()
{
    .
    .
    .
}
```

The declaration occurs outside of all functions, so it cannot be determined until link time (too late for initialization) whether *i* is a reference to a global variable defined and initialized elsewhere, or a definition of a global variable (with a default initial value of 0).

Non-constant offset
An initializer uses a non-constant offset. For example, the declaration "int i, j, *p = &i + j;" in the following example is illegal.

```
int i, j, *p = &i + j;
main()
{
    .
    .
    .
}
```

The declaration occurs outside of all functions, so it cannot be determined until link time (too late for initialization) whether *i* and *j* are references to global variables defined and initialized elsewhere, or definitions of global variables (with default initial values of 0).

Non-integer switch expression
Switch expressions must be integral.

Non−integral index
Only integral expressions are allowed in array subscripts.

'*identifier*' : not a function
The given *identifier* was not declared as a function but an attempt was made to use it as a function. For example,

```
int i;
  .
  .
  .
i();
```

'*identifier*' : not a label
The *identifier* specified in a **goto** statement does not correspond to a statement label.

'*identifier*' : not struct/union member
The given *identifier* is used in a context that requires a structure or union member.

'&' on bit field ignored
Bit fields cannot have their address taken.

'&' on constant
Only variables and functions can have their address taken.

'&' on register variable
Register variables cannot have their address taken.

parameter has type void
Only functions have **void** type, and formal parameters may not be functions.

pointer + non−integer
Only integral values may be added to pointers.

'*operator*' : pointer on left. Needs integral right.
The left operand of the given *operator* is a pointer; the right operand must be an integral value.

'+' : 2 pointers
Two pointers may not be added.

Preprocessor command must start as first non−white
Non−whitespace characters appear before the number sign (#) of a preprocessor directive on the same line.

'*identifier*' : redefinition
The given *identifier* was defined more than once in the same scope.

'.' requires struct/union name
The expression before the member selection operator '.' is not the name

of a structure or union.

'->' requires struct/union pointer
: The expression before the member selection operator '->' is not a pointer to a structure or union.

'-': right operand pointer
: If the left-hand operand in a subtraction (-) operation is not a pointer, the right-hand operand is not permitted to be a pointer.

Static procedure '*identifier*' not found.
: A forward reference was made to a missing static procedure.

Structure/Union comparison illegal
: You cannot compare a structure type to a union type. (You can, however, compare individual members of structure and unions).

Subscript on non-array.
: A subscript was used on a variable that is not an array.

syntax error
: This statement or the preceding statement is not formed correctly.

'*n*': too big for char
: The number *n* is too large to be represented as a character.

Too many chars in constant
: A character constant is limited to a single character. (Multi-character character constants are not supported).

Too many initializers.
: The number of initializers exceeds the number of objects to be initialized.

Typedef specifies different enum
: Two enumeration types defined with **typedef** are used to declare an item, but the enumeration types are different.

Typedef specifies different struct
: Two structure types defined with **typedef** are used to declare an item, but the structure types are different.

Typedef specifies different union
: Two union types defined with **typedef** are used to declare an item, but the union types are different.

'typedefs' both define indirection
: Two **typedef** types are used to declare an item and both **typedef** types have indirection. For example, the declaration of *p* in the following example is illegal.

```
typedef int *P_INT;
typedef short *P_SHORT;
 P_SHORT P_INT p;          /* this declaration is illegal */
```

'*identifier*' : undefined
The given *identifier* is not defined.

'*c*' : unexpected in formal list
The character *c* is misused in a macro definition's list of formal parameters.

'*c*' : unexpected in macro definition
The character *c* is misused in a macro definition.

unknown character '0x*n*'
The given hexadecimal number does not correspond to to a character in the C character set.

'*identifier*' : unknown size
A member of a structure or union has an undefined size.

'void' illegal with all types
The void type cannot be used in operations with other types.

'*expression*' was the use of the struct/union
An undefined structure or union type variable is used in the given *expression*.

## D.2.3 Fatal Error Messages

The following is a complete list of fatal error messages. After printing a fatal error message, the compiler terminates processing and returns control to the system.

fatal : Bad flag = *option*
The given *option* is illegal or inconsistent with another option appearing on the same line.

fatal : Bad parenthesis nesting
The parentheses in a preprocessor directive are not matched.

fatal : Bad preprocessor command '*string*'.
The characters following the number sign (#) do not form a preprocessor directive.

fatal : Cannot open '*filename*'
The compiler ran out of disk space, or the disk is protected against writing.

fatal : Compiler limit : Macro expansion too big
The expansion of a macro exceeds the space available for it.

fatal: Compiler limit: possibly a recursively defined macro
: The expansion of a macro exceeds the space available for it. Check to see whether the macro is recursively defined.

fatal: DGROUP data allocation exceeds 64K
: Long model allocation of variables to the default segment exceeds 64K.

fatal: #if[n]def expected an identifier
: You must specify an identifier with the **#ifdef** and **#ifndef** directives.

fatal: expected '#endif'
: An **#if**, **#ifdef**, **#ifndef**, or **#if defined** directive was not terminated with an **#endif** directive.

fatal: only one memory model allowed
: Conflicting memory model options appear on the command line.

fatal: Parser stack overflow, please simplify your program
: Your program cannot be processed because the space required to parse the program exceeds a compiler limit. To solve this problem, try to simplify your program.

fatal: Too many include files
: Nesting of **#include** directives exceeds the limit of 10 levels.

fatal: unexpected '#elif'
: The **#elif** directive is legal only when it appears within an **#if**, **#if defined**, **#ifdef**, or **#ifndef directive.**

fatal: unexpected '#else'
: The **#else** directive is legal only when it appears within an **#if**, **#if defined**, **#ifdef**, or **#ifndef** directive.

fatal: unexpected '#endif'
: An **#endif** directive appears without a matching **#if**, **#if defined**, **#ifdef**, or **#ifndef** directive.

fatal: Unexpected EOF
: The end of a file was encountered. This message appears when you have insufficient space on the default disk drive for the compiler to create the temporary files it needs. The space required is approximately three times the size of the source file.

fatal: Unknown configuration string '*string*'
: The configuration string given with the **−M** option contains an unrecognized character.

fatal: Unknown model type
: The configuration string given with the **−M** option contains an unrecognized character.

## D.3 Compiler Requirements and Limits

The following list summarizes the limits imposed by the C compiler. If your program exceeds any of these limits, an error message will inform you of the problem.

1. Disk Space

| | |
|---|---|
| Minimum disk space for compilation | 3 times source file size |

2. Declarations

| | |
|---|---|
| Maximum number of dimensions in an array | 5 dimensions |
| Maximum level of nesting for structure/union definitions | 5 levels |
| Maximum level of indirection | 5 levels |
| Maximum level of nesting for aggregate initializers (depends on the combination of aggregate types; higher levels of nesting are possible with array initialization than with structure and union initialization) | 10–15 levels |

3. Constants

| | |
|---|---|
| Maximum length of a string, including the terminating null character (\0) | 512 bytes |

4. Identifiers

| | |
|---|---|
| Maximum length of an identifier (characters in excess of this limit do not cause an error, but they are not significant) | 31 characters |

5. Preprocessor Directives

| | |
|---|---|
| Maximum size of a macro definition | 512 bytes |
| Maximum number of actual arguments to a macro definition | 8 arguments |
| Maximum length of an actual preprocessor argument | 256 bytes |
| Maximum level of nesting for **#if**, **#ifdef**, **#ifndef**, and **#if defined** directives | 32 levels |
| Maximum level of nesting for include files | 10 levels |

The compiler does not set explicit limits on the number and complexity of declarations, definitions, and statements in an individual function or in a program. If the compiler encounters a function or program that is too large or too complex to be processed, it produces an error message to that effect.

## D.4 Assembler Error Messages

This section lists and explains the messages displayed by the XENIX assembler. As displays a message whenever it encounters an error during processing. It displays a warning message whenever it encounters questionable statement syntax.

An end-of-assembly message is displayed at the end of processing, even if no errors occurred. The message contains a count of errors and warning messages it displayed during the assembly. The message has the form:

```
Warning  Fatal
Errors   Errors
n        n
```

This message is also copied to the source listing.

Error messages are listed in alphabetical order with a short explanation where necessary.

**Assembler Errors**

Already defined locally (Code 23)
: Tried to define a symbol as EXTERNAL that had already been defined locally.

Already had ELSE clause (Code 7)
: Attempt to define an ELSE clause within an existing ELSE clause (you cannot nest ELSE without nesting IF...ENDIF).

Already have base register (Code 46)
: Trying to double base register.

Already have index register (Code 47)
: Trying to double index address.

Block nesting error (Code 0)
: Nested procedures, segments, structures, macros, IRC, IRP, or REPT are not properly terminated. An example of this error is close of an outer level of nesting with inner level(s) still open.

Byte register is illegal (Code 58)
: Use of one of the byte registers in context where it is illegal. For example, PUSH AL.

Can't override ES segment (Code 67)
: Trying to override the ES segment in an instruction where this override is not legal. For example, store string.

Can't reach with segment reg (Code 68)
: There is no ASSUME that makes the variable reachable.

Can't use EVEN on BYTE segment (Code 70)
: Segment was declared to be byte segment and attempt to use EVEN was made.

Circular chain of EQU aliases (Code 83)
An alias EQU eventually points to itself.

Constant was expected (Code 42)
Expecting a constant and received something else.

CS register illegal usage (Code 59)
Trying to use the CS register illegally. For example, XCHG CS,AX.

Directive illegal in STRUC (Code 78)
All statements within STRUC blocks must either be comments preceded by a semicolon (;), or one of the Define directives.

Division by 0 or overflow (Code 29)
An expression is given that results in a divide by 0.

DUP is too large for linker (Code 74)
Nesting of DUP's was such that too large a record was created for the linker.

8087 opcode can't be emulated (Code 84)
Either the 8087 opcode or the operands you used with it produce an instruction that the emulator cannot support.

Extra characters on line (Code 1)
This occurs when sufficient information to define the instruction directive has been received on a line and superfluous characters beyond are received.

Field cannot be overridden (Code 80)
In a STRUC initialization statement, you tried to give a value to a field that cannot be overridden.

Forward needs override (Code 71)
This message is not currently used.

Forward reference is illegal (Code 17)
Attempt to forward reference something that must be defined in pass 1.

Illegal register value (Code 55)
The register value specified does not fit into the "reg" field (the reg field is greater than 7).

Illegal size for item (Code 57)
Size of referenced item is illegal. For example, shift of a double word.

Illegal use of external (Code 32)
Use of an external in some illegal manner. For example, DB M DUP(?) where M is declared external.

Illegal use of register (Code 49)
Use of a register with an instruction where there is no 8086 or 8088

instruction possible.

Illegal value for DUP count (Code 72)
DUP counts must be a constant that is not 0 or negative.

Improper operand type (Code 52)
Use of an operand such that the opcode cannot be generated.

Improper use of segment reg (Code 61)
Specification of a segment register where this is illegal. For example, an immediate move to a segment register.

Index displ. must be constant (Code 54)
Illegal use of index display.

Label can't have seg. override (Code 65)
Illegal use of segment override.

Left operand must have segment (Code 38)
Used something in right operand that required a segment in the left operand. (For example, ":.")

More values than defined with (Code 76)
Too many fields given in REC or STRUC allocation.

Must be associated with code (Code 45)
Use of data related item where code item was expected.

Must be associated with data (Code 44)
Use of code related item where data related item was expected. For example, MOV AX,<code-label>.

Must be AX or AL (Code 60)
Specification of some register other than AX or AL where only these are acceptable. For example, the IN instruction.

Must be index or base register (Code 48)
Instruction requires a base or index register and some other register was specified in square brackets, .

Must be declared in pass 1 (Code 13)
Assembler expecting a constant value but got something else. An example of this might be a vector size being a forward reference.

Must be in segment block (Code 69)
Attempt to generate code when not in a segment.

Must be record field name (Code 33)
Expecting a record field name but got something else.

Must be record or field name (Code 34)
Expecting a record name or field name and received something else.

Must be register (Code 18)
Register expected as operand but you furnished a symbol -- was not a register.

Must be segment or group (Code 20)
Expecting segment or group and something else was specified.

Must be structure field name (Code 37)
Expecting a structure field name but received something else.

Must be symbol type (Code 22)
Must be WORD, DW, QW, BYTE, or TB but received something else.

Must be var, label or constant (Code 36)
Expecting a variable, label, or constant but received something else.

Must have opcode after prefix (Code 66)
Use of one of the prefix instructions without specifying any opcode after it.

Near JMP/CALL to different CS (Code 64)
Attempt to do a NEAR jump or call to a location in a different CS ASSUME.

No immediate mode (Code 56)
Immediate mode specified or an opcode that cannot accept the immediate. For example, PUSH.

No or unreachable CS (Code 62)
Trying to jump to a label that is unreachable.

Normal type operand expected (Code 41)
Received STRUCT, FIELDS, NAMES, BYTE, WORD, or DW when expecting a variable label.

Not in conditional block (Code 8)
An ENDIF or ELSE is specified without a previous conditional assembly directive active.

Not proper align/combine type (Code 25)
SEGMENT parameters are incorrect.

One operand must be const (Code 39)
This is an illegal use of the addition operator.

Only initialize list legal (Code 77)
Attempt to use STRUC name without angle brackets, < >.

Operand combination illegal (Code 63)
Specification of a two-operand instruction where the combination specified is illegal.

Operands must be same or 1 abs (Code 40)
Illegal use of the subtraction operator.

Operand must have segment (Code 43)
Illegal use of SEG directive.

Operand must have size (Code 35)
Expected operand to have a size, but it did not.

Operand not in IP segment (Code 51)
Access of operand is impossible because it is not in the current IP segment.

Operand types must match (Code 31)
Assembler gets different kinds or sizes of arguments in a case where they must match. For example, MOV.

Operand was expected (Code 27)
Assembler is expecting an operand but an operator was received.

Operator was expected (Code 28)
Assembler was expecting an operator but an operand was received.

Override is of wrong type (Code 81)
In a STRUC initialization statement, you tried to use the wrong size on override. For example, 'HELLO' for DW field.

Override with DUP is illegal (Code 79)
In a STRUC initialization statement, you tried to use DUP in an override.

Phase error between passes (Code 6)
The program has ambiguous instruction directives such that the location of a label in the program changed in value between pass 1 and pass 2 of the assembler. An example of this is a forward reference coded without a segment override where one is required. There would be an additional byte (the code segment override) generated in pass 2 causing the next label to change. You can use the −D option to produce a listing to aid in resolving phase errors between passes. See *as*(CP) in the XENIX *Reference Manual*.

Redefinition of symbol (Code 4)
This error occurs on pass 2 and succeeding definitions of a symbol.

Reference to mult defined (Code 26)
The instruction references something that has been multi−defined.

Register already defined (Code 2)
This will only occur if the assembler has internal logic errors.

Register can't be forward ref (Code 82)

Relative jump out of range (Code 53)
: Relative jumps must be within the range −128 to +127 of the current instruction, and the specific jump is beyond this range.

Segment parameters are changed (Code 24)
: List of arguments to SEGMENT were not identical to the first time this segment was used.

Shift count is negative (Code 30)
: A shift expression is generated that results in a negative shift count.

Should have been group name (Code 12)
: Expecting a group name but something other than this was given.

Symbol already different kind (Code 15)
: Attempt to define a symbol differently from a previous definition.

Symbol already external (Code 73)
: Attempt to define a symbol as local that is already external.

Symbol has no segment (Code 21)
: Trying to use a variable with SEG, and the variable has no known segment.

Symbol is multi−defined (Code 5)
: This error occurs on a symbol that is later redefined.

Symbol is reserved word (Code 16)
: Attempt to use an assembler reserved word illegally. For example, to declare MOV as a variable.

Symbol not defined (Code 9)
: A symbol is used that has no definition.

Symbol type usage illegal (Code 14)
: Illegal use of a PUBLIC symbol.

Syntax error (Code 10)
: The syntax of the statement does not match any recognizable syntax.

Type illegal in context (Code 11)
: The type specified is of an unacceptable size.

Unexpected end of file (Code 85)
: You forgot an end statement or there is a nesting error.

Unknown symbol type (Code 3)
: Symbol statement has something in the type field that is unrecognizable.

Usage of ? (indeterminate) bad (Code 75)
: Improper use of the "?". For example, ? + 5.

Value is out of range (Code 50)

Value is too large for expected use. For example, MOV AL,5000.

Wrong type of register (Code 19)

Directive or instruction expected one type of register, but another was specified. For example, INC CS.

**Numerical List of Messages**

| Code | Message |
|---|---|
| 0 | Block nesting error |
| 1 | Extra characters on line |
| 2 | Register already defined |
| 3 | Unknown symbol type |
| 4 | Redefinition of symbol |
| 5 | Symbol is multi-defined |
| 6 | Phase error between passes |
| 7 | Already had ELSE clause |
| 8 | Not in conditional block |
| 9 | Symbol not defined |
| 10 | Syntax error |
| 11 | Type illegal in context |
| 12 | Should have been group name |
| 13 | Must be declared in pass 1 |
| 14 | Symbol type usage illegal |
| 15 | Symbol already different kind |
| 16 | Symbol is reserved word |
| 17 | Forward reference is illegal |
| 18 | Must be register |
| 19 | Wrong type of register |
| 20 | Must be segment or group |
| 21 | Symbol has no segment |
| 22 | Must be symbol type |
| 23 | Already defined locally |
| 24 | Segment parameters are changed |
| 25 | Not proper align/combine type |
| 26 | Reference to mult defined |
| 27 | Operand was expected |
| 28 | Operator was expected |
| 29 | Division by 0 or overflow |
| 30 | Shift count is negative |
| 31 | Operand types must match |
| 32 | Illegal use of external |
| 33 | Must be record field name |
| 34 | Must be record or field name |
| 35 | Operand must have size |
| 36 | Must be var, label or constant |
| 37 | Must be structure field name |
| 38 | Left operand must have segment |
| 39 | One operand must be const |
| 40 | Operands must be same or 1 abs |
| 41 | Normal type operand expected |
| 42 | Constant was expected |

| | |
|---|---|
| 43 | Operand must have segment |
| 44 | Must be associated with data |
| 45 | Must be associated with code |
| 46 | Already have base register |
| 47 | Already have index register |
| 48 | Must be index or base register |
| 49 | Illegal use of register |
| 50 | Value is out of range |
| 51 | Operand not in IP segment |
| 52 | Improper operand type |
| 53 | Relative jump out of range |
| 54 | Index displ. must be constant |
| 55 | Illegal register value |
| 56 | No immediate mode |
| 57 | Illegal size for item |
| 58 | Byte register is illegal |
| 59 | CS register illegal usage |
| 60 | Must be AX or AL |
| 61 | Improper use of segment reg |
| 62 | No or unreachable CS |
| 63 | Operand combination illegal |
| 64 | Near JMP/CALL to different CS |
| 65 | Label can't have seg. override |
| 66 | Must have opcode after prefix |
| 67 | Can't override ES segment |
| 68 | Can't reach with segment reg |
| 69 | Must be in segment block |
| 70 | Can't use EVEN on BYTE segment |
| 71 | Forward needs override |
| 72 | Illegal value for DUP count |
| 73 | Symbol already external |
| 74 | DUP is too large for linker |
| 75 | Usage of ? (indeterminate) bad |
| 76 | More values than defined with |
| 77 | Only initialize list legal |
| 78 | Directive illegal in STRUC |
| 79 | Override with DUP is illegal |
| 80 | Field cannot be overridden |
| 81 | Override is of wrong type |
| 82 | Register can't be forward ref |
| 83 | Circular chain of EQU aliases |
| 84 | 8087 opcode can't be emulated |
| 85 | Unexpected end of file |

## D.5 Linker Error Messages

This section lists and explains the messages displayed by the XENIX linker. Ld displays a message whenever it encounters an error during processing.

Array element size mismatch

A far communal array has been declared with two or more different array element sizes (e.g. declared once as an array of characters and once as an array of reals). Match definitions and recreate object module.

Attempt to access data outside segment bounds

One of the object modules is invalid. Try recompiling the invalid module. If the link still fails, note exactly how the module was compiled and report the bug to Microsoft.

Attempt to put segment *name* in more than one group in file *filename*

A segment was declared to be a member of two different groups. Correct the source and recreate the object files.

Cannot find file *filename*

Specified file cannot be found. Try again after locating the file in question.

Cannot open list file

The directory or disk is full. Make space on the disk or in the directory.

Cannot open run file

The directory or disk is full. Make space on the disk or in the directory.

Cannot open temporary file

The directory or disk is full. Make space on the disk or in the directory.

Common area longer than 65536 bytes

User's program has more than 64K of communal variables. At the present time, only C language programs can possibly cause this message to be displayed. Rewrite your program using fewer communal variables or making some of your communal variables far, or recompile your program large model.

Data record too large

LEDATA record contains more than 1024 bytes of data. This is a translator error.

Dup record too large

LIDATA record contains more than 512 bytes of data. Most likely, an assembly module contains a struc definition that is very complex, or a series of deeply nested DUP statements (e.g. table db 10 dup(11 dup (12 dup(13 dup(...))))). Simplify and reassemble.

Error accessing library

File in question is an invalid library. Use a valid library.

Fixup overflow near *num* in segment *name* in *filename*(*name*) offset *num*

A fixup overflow can be caused by: 1) a group larger than 64K bytes, 2) the user's program contains an intersegment short jump or intersegment short call, 3) the user has a data item whose name conflicts with that of a subroutine in a library included in the link, and 4) an assembly language source file has an EXTRN declaration for a far procedure inside the body of a segment.

Group *name* larger than 64Kbytes

User has defined a group containing more than 64Kbytes of code or data. Make the offending group smaller and relink.

Invalid object module

One of the object modules is invalid. Try recompiling.

List file name missing

Name missing after −m option. Try again with correct command line.

Multiple code segments −− should be medium model

User's program contains more than one code segment, and the user has not informed the linker that the program is middle or large model. Unless the program is hybrid model, relink using −Mm option.

Multiple data segments −− should be large model

User's program contains more than one data segment, and the user has not informed the linker that the program is large model. Unless the program is hybrid model, relink using −Ml option.

Name length missing

Number missing after the −nl option. Try again with correct command line.

NEAR/HUGE conflict

Conflicting near and huge definitions for a communal variable. Revise definitions to be consistent (Note: a communal variable is "huge" if it is larger than 65536 bytes).

No object files specified

No object files were specified on the command line and the −u option was not used. Try again with correct command line.

No object modules specified

User failed to supply the linker with any object file names. Try again.

Out of space on list file

Disk on which list file is being written is full. Free more space on the disk and try again.

Out of space on run file

Disk on which executable is being written is full. Free more space on the disk and try again.

Out of space on scratch file

Disk in default drive is full. Delete some files on that disk, or replace with another diskette, and restart the linker.

Relocation table overflow

More than 16384 long calls or long jumps or other long pointers in the user's program. Rewrite program replacing long references with short references where possible and recreate object module.

Run file name missing

Name missing after the −o option. Try again with correct command line.

Segment limit set too high

The limit on the number of segments allowed was set higher than 1024 using the −S option. Try link again with a smaller number.

Segment limit too high

There is insufficient memory for the linker to allocate tables to describe the number of segments requested (either the value specified with −S or the default: 128). Try the link again using −S to select a smaller number of segments (e.g. 64, if the default was used previously).

Segment size exceeds 64K

User has a small model program with more than 64Kbytes of code, or user has a middle model program with more than 64Kbytes of data. Try compiling and linking middle or large model.

Stack size exceeds 65536 bytes

The value specified using the −F option exceeds 0x10000. Try again.

Stack size missing

Number missing after −F option. Try again with correct command line.

Symbol missing

Symbol missing after the −u option. Try again with correct command line.

Symbol table overflow

The user's program has greater than 256K of symbolic information (Publics, externs, segments, groups, classes, files, etc). Combine modules and/or segments and recreate the object files. Eliminate as many public symbols as possible.

Terminated by user

The user pressed the delete key.

Too many external symbols in one module

User's object module specified more than the allowed number of external symbols. Break up the module.

Too many group−, segment−, and class−names in one module

User's program contains too many group, segment, and class names. Reduce the number of groups, segments, or classes and recreate the object files.

Too many groups

User's program defines more than nine groups. Reduce the number of groups.

Too many GRPDEFs in one module

Linker encountered more than 9 GRPDEFs in a single module. Reduce the number of GRPDEFs or split up the module.

Too many libraries
: User tried to link with more than 32 libraries. Combine libraries or link modules that require fewer libraries.

Too many segments in one module
: The user's object module has more than 255 segments. Split the modules or combine segments.

Too many segments
: The user's program has too many segments. Relink using the −S option with an appropriate number of segments specified.

Too many TYPDEFs
: TYPDEFs are records emitted by the compiler to describe communal variables. Create two sources from the old source, dividing the communal variable definitions between them; recompile and relink.

Unexpected end−of−file on library
: The diskette containing the library has probably been removed. Try again after replacing the diskette with the library.

Unexpected end−of−file on scratch file
: The Diskette containing the VM.TMP file was removed. Try again after replacing the diskette with the VM.TMP file.

Unknown model specifier '−M*x*'
: *x* was none of the following: s, m, or l. Try again with correct command line.

Unknown option '−*x*'
: Specified option is not recognized by the linker. Try again with correct command line.

Unrecognized Xenix version number
: Number after −v option was neither 2 nor 3. Try again with correct command line.

Use −i option
: User's program is not small model impure (i.e., it consists of more than one segment). Relink using the −i option.

Version number missing
: Number missing after −v option. Try again with correct command line.

Warning: Groups *name* and *name* overlap
: User's program contains overlapping groups. Unless one group is completely contained by the other, fix the source code, recompile, and relink.

Warning: model mismatch

One or more object modules were not compiled using the memory model specified by the −M option. Recompile the offending module and relink.

Warning: no stack segment

User's program contains no segment of combine−type stack.

Warning: too many public symbols

The user has asked for a sorted listing of public symbols in the list file, but there are too many symbols to sort. The linker will produce an unsorted listing of the public symbols.

−u seen before −nl

User has specified a symbol to look for (using the −u option) before specifying the maximum symbol length with the −nl option. Try again placing the −nl option and its argument before all −u options and their arguments.

# Index

# Contents

## *Programming Commands* (CP)

| | |
|---|---|
| **strings** | Finds the printable strings in an object file. |
| **strip** | Removes symbols and relocation bits. |
| **time** | Times a command. |
| **tsort** | Sorts a file topologically. |
| **unget** | Undoes a previous get of an SCCS file. |
| **val** | Validates an SCCS file. |
| **xref** | Cross-references C programs. |
| **xstr** | Extracts strings from C programs. |
| **yacc** | Invokes a compiler-compiler. |

**Name**

intro – Introduces XENIX Development System commands.

**Description**

This section describes use of the individual commands available in the XENIX Development System. Each individual command is labeled with the letters CP to distinguish it from commands available in the XENIX Operating and Text Processing Systems. These letters are used for easy reference from other documentation. For example, the reference *cc*(CP) indicates a reference to a discussion of the cc command in this section, where the letter "C" stands for "Command" and the letter "P" stands for "Programming".

**Syntax**

Unless otherwise noted, commands described in this section accept options and other arguments according to the following syntax:

*name* [*options*] [*cmdarg*]

where:

| | |
|---|---|
| *name* | The filename or pathname of an executable file |
| *option* | A single letter representing a command option. By convention, most options are preceded with a dash. Option letters can sometimes be grouped together as in **–abcd** or alternatively they are specified individually as in **–a –b –c –d** . The method of specifying options depends on the syntax of the individual command. In the latter method of specifying options, arguments can be given to the options. For example, the **–f** option for many commands often takes a following filename argument. |
| *cmdarg* | A pathname or other command argument *not* beginning with a dash. It may also be a dash alone by itself indicating the standard input. |

**See Also**

getopt(C), getopt(S)

**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*(S) and *exit*(S)). The former byte is 0 for normal termination; the latter is customarily 0 for 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.

## Name

adb – debugger

## Syntax

**adb** [– **w**] [ objfil [ corfil ] ]

## Description

*Adb* is a general purpose debugging program. It may be used to examine files and to provide a controlled environment for the execution of XENIX programs.

*Objfil* is normally an executable program file, preferably containing 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.** *Corfil* is assumed to be a core image file produced after executing *objfil*; the default for *corfil* is **core.**

Requests to *adb* are read from the standard input and responses are to the standard output. If the – **w** flag is present then both *objfil* and *corfil* are created if necessary and opened for reading and writing so that files can be modified using *adb*. *Adb* ignores QUIT (CTRL-\); INTERRUPT (DEL) causes return to the next *adb* command.

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 0. For most commands *count* specifies how many times the command will be executed. The default *count* is 1. *Address* and *count* are expressions.

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 further details of address mapping see Addresses.

**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. \ may be used to escape a '.

< *name*
The value of *name*, which is either a variable name or a register name. *Adb* maintains a number of variables (see Variables) 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 *corfil*.

*symbol* A *symbol* is a sequence of upper or lower case letters, underscores or digits, not starting with a digit. The value of the *symbol* is taken from the symbol table in *objfil*. An initial _ or ~ 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* in *corfil*. |
| @ *exp* | The contents of the location addressed by *exp* in *objfil*. |
| − *exp* | Integer negation. |
| ~*exp* | Bitwise complement. |

**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 conjunction. |
| *e1*\|*e2* | Bitwise disjunction. |
| *e1*#*e2* | *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 Addresses for further details.)

| | |
|---|---|
| ?*f* | Locations starting at *address* in *objfil* are printed according to the format *f*. |
| /*f* | Locations starting at *address* in *corfil* 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
: Print 2 bytes in octal. All octal numbers output by *adb* are preceded by 0.

**O** 4
: Print 4 bytes in octal.

**q** 2
: Print in signed octal.

**Q** 4
: Print long signed octal.

**d** 2
: Print in decimal.

**D** 4
: Print long decimal.

**x** 2
: Print 2 bytes in hexadecimal.

**X** 4
: Print 4 bytes in hexadecimal.

**u** 2
: Print as an unsigned decimal number.

**U** 4
: Print long unsigned decimal.

**f** 4
: Print the 32 bit value as a floating point number.

**F** 8
: Print double floating point.

**b** 1
: Print the addressed byte in octal.

**c** 1
: Print the addressed character.

**C** 1
: Print the addressed character using the following escape convention. Character values 000 to 040 are printed as @ followed by the corresponding character in the range 0100 to 0140. The character @ is printed as @@.

**s** *n*
: Print the addressed characters until a zero character is

reached.

**S** *n*

Print a string using the @ escape convention. *n* is the length of the string including its zero terminator.

**Y** 4

Print 4 bytes in date format (see *ctime*(3)).

**i** n

Print as PDP11 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

Print 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 |

**p** 2

Print the addressed value in symbolic form using the same rules for symbol lookup as **a**.

**t** 0

When preceded by an integer tabs to the next appropriate tab stop. For example, **8t** moves to the next 8-space tab stop.

**r** 0

Print a space.

**n** 0

Print a newline.

**"..."** 0

Print the enclosed string.

^ *Dot* is decremented by the current increment. Nothing is printed.

+ *Dot* is incremented by 1. Nothing is printed.

- *Dot* is decremented by 1. Nothing is printed.

newline

If the previous command temporarily incremented *dot*, make 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* ...

Write the 2-byte *value* into the addressed location. If the command is **W**, write 4 bytes. Odd addresses are not allowed when writing to the subprocess address space.

[**?** /]**m** *b1 e1 f1*[**?** /]

New values for (*b1, e1, f1*) are recorded. If less than three expressions are given then the remaining map parameters are left unchanged. If the '?' or '/' is followed by '*' then the second segment (*b2*, *e2*, *f2*) of the mapping is changed. If the list is terminated by '?' or '/' then the file (*objfil* or *corfil* respectively) is used for subsequent requests. (So that, for example, '/m?' will cause '/' to refer to *objfil*.)

>*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 **pc**. *Dot* is set to **pc**.
- **f** Print the floating registers in single or double length. If the floating point status of **ps** is set to double (0200 bit) then double length is used anyway.
- **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 **r5**). 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** All integers input are regarded as octal.
- **d** Reset integer input as described in Expressions.
- **q** Exit from *adb*.
- **v** Print all non zero variables in octal.
- **m** Print the address map.

**:***modifier*

Manage a subprocess. Available modifiers are:

**b***c* 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.

**d** Delete breakpoint at *address*.

**r** 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 on entry to the subprocess.

**c***s* The subprocess is continued with signal *s* c *s*, see *signal*(2). 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**.

**s***s* As for **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 *corfil*. If *corfil* does not appear to be a **core** file then these values are set from *objfil*.

| | |
|---|---|
| b | The base address of the data segment. |
| d | The data segment size. |
| e | The entry point. |
| s | The stack segment size. |
| t | The text segment size. |

## Addresses

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by two triples (*b1, e1, f1*) and (*b2, e2, f2*) and the *file address* corresponding to a written *address* is calculated as follows.

$b1 \leq address < e1$ => *file address*=*address*+ *f1*- *b1*, otherwise,

$b2 \leq address < e2$ => *file address*=*address*+ *f2*- *b2*,

otherwise, the requested *address* is not legal. In some cases (e.g. for programs with separated I and D space) the two segments for a file may overlap. If a **?** or / is followed by an * then only the second triple is used.

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, *b1* is set to 0, *e1* is set to the maximum file size and *f1* is set to 0; 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

a.out(F), core(F)

## Diagnostics

'Adb' when there is no current command or format. Comments about inaccessible files, syntax errors, abnormal termination of commands, etc. Exit status is 0, unless last command failed or returned nonzero status.

## Name

admin – Creates and administers SCCS files.

## Syntax

admin [– n] [– i[name]] [– rrel] [– t[name]] [– fflag[flag-val]] [– dflag[flag-val]] [– alogin] [– elogin] [– m[mrlist]] [– y[comment]] [– h] [– z] files

## Description

*Admin* is used to create new SCCS files and to change parameters of existing ones. Arguments to *admin*may appear in any order. They consist of options, which begin with – , and named files (note that SCCS filenames must begin with the characters s.). If a named file doesn't exist, it is created, and its parameters are initialized according to the specified options. Parameters not initialized by a 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 nonSCCS files (last component of the pathname 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, nonSCCS 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 – r below for delta numbering scheme). If the i option is used, but the filename 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 require 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 also used), the descriptive text filename must also be supplied. In the case of existing SCCS files: a – t option without a filename causes removal of descriptive text (if any) currently in the SCCS file, and a – t option with a filename causes text (if any) in the named file to replace the descriptive text (if any) currently in the SCCS file.

– f*flag* 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 *flags* and their values are:

b Allows use of the – b option on a *get*( CP) command to create branch deltas.

c*ceil* The highest release (i.e., "ceiling"), a number less than or equal to 9999, which may be retrieved by a *get*(CP) command for editing. The default value for an unspecified c flag is 9999.

f*floor* The lowest release (i.e., "floor"), a number greater than 0 but less than 9999, which may be retrieved by a *get*(CP) command for editing. The default value for an unspecified f flag is 1.

d*SID* The default delta number (SID) to be used by a *get*( CP) command.

i Causes the "No id keywords (ge6)" message issued by *get*(CP) or *delta*(CP) 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*(CP)) are found in the text retrieved or stored in the SCCS file.

j Allows concurrent *get*(CP) 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 (get – e against one of these "locked" releases fails). The *list* has the following syntax:

<list> ::= <range> | <list> , <range>
<range> ::= *RELEASE NUMBER* | a

The character a in the *list* is equivalent to specifying *all releases* for the named SCCS file.

n Causes *delta*(CP) to create a "null" delta in each of those releases (if any) being skipped when a delta is made in a *new* release (e.g., 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 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*( CP).

m*mod* *Mod*ule name of the SCCS file substituted for all occurrences of the admin.CP keyword in SCCS file text retrieved by *get*(CP). 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*( CP).

v[*pgm*] Causes *delta*(CP) 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 *delta*(CP)). (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 which 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*(CP). 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 (i.e., 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*(CP). 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 MR validation fails.

− h Causes *admin* to check the structure of the SCCS file (see *sccsfile*(F)), 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, 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**, above).

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 filenames must be of the form s.*file-name.* New SCCS files are created read-only (444 modified by umask) (see *chmod*(C)). 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*(CP)), created with read-only permission 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 755 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 644 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 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*(CP) for further information.

## See Also

delta(CP), ed(C), get(CP), help(CP), prs(CP), what(C), sccsfile(F)

## Diagnostics

Use *help*(CP) for explanations.

**Name**

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 **vualbcln**. *Afile* is the archive file. The *names* are constituent files in the archive file. The *posname* is the name of a constituent file, and is required when certain keys are used. The meanings of the *key* characters are:

**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 and 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**, 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. Unless the optional character n is used with x, an extracted file's modification date will be set to the date stored in that file's archive header. In neither case does **x** alter the archive file.

**v** Verbose. Under the verbose option, *ar* gives a file−by−file description of the making 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**, 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.

**n** New. When used with the *key* character **x** it sets the extracted file's modification date to the current date.

When *ar* creates an archive, it always creates the header in the format of the local system (see *ar*(F)).

**Files**

/tmp/v* Temporary files

**See Also**

ld(CP), lorder(CP), ar(F)

**Notes**

If the same file is mentioned twice in an argument list, it may be put in the archive twice.

## Name

as – Xenix 8086/186/286 Assembler.

## Syntax

as [ *options* ] *source-file*

## Description

*As* assembles 8086/186/286 assembly language source files and produces linkable object modules. The command accepts one *source-file*. The source file name must have the ".s" extension. The resulting file containing the object module is given the same base name as the source, with the ".o" extension replacing the ".s" extension.

There are the following options:

**-a** Assembled segments are output in alphabetic order, instead of in order of occurrence in the source file.

**-d** Creates program listings for both passes of the assembler. This listing can be used to resolve phase errors between assembler passes. The **-d** option is ignored if the -l option is not in effect.

-l Produces a listing file. The listing file has the same base name as the source file, but has the ".lst" extension.

**-Mu** Disables case sensitivity for all names and symbols. This option makes upper and lowercase letters in names and symbols indistinguishable to the assembler. This option also causes the symbols defined by the EXTRN and PUBLIC directives to be output in uppercase regardless of their original spelling.

**-Mx** Disables case sensitivity for all names and symbols except those names defined by the EXTRN and PUBLIC directives. This option is similar to the **-Mu** option except that public and external names copied to the object file retain their original spelling.

**-n** Suppresses the generation of the symbol table in the program listing. This option is ignored if the -l option is not in effect.

**-o** *filename*
Directs the generated object module to the file named *filename*. No default extension is assumed.

**-O** Causes values in the program listing to be displayed in octal. The default radix is hexadecimal.

**-r** Causes generation of actual 8087/287 instructions instead of software interrupts for the floating point emulation package. Object modules created using this option can only be executed on machines with an 8087 or 287.

**-X** Directs the assembler to list any conditional block whose IF condition resolves to false. This option can be overridden in the source file by using the .TFCOND directive. This option is ignored if the -l option is not in effect.

By default, *as* recognizes 8086 instruction mnemonics only. To assemble 186, 286, 8087, or 287 instructions, the corresponding .186, .286c, .286p, .8087, or .287 directive must be given in the source file.

## Files

/bin/as

## See Also

cc(C), ld(CP), *XENIX Programmer's Guide*

## Note

Unless the **-r** is given, **as** assumes all 8087/287 instructions are to be carried out using floating point emulation. The **-r** option should only be used on machines with an 8087 or 287 coprocessor.

## Name

cb – Beautifies C programs.

## Syntax

```
cb [file]
```

## Description

*Cb* places a copy of the C program in *file* (standard input if *file* is not given) on the standard output with spacing and indentation that displays the structure of the program.

**Name**

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* can name any C or assembly language source file or any object or library file. C source files must have a ".c" filename extension. Assembly language source files must ".s", object files ".o", and library files ".a" extensions. *Cc* invokes the C compiler for each C source file and copies the result to an object file whose basename is the same as the source file but whose extension is ".o". *Cc* invokes the XENIX assembler, *as* , for each assembly source file and copies the result 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 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*(CP) 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 library functions and program startup routines. Which libraries are used depends on the program's memory model (see "Memory Models" below). The entry point of the resulting program is set to the beginning of the "main" program function.

There are the following options:

**– P**
Preprocesses each source file and copies the result to a file whose basename is the same as the source but whose extension is ".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.

**– EP**
Preprocesses each source file as described for **– E** , but does not place a #line directive at the beginning of the file.

**– C**
Preserves comments when preprocessing a file with **– E** or **– P**. That is, comments are not removed from the preprocessed source. This option may only be used in conjunction with **– E** or **– P** .

**– D** *name [ = string ]*
Defines *name* to the preprocessor as if 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 filename. If the file 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 1, only warnings about program structure and overt type mismatches are issued. If 2, warnings about strong typing mismatches are issued. If 3, warnings for all automatic conversions are issued. This option does not affect compiler 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 using the *prof*( CP) command.

**– 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 may be shared by all users executing the file. The option is implied when creating middle or large model program. (Not implemented on all machines.)

**– F** ***num***
Sets the size of the program stack to ***num*** bytes. Default stack size if not given, is 2 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, data threshold. It also enables C language enhancments such as advanced instruction set and keywords. The ***string*** may be any combination of the following (the "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. High order word is first. Default is low order word first. |
| t ***num*** | Sets the size of the largest data item in the data 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**.

**– dos**

Directs **cc** to create an executable program for MS-DOS systems.

**– l*library***

Searches ***library*** for unresolved references to functions. The ***library*** must be an object file archive library in **ranlib** format.

**– O**

Invokes the object code optimizer.

**– S**

Creates an assembly source listing of the compiled C source file and copies this listing to the file whose basename is the same as the source but whose extension is ".s". It should be noted that this file is not suitable for assembly. This option provides code for reading only.

**– L**
Creates an assembler listing file containing assembled code and assembly source instructions. The listing is copied to the file whose basename 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 filename 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" 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.

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 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. These 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 64 Kbyte physical segment in which both text and data are combined. *Cc* creates impure small model programs by default. They can also be created using the "-Ms" option.

Pure-Text Small Model

These programs occupy two 64 Kbyte physical segments. 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 128 Kbytes. Pure small model programs are created 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 call and returns are used to access functions in other segments. Text can be any size. Data must not exceed 64 Kbytes. Middle models programs are created 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 calls and returns are used to access functions in other segments. Special addresses are used to access data in other segments. Text and data may be any size, but no data item may be larger than 64 Kbytes. Large model programs are created using the "-Ml" option. These programs are always pure.

Small, middle, and large model object files can only be linked with object and library files of the same model. It is not possible to combine small, medium, and large model object files in one executable program. *Cc* automatically selects the correct small, middle, or large versions of the standard libraries based on the configuration option. It is up to the user to make sure that all of his own 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 which makes heavy use of functions and function pointers may differ noticably 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*(CP) will help to check for correct use.

The - NM, - NT, and - ND 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. Similarly, all data having the same data segment name is placed in a single physical segment.

## Files

/bin/cc

## See Also

as(CP), ar(CP), ld(CP), lint(CP), ranlib(CP)

## 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 the link loader.

All object module libraries must have a current *ranlib* directory.

## Name

cdc - Changes the delta commentary of an SCCS delta.

## Syntax

**cdc - rSID [- m[mrlist]] [- y[comment]] files**

## Description

*Cdc* changes the delta commentary for the *SID* specified by the **- r** option, of each named SCCS file.

*Delta commentary* is defined to be the Modification Request (MR) and comment information normally specified via the *delta*(CP) 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 nonSCCS files (last component of the pathname does not begin with **s.**) and unreadable files are silently ignored. If a name of - is given, the standard input is read (see Warning); 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 options and file names.

All the described options apply independently to each named file:

- r*SID* Used to specify the *S*CCS *ID*entification (*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 *admin*(CP)) 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*(CP). In order to delete an MR, precede the MR number with the character **!** (see Examples). 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 – **y** option).

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*(CP)), it is taken to be the name of a program (or shell procedure) which 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 made the delta, you can change its delta commentary; or if you own the file and directory you can modify the delta commentary.

## Examples

The following:

```
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 does the same thing.

```
cdc – r1.6 s.file
MRs? !bl77-54321 bl78-12345 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*(CP) |
| z-file | See *delta*(CP) |

## See Also

admin(CP), delta(CP), get(CP), help(CP), prs(CP), sccsfile(F)

## Diagnostics

Use *help*(CP) for explanations.

## Name

comb – Combines SCCS deltas.

## Syntax

comb [– o] [– s] [– psid] [– clist] files

## 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*(C)).

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 nonSCCS files (last component of the pathname 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.

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 *S*CCS *ID*entification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

– c*list* A *list* (see *get*(CP) for the syntax of a *list*) of deltas to be preserved. 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 filename, 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 options 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(CP), delta(CP), get(CP), help( CP), prs(CP), sccsfile(F)

## Diagnostics

Use *help*(CP) 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.

## Name

config – Configures a XENIX system.

## Syntax

**/etc/config** [– **t**] [– **c** file] [– **m** file] dfile

## Description

*Config* is a program that takes a description of a XENIX system and generates a file which is a C program defining the configuration tables for the various devices on the system.

The – **c** option specifies the name of the configuration table file; **c.c** is the default name.

The – **m** option specifies the name of the file that contains all the information regarding supported devices; **/etc/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 user must supply *dfile*; 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, such as the system clock, which must be present to run XENIX . 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 the device, and *number* is the number (decimal) of devices associated with the corresponding controller. The device name can be any name given in part 1 of the **/etc/master** file, or any alias given in part 3 of the same file; *number* is optional, and if omitted, a default value which is the maximum value for that controller is used.

There are certain drivers that may be provided with the system, that are actually *pseudo-device* drivers; that is, there is no real hardware associated with the driver. If the system has such drivers, they are described in section M of the XENIX *Reference Manual.*

## Second Part of *dfile*

The second part contains three different types of lines. Note that *all* specifications of this part *are required,* although their order is arbitrary.

1. *Root/pipe device specification*

   Two lines, each having three fields:

   ```
   root   devname     minor
   pipe   devname     minor
   ```

   where *devname* is the name of the device, and *minor* is the minor device number (in octal). The device name can be any name given in part 1 of the **/etc/master** file, or any alias given in part 3 of the same file.

2. *Swap device specification*

   One line that contains five fields as follows:

   ```
   swap   devname     minor  swplo  nswap
   ```

   where *devname* is the name of the device, *minor* is the minor device number (in octal), *swplo* is the lowest disk block (decimal) in the swap area, and *nswap* is the number of disk blocks (decimal) in the swap area. The device name can be any name given in part 1 of the **/etc/master** file, or any alias given in part 3 of the same file.

3. *Parameter specification*

One or more lines, each having two fields as follows:

name number

where *name* is a tunable parameter name, and *number* is the desired value (in decimal) for the given parameter. Only names that have been defined in part 4 of the **/etc/master** file can be used; *number* overrides the default value for the given parameter. The following is a list of the available parameters:

| | |
|---|---|
| **buffers** | Maximum number of external (mapped-out) buffers available to the kernel. If set to 0, *config* computes the optimum number for the system. |
| **sabufs** | Maximum number of internal (non-mapped) buffers available. |
| **hashbuf** | Maximum number of hash buffers. |
| **inodes** | Maximum number of inodes per file system. |
| **files** | Maximum number of files per file system. |
| **mounts** | Maximum number of mounted file systems. |
| **coremap** | Maximum number of core map elements. |
| **swapmap** | Maximum number of swap map elements. |
| **pages** | Number of memory pages. On segmented systems such as the 286, this value should be 0. |
| **calls** | Maximum number of entries in the system timeout table. |
| **procs** | Maximum number of processes per system. |
| **maxproc** | Maximum number of processes per user. |
| **texts** | Maximum number of text segments per system. |

| | |
|---|---|
| **clists** | Maximum number of clists per system. |
| **locks** | Maximum number of file locks per system. |
| **shdata** | Maximum number of shared data segments per system. |
| **timezone** | Number of minutes difference between the local timezone and Greenwich Mean Time. |
| **daylight** | Daylight savings time in effect (1), or not in effect (0). |
| **cmask** | Default file creation mask for process 0. |
| **maxprocmem** | Maximum amount of memory available per process. This value cannot be greater than 75% of total user memory. If set to 0, *config* computes the optimum value. |

**Example**

Suppose we wish to configure a system with the following devices:

one HD disk drive controller with 1 drive
one FD floppy disk drive controller with 1 driver

We must also specify the following parameter information:

root device is an HD (pseudo disk 3)
pipe device is an HD (pseudo disk 3)
swap device is an 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**

| | |
|---|---|
| /etc/master | default input master device table |
| c.c | default output configuration table file |

**See Also**

master(F)

**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.

## Name

cpp – The C language preprocessor.

## Syntax

**/lib/cpp** [ option ... ] [ ifile [ ofile ] ]

## Description

*Cpp* is the C language preprocessor which is invoked as the first pass of any C compilation using the *cc*(CP) command. Thus the output of *cpp* is designed to be in a form acceptable as input to the next pass of the C compiler. As the C language evolves, therefore, the use of *cpp* other than in this framework is not suggested. The preferred way to invoke *cpp* is through the *cc*(CP) command. See *m4*(CP) for a general macro processor.

*Cpp* optionally accepts two file names as arguments. *Ifile* and *ofile* are respectively the input and output for the preprocessor. They default to standard input and standard output if not supplied.

The following *options* to *cpp* are recognized:

**– P**
Preprocess the input without producing the line control information used by the next pass of the C compiler.

**– C**
By default, *cpp* strips C-style comments. If the **– C** option is specified, all comments (except those found on *cpp* directive lines) are passed along.

**– U*name***
Remove any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor.

**– D*name***
**– D*name*=*def***
Define *name* as if by a **#define** directive. If no *=def* is

given, *name* is defined as 1.

– I*dir*
Change the algorithm for searching for **#include** files whose names do not begin with / to look in *dir* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in "" will be searched for first in the directory of the *ifile* argument, then in directories named in – I options, and last in directories on a standard list. For **#include** files whose names are enclosed in <>, the directory of the *ifile* argument is not searched.

Two special names are understood by *cpp*. The name __LINE__ is defined as the current line number (as a decimal integer) as known by *cpp*, and __FILE__ is defined as the current file name (as a C string) as known by *cpp*. They can be used anywhere (including in macros) just as any other defined name.

All *cpp* directives start with lines begun by #. The directives are:

**#define** *name token-string*
Replace subsequent instances of *name* with *token-string*.

**#define** *name( arg, ..., arg ) token-string*
Notice that there can be no space between *name* and the (. Replace subsequent instances of *name* followed by a (, a list of comma separated tokens, and a ) by *token-string* where each occurrence of an *arg* in the *token-string* is replaced by the corresponding token in the comma separated list.

**#undef** *name*
Cause the definition of *name* (if any) to be forgotten from now on.

**#include** *"filename"*
**#include** *<filename>*
Include at this point the contents of *filename* (which will then be run through *cpp*). When the *<filename>* notation is used, *filename* is only searched for in the standard places. See the – I option above for more detail.

**#line** ***integer-constant "filename"***
Causes *cpp* to generate line control information for the next pass of the C compiler. ***Integer-constant*** is the line number of the next line and ***filename*** is the file where it comes from. If ***"filename"*** is not given, the current file name is unchanged.

**#endif**
Ends a section of lines begun by a test directive (**#if**, **#ifdef**, or **#ifndef**). Each test directive must have a matching **#endif**.

**#ifdef** ***name***
The lines following will appear in the output if and only if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

**#ifndef** ***name***
The lines following will not appear in the output if and only if *name* has been the subject of a previous **#define** without being the subject of an intervening **#undef**.

**#if defined** *identifier*
May be used in place of the **#if** directive. If the *identifier* is defined the directive has a value of 1, otherwise 0. This is frequently used for conditional environment-specific text.

**#elif** *constant-expression*
Allows for the conditional compilation of portions of the text. The *constant-expression* is evaluated and if it is not zero the text immediately following (until the next **elif, else, endif**) is passed to the compiler.

**#if** *constant-expression*
Lines following will appear in the output if and only if the *constant-expression* evaluates to non-zero. All binary non-assignment C operators, the **?:** operator, the unary −, !, and ~ operators are all legal in *constant-expression*. The precedence of the operators is the same as defined by the C language. There is also a unary operator **defined**, which can be used in *constant-expression* in these two forms: **defined** ( *name* ) or **defined** *name*. This allows the utility of **#ifdef** and **#ifndef** in a **#if** directive. Only these

operators, integer constants, and names which are known by *cpp* should be used in ***constant-expression***. In particular, the **sizeof** operator is not available.

**#else**
: Reverses the notion of the test directive which matches this directive. So if lines previous to this directive are ignored, the following lines will appear in the output. And vice versa.

The test directives and the possible **#else** directives can be nested.

## Files

| | |
|---|---|
| /usr/include | standard directory for **#include** files |

## See Also

cc(CP), m4(CP).

## Diagnostics

The error messages produced by *cpp* are intended to be self-explanatory. The line number and filename where the error occurred are printed along with the diagnostic.

## Notes

When newline characters were found in argument lists for macros to be expanded, previous versions of *cpp* put out the newlines as they were found and expanded. The current version of *cpp* replaces these newlines with blanks to alleviate problems that the previous versions had when this occurred.

## Name

cref – Makes a cross-reference listing.

## Syntax

**cref [ – acilnostux123 ]** files

## 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. Filename
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 *FILES* below). Assembler predefined symbols or 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, 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 (see above) |
| l | Puts line number in column 3 (instead of current symbol) |

| | |
|---|---|
| **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 only symbols that occur exactly once |
| **x** | Prints only 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(CP), cc(CP), sort(C), xref(CP)

## 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.

## Name

ctags − Creates a tags file.

## Syntax

**ctags** [ **−u** ] [ **−w** ] [ **−x** ] name ...

## Description

*Ctags* makes a tags file for *vi*(C) from the 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, as well as the text of that line and prints this on the standard output. With the **−x** option no tags file is created. This is a simple index which can be printed out as an off−line readable function index.

Files whose name ends 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 which 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, with a trailing .c removed, if any, and leading pathname components also removed. This makes use of *ctags* practical in directories with more than one program.

## Files

| | |
|---|---|
| tags | Output tags file |

**See Also**

ex(C), vi(C)

**Credit**

This utility was developed at the University of California at Berkeley and is used with permission.

## Name

delta - Makes a delta (change) to an SCCS file.

## Syntax

delta [- rSID] [- s] [- n] [- glist] [- m[mrlist]] [- y[comment]] [- p] files

## Description

*Delta* is used to permanently introduce into the named SCCS file changes that were made to the file retrieved by *get*(CP) (called the *g-file*, or generated file).

*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 nonSCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored. If a name of - is given, the standard input is read (see Warning); 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 flags (see *admin*(CP)) 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 *get*(CP)). A diagnostic results if the specified SID is ambiguous, or if it is necessary and omitted on the command line. |
| - s | Suppresses the issue, on the standard output, of the created delta's SID, as well as the number of lines inserted, deleted and unchanged in the SCCS file. |
| - n | Specifies retention of the edited *g-file* (normally removed at completion of delta processing). |

– **g*list*** Specifies a ***list*** (see *get*(CP) for the definition of ***list***) of deltas which are to be *ignored* when the file is accessed at the change level (SID) created by this delta.

– m[*mrlist*] If the SCCS file has the v flag set (see *admin*(CP)) 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 – y keyletter).

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*(CP)), it is taken to be the name of a program (or shell procedure) which will validate the correctness of the MR numbers. If a nonzero exit status is returned from MR number validation program, *delta* terminates (it is assumed 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*(C) format.

## Files

All files of the form *?*-file are explained in Chapter 5, "SCCS: A Source Code Control System" in the *XENIX Programmer's Guide*. The naming convention for these files is also described there.

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*(F)) 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(CP), bdiff(C), get(CP), help(CP), prs(CP), sccsfile(F)

## Diagnostics

Use *help*(CP) for explanations.

**Name**

dosld − XENIX to MS−DOS cross linker

**Syntax**

**dosld** [*options* ] file ...

**Description**

*Dosld* links the object files(s) given by *file* to create a program for execution under MS−DOS. Although similar to *ld*(CP), *dosld* has many options that differ significantly from *ld*. The options are described below:

**−D** DS Allocate. This instructs *dosld* to perform DS alloca− tion. It is generally used inconjunction with the **−H** option.

**−H** Load high. This option instructs *dosld* to set a field in the header of the executable file to tell MS−DOS to load the program at the highest available position in memory. It is most often used with programs in which data precedes code in the memory image.

**−L** Include line numbers. This option instructs *dosld* to include line numbers in the listing file (if any). Note that *dosld* cannot put line numbers in the listing file if the source translator hasn't put them in the object file.

**−M** Include public symbols. This option instructs *dosld* to include public symbols in the list file. The symbols are sorted twice, lexicographically and by address.

**−C** Ignore case. This option instructs *dosld* to treat upper and lower case characters in symbol names as identical.

**−F** *num*

Set stack size. This option should be followed by a hexa− decimal number. *Dosld* will use this number for the size in bytes of the stack segment in the output file.

**−S** *num*

Set segment limit. This option should be followed by a decimal nbumber between 1 and 1024. The number sets the limit on the number of different segments that may be linked together. The default is 128. Note that the higher the value given, the slower the link will be.

**−m** *filename*

Create map file. This option should be followed by a filename. *Dosld* will create a file with the given name in which it will put information about the segments and goups in the executable. Additionally, public symbols and line numbers will be listed in this file if the **−M** and **−L** options are given.

**−nl** *num*

Set name length. This optinb should be followed by a decimal number. The option instructs *dosld* to truncate all public and external symbols longer than *num* characters.

**−o** *filename*
: Name output file. This option should be followed by a filename which *dosld* will use as the name of the execut− able file it creates. The default name is **a.out**.

**−u** *name*
: Name undefined symbol. This option should be followed by a symbol name. *Dosld* will enter the given name into its symbol table as an undefined symbol. The **−u** option may appear more than once on the command line.

**−G** Ignore group associations. This option instructs *dosld* to ignore any group definitions it may find in the input files. This option is provided for compatibility with old versions of MS−LINK; generally, it should never be used.

As with *ld*, the files passes to *dosld* may be either XENIX−style libraries (objects collected using *ar*(CP) and indexed using *ranlib*(CP)) or ordinary 8086 object files. Unless the **−u** option appears, at least one of the files passed to *dosld* must be an ordinary object file. Libraries are searched only after all the ordinary object files have been processed.

**Files**

/usr/bin/dosld

**See Also**

ar(CP), as(CP), cc(CP), ld(CP), ranlib(CP)

## Name

get - Gets a version of an SCCS file.

## Syntax

**get** [- rSID] [- ccutoff] [- ilist] [- xlist] [- aseq-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 options, 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 nonSCCS files (last component of the pathname 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, nonSCCS files and unreadable files are silently ignored.

The generated text is normally written into a file called the *g-file* whose name is derived from the SCCS filename by simply removing the leading s.; (see also *FILES*, below).

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 *S*CCS *ID*entification 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 2 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*(CP). The - e option used in a *get* for a particular version (SID) of the SCCS file prevents further *gets* for editing on the same SID until *delta* is executed or the **j** (joint edit) flag is set in the SCCS file (see *admin*(CP)). 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 reexecuting 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*(CP)) 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*(CP)) 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 non-leaf *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 Chapter 5, "SCCS: A Source Code Control System," in the XENIX *Programmer's Guide*.

**– 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 (see below) 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 *FILES* 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, followed by a horizontal tab, followed by the text line.

**– n** Causes each generated text line to be preceded with the %M% identification keyword value (see below). The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the **– m** and **– n** options are used, the format is: %M% value, followed by a horizontal tab, followed by 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 (e.g., **– r1**), or release and level (e.g., **– r1.2**).

**– a***seq-no.* The delta sequence number of the SCCS file delta (version) to be retrieved (see *sccsfile*(F)). This option is used by the *comb*(CP) command; it is not particularly useful 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 filename 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*(CP)), 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*(CP)). |
| %F% | SCCS filename. |
| %P% | Fully qualified SCCS filename. |
| %Q% | The value of the q flag in the file (see *admin*(CP)). |
| %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*(C). |
| %W% | A shorthand notation for constructing *what*(C) strings for XENIX program files. %W% = %Z%%M%<horizontal-tab>%I% |
| %A% | Another shorthand notation for constructing *what*(C) strings for nonXENIX 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 filename is formed from the SCCS filename: the last component of all SCCS filenames 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: the *g-file* is named by removing the s. prefix. For example, s.xyz.c, the auxiliary filenames 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, the *g-file*'s mode is 644; otherwise the mode is 444. 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 444 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. A blank character if the delta was applied;
   * otherwise
b. A blank character if the delta was applied or wasn't applied and ignored;
   * if the delta wasn't applied and wasn't ignored
c. 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)
d. Blank
e. SCCS identification (SID)
f. Tab character
g. Date and time (in the form YY/MM/DD HH:MM:SS) of creation
h. Blank
i. 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*(CP)) 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 644 and it is owned by the effective user. The format of the *p-file* is: the gotten SID, followed by a blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time the *get* was executed, followed by a blank and the – i option if it was present, followed by a blank and the – x option if it was present, followed by 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 (i.e., *get*) that created it. The *z-file* is created in the directory containing the SCCS file for the duration of *get*. The same protection restrictions as those for the *p-file* apply for the *z-file*. The *z-file* is created mode 444.

## See Also

admin(CP), delta(CP), help(CP), prs(CP), what(C), sccsfile(F)

## Diagnostics

Use *help*(CP) for explanations.

## Notes

If the effective user has write permission (either explicitly or implicitly) 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.

## Name

gets – Gets a string from the standard input.

## Syntax

gets [ string ]

## Description

*Gets* can be used with *csh*(CP) to read a string from the standard input. If *string* is given it is used as a default value if an error occurs. The resulting string (either *string* or as read from the standard input) is written to the standard output. If no *string* is given and an error occurs, *gets* exits with exit status 1.

## See Also

line(C), csh(CP)

## Name

hdr – Displays selected parts of object files.

## Syntax

**hdr** [ – dhprsSt ] 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 which represents the symbol's type as in *nm*(CP), except 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. The last field will indicate, if present, 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 and their meanings are:

**– 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 **x.out** segmented executable files.)

## See Also

a.out(F), nm(CP)

## Name

help – Asks for help about SCCS commands.

## Syntax

**help** [ args]

## Description

*Help* finds information to explain a message from an SCCS command or 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. There are the following types of arguments:

| | |
|---|---|
| type 1 | Begins with nonnumerics, ends in numerics. The non-numeric prefix is usually an abbreviation for the program or set of routines which produced the message (e.g., **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 (e.g., **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 |

**Name**

ld – Invokes the link editor.

**Syntax**

**ld** [ *options* ] *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 to 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 library) extensions. If more than one name is 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 of 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*(CP) format, that is, the first member must be named __.SYMDEF, which is a dictionary for the library. *Ld* ignores the modification dates of the library and the __.SYMDEF entry, so if object files have been added to the library since __.SYMDEF was created, the link may result in an "invalid object module."

The library is searched iteratively to satisfy as many references as possible and 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 library files. *Ld* sets the entry point of the resulting program to the beginning of the first routine.

There are the following options:

**–A*****num***
  Creates a standalone program whose expected load address

(in hexadecimal) is *num*. This option sets the absolute flag in the header of the a.out file. Such program files can only be executed as standalone programs.

**– B*num***
Sets the text selector bias to the specified hexadecimal number.

**– C**
Causes the editor to ignore the case of symbols.

**– D*num***
Sets the data selector bias to the specified hexadecimal number.

**– F*num***
Sets the size of the program stack to *num* bytes. Default stack size if not given, is 4086 bytes.

**– 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 shared by all users executing the file.

**– m name**
Creates a link map file named *name* that includes public symbols.

**– Ms**
Creates small model program and checks for error, 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 program 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** .

**– nl***num*
Truncates symbols to the length specified by *num.*

**– o** *name*
Sets the executable program filename to *name* instead of **a.out.**

**– s** Strips the symbol table.

**– S***num*
Sets the maximum number of data segments to *num.* If no argument is given, the default is 128.

**– u***symbol*
Designates the specified *symbol* as undefined.

**– v***num*
specifies the Xenix version number. Acceptable values for *num* are 2 or 3; 3 is the default.

*Ld* should be invoked using the *cc*( CP) 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 command line arguments in the correct order can result in errors.

## Files

/bin/ld

## See Also

as( CP), ar( CP), cc( CP), ranlib( CP)

## Notes

The **– A***num,* **– B***num,* and **– D***num* options to *ld* should not be used when creating a binary for an 8086/88 or 80186/88 system.

The user must make sure that the most recent library versions have been processed with *ranlib*(CP) before linking. If this is not done, *ld* cannot create executable programs using these libraries.

## Name

lex – Generates programs for lexical analysis.

## Syntax

lex [– ctvn] [ file ] ...

## Description

*Lex* generates programs to be used in simple lexical analysis of text.

The input *files* (standard input default) contain strings and expressions to be searched for, and C text to be executed when strings are found.

A file lex.yy.c is generated which, when loaded with the 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 actual string matched is left in *yytext*, an external character array. Matching is done in order of the strings in the file. The 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 occurrences of, the previous character or character class. 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 or preceded by \. Thus [a– zA– Z]+ matches a string of letters.

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.

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 preceding %% which 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 {}. Note that curly brackets do not imply parentheses; only string substitution is done.

## 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() != '*');
                switch (input())
                        {
                        case '/': break;
                        case '*': unput('*');
                        default: go to loop;
                        }
                }
```

The external names generated by *lex* all begin with the prefix yy or YY.

The options must appear before any files. The option – 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 statistics of the machine generated, – n will not print out the – summary. Multiple files are treated as a single file. If no files are specified, standard input is used.

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.

## See Also

yacc(CP)
Xenix *Software Development Guide*

## Name

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, nonportable, or wasteful. It also checks type usage more strictly than the C compiler. Among the things which are 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 which 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 the 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, in any order. The following options are used to 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 used and not defined, or 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*(CP) 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 filename will be printed followed by a question mark.

## Files

| | |
|---|---|
| /usr/lib/lint[12] | Program files |

/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
    Standard lint libraries (source format)

/usr/tmp/*lint* Temporaries

## See Also

cc(CP)

## Notes

*Exit*(S), and other functions which do not return, are not understood. This can cause improper error messages.

## Name

lorder – Finds ordering relation for an object library.

## Syntax

lorder file ...

## Description

*Lorder* creates an ordered listing of object filenames, showing which files depend on variables declared in other files. The *file* is one or more object or library archive files (see *ar*(CP)). The standard output is a list of pairs of object filenames. The first file of the pair refers to external identifiers defined in the second. The output may be processed by *tsort*(CP) to find an ordering of a library suitable for one-pass access by *ld*(CP).

## Example

The following command builds a new library from existing .o files:

```
ar cr library `lorder *.o |tsort`
```

## Files

| | |
|---|---|
| *symref, *symdef | Temp files |

## See Also

ar(CP), ld(CP), tsort(CP)

## Notes

Object files whose names do not end with .o, even when contained in library archives, are overlooked. Their global symbols and references are attributed to some other file.

## Name

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 filename 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 filenames 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. Potential macro names consist of alphabetic letters, digits, and underscore _, where the first character is not a digit.

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 which 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 **$*n*** in the replacement text, where *n* is a digit, is replaced by the *n*-th argument. Argument **0** is the name of the macro; missing arguments are replaced by the null string; **$#** 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 one 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 (i.e., ` '). |
| 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 0-9. 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. 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 if it is not present, null. |
| 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; 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*. |

## Name

make – Maintains, updates, and regenerates groups of programs.

## Syntax

make [– f makefile] [– p] [– i] [– k] [– s] [– r] [– n] [– b] [– e] [– t] [– q] [– d] [names]

## Description

The following is a brief description of all options and some special names:

| | |
|---|---|
| – f *makefile* | Description filename. *Makefile* is assumed to be the name of a description file. A filename 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) rather than issues the usual commands. |
| – d | Debug mode. Prints out detailed information on files and times examined. |

| | |
|---|---|
| – 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. |
| **.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 are hit. |
| **.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 :, 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. (#) 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 (see discussion of the **MAKEFLAGS** macro under *Environment*). 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 explicitly that 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 "super-makes". 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 is shown under *Libraries*.

*Internal Macros*

There are five internally maintained macros which are useful for writing rules for building targets:

**$*** The macro **$*** stands for the filename 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 only evaluated for inference rules or the .DEFAULT rule. It is the module which is out of date with respect to the target (i.e., the "manufactured" dependent filename). Thus, in the .c.o rule, the **$<** macro would evaluate to the .c file. An example for making optimized .o files from .c files is:

```
.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 which must be rebuilt.

**$%** The **$%** macro is only evaluated 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 upper case 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 ./ 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, .s, etc. 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 which 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 the *make* on any machine in a form suitable for recompilation, the following command is used:

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

The only peculiarity in this output is the (null) string which *printf*(S) prints when handed a null string.

A tilde in the above rules refers to an SCCS file (see *sccsfile*(F)). Thus, the rule .c~.o would transform an SCCS C source file into an 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 (i.e. .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 (e.g., 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*(CP), *lex*(CP), and *yacc*(CP) 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. Here, we assume 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 listed above 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 filenames (inside lib) whose

C source files are out of date. The substitution mode translates the .o to .c. (Unfortunately, one cannot as yet transform 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. This type of construct 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(C)

## Notes

Some commands return nonzero status inappropriately; use -i to overcome the difficulty. Commands that are directly executed by the shell, notably *cd*(C), 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.

## Name

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 make programs with large numbers of error diagnostics much smaller, and reduce system overhead in running the program as the 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 placed at the end of the specified message file for recompiling part of a large *mkstr*ed program.

A typical *mkstr* command line is

```
mkstr pistrings xx *.c
```

This command causes 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 followed by a null character and a newline character; the null character terminates the message so it can be easily used when retrieved, 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 a *lseek* pointer into the file which can be used to retrieve the message. For example, 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* which opens the message file, reads the string, and prints it out. The following example 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

lseek(S), xstr(CP)

## 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.

## Name

nm − Prints name list.

## Syntax

**nm** [ **−acgnoOprsuv** ] [ +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), **S** (segment name), **C** (common symbol), or **K** (8086 common segment). If the symbol table is in segmented format, symbol values are displayed as **segment:offset**. If the symbol is local (non−external) 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 which 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** | Switch the display format. If the symbol table is in segmented format, print values in non−segmented format. If not segmented, print values in segmented format. |
| **−u** | Print only undefined symbols. |
| **−v** | Also describe the object file and symbol table format. |

**Files**

| | |
|---|---|
| a.out | Default input file |

**See Also**

ar(CP), ar(F), a.out(F)

## Name

prof – Displays profile data.

## Syntax

**prof** [ – **a** ] [ – **l** ] [ 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 are reported rather than just external symbols. If the – **l** option is used, the output is listed by symbol value rather than 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

monitor(S), profil(S), cc(CP)

## Notes

Beware of quantization errors.

If you use an explicit call to ***monitor***(S) you will need to make sure that the buffer size is equal to or smaller than the program size.

## Name

prs – Prints an SCCS file.

## Syntax

**prs** [– d[dataspec]] [– r[SID]] [– e] [– l] [– a] files

## Description

*Prs* prints, on the standard output, all or part of an SCCS file (see *sccsfile*(F)) 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 nonSCCS files (last component of the pathname 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; nonSCCS files and unreadable files are silently ignored.

Arguments to *prs*, which may appear in any order, consist of options, and filenames.

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*) interspersed with optional user-supplied text. |
| – r[*SID*] | Used to specify the *S*CCS *ID*entification (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, i.e., delta type = *R*, (see *rmdel*(CP)) and existing, i.e., delta type = *D*, deltas. If the – a option is not specified, information for existing deltas only is provided. |

## Data Keywords

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file (see *sccsfile*(F)) 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.

TABLE 1. SCCS Files Data Keywords

| *Keyword* | *Data Item* | *File Section* | *Value* | *Format* |
|---|---|---|---|---|
| :Dt: | Delta information | Delta Table | See below* | S |
| :DL: | Delta line statistics | " | :Li:/:Ld:/:Lu: | S |
| :Li: | Lines inserted by Delta | " | nnnnn | S |
| :Ld: | Lines deleted by Delta | " | nnnnn | S |
| :Lu: | Lines unchanged by Delta | " | nnnnn | S |
| :DT: | Delta type | " | *D* or *R* | S |
| :I: | SCCS ID string (SID) | " | :R:.:L:.:B:.:S: | S |
| :R: | Release number | " | nnnn | S |
| :L: | Level number | " | nnnn | S |
| :B: | Branch number | " | nnnn | S |
| :S: | Sequence number | " | nnnn | S |
| :D: | Date Delta created | " | :Dy:/:Dm:/:Dd: | S |
| :Dy: | Year Delta created | " | nn | S |
| :Dm: | Month Delta created | " | nn | S |
| :Dd: | Day Delta created | " | nn | S |
| :T: | Time Delta created | " | :Th:::Tm:::Ts: | S |
| :Th: | Hour Delta created | " | nn | S |
| :Tm: | Minutes Delta created | " | nn | S |
| :Ts: | Seconds Delta created | " | nn | S |
| :P: | Programmer who created Delta | " | logname | S |
| :DS: | Delta sequence number | " | nnnn | S |
| :DP: | Predecessor Delta seq-no. | " | nnnn | S |
| :DI: | Seq-no. of deltas incl., excl., ignored | " | :Dn:/:Dx:/:Dg: | S |
| :Dn: | Deltas included (seq #) | " | :DS: :DS:... | S |
| :Dx: | Deltas excluded (seq #) | " | :DS: :DS:... | S |
| :Dg: | Deltas ignored (seq #) | " | :DS: :DS:... | S |
| :MR: | MR numbers for delta | " | text | M |
| :C: | Comments for delta | " | text | M |
| :UN: | User names | User Names | text | M |
| :FL: | Flag list | Flags | text | M |
| :Y: | Module type flag | " | text | S |
| :MF: | MR validation flag | " | *yes* or *no* | S |
| :MP: | MR validation pgm name | " | text | S |
| :KF: | Keyword error/warning flag | " | *yes* or *no* | S |
| :BF: | Branch flag | " | *yes* or *no* | S |
| :J: | Joint edit flag | " | *yes* or *no* | S |
| :LK: | Locked releases | " | :R:... | S |
| :Q: | User defined keyword | " | text | S |
| :M: | Module name | " | text | S |
| :FB: | Floor boundary | " | :R: | S |
| :CB: | Ceiling boundary | " | :R: | S |
| :Ds: | Default SID | " | :I: | S |
| :ND: | Null delta flag | " | *yes* or *no* | S |
| :FD: | File descriptive text | Comments | text | M |
| :BD: | Body | Body | text | M |
| :GB: | Gotten body | " | text | M |
| :W: | A form of *what*(C) string | N/A | :Z::M:\t:I: | S |
| :A: | A form of *what*(C) string | N/A | :Z::Y: :M: :I::Z: | S |
| :Z: | *what*(C) string delimiter | N/A | @(#) | S |
| :F: | SCCS filename | N/A | text | S |
| :PN: | SCCS file pathname | N/A | text | S |

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

## Examples

The following:

```
prs – d"Users and/or user IDs for :F: are:\n:UN:" s.file
```

may produce on the standard output:

```
Users and/or user IDs for s.file are:
xyz
131
abc
```

```
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 allowed to be used with the *special case* is the – a option.

## Files

/tmp/pr?????

## See Also

admin(CP), delta(CP), get(CP), help(CP), sccsfile(F)

## Diagnostics

Use *help*(CP) for explanations.

## Name

ranlib – Converts archives to random libraries.

## Syntax

ranlib archive

## Description

*Ranlib* converts each *archive* to a form which can be loaded more rapidly by the loader, by adding a table of contents named __.SYMDEF to the beginning of the archive. It uses *ar*(CP) to reconstruct the archive, so sufficient temporary file space must be available in the file system containing the current directory.

## See Also

ld(CP), ar(CP), copy(C), settime(C)

## Notes

Failure to process a library with *ranlib*, or failure to reprocess a library with *ranlib*, will cause *ld* to fail. Because generation of a library 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 get the warning even if you only copy the library.

## Name

ratfor – Converts Rational FORTRAN into standard FORTRAN.

## Syntax

**ratfor** [ option ... ] [ filename ... ]

## Description

*Ratfor* converts a rational dialect of FORTRAN into ordinary irrational 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.

## Name

regcmp - Compiles regular expressions.

## Syntax

regcmp [- ] files

## Description

*Regcmp*, in most cases, precludes the need for calling *regcmp* (see *regex*(S)) 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 which 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(S)

## Name

rmdel – Removes a delta from an SCCS file.

## Syntax

rmdel – rSID files

## 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* exists for the named SCCS file, the SID specified must ***not*** appear in any entry of the *p-file*(see *get*(CP)).

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 the pathname 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*(CP) |
| z-file | See *delta*(CP) |

## See Also

delta(CP), get(CP), help(CP), prs(CP), sccsfile(F)

## Diagnostics

Use *help*(CP) for explanations.

## Name

sact – Prints current SCCS file editing activity.

## Syntax

sact files

## Description

*Sact* informs the user of any impending deltas to a named SCCS file. This situation occurs when *get*(CP) with the – e option has been previously executed without a subsequent execution of *delta*(CP). 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 nonSCCS 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 i.e., 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(CP), get(CP), unget(CP)

## Diagnostics

Use *help*(CP) for explanations.

## Name

sccsdiff – Compares two versions of an SCCS file.

## Syntax

sccsdiff – rSID1 – rSID2 [– p] [– sn] files

## 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*(C) in the order given. |
| – p | Pipe output for each file through *pr*(C). |
| – s*n* | *n* is the file segment size that *bdiff* will pass to *diff*(C). This is useful when *diff* fails due to a high system load. |

## Files

/tmp/get????? Temporary files

## See Also

bdiff(C), get(CP), help(CP), pr(C)

## Diagnostics

file: *No differences* If the two versions are the same.

Use *help*(CP) for explanations.

## Name

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 decimal and hexadecimal, of each object-file argument. If no file is specified, a.out is used.

## See Also

a.out(F)

## Name

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

$$y_0'' = ky_1' \; , \; . \; . \; . \; , \; y_n'' = ky_{n-1}'$$

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.

## Name

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 things.

## See Also

hd(C), od(C)

## Credit

This utility was developed at the University of California at Berkeley and is used with permission.

## Name

strip – Removes symbols and relocation bits.

## Syntax

strip name ...

## Description

*Strip* removes the symbol table and relocation bits ordinarily attached to the output of the assembler and link editor. This is useful for saving space after a program has been debugged.

The effect of *strip* is the same as use of the – s option of *ld*.

If *name* is an archive file, *strip* will remove the local symbols from any *a.out* format files it finds in the archive. Certain libraries, such as those residing in /lib, have no need for local symbols. By deleting them, the size of the archive is decreased and link editing performance is increased.

## Files

| | |
|---|---|
| /tmp/stm* | Temporary file |

## See Also

ld(CP)

## Name

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, the time spent in the system, and the time spent in execution of the command. Times are reported in seconds.

The times are printed on the standard error.

## See Also

times(S)

## Name

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(CP)

## Diagnostics

*Odd data:* There is an odd number of fields in the input file.

## Notes

The *sort* algorithm is quadratic, which can be slow if you have a large input list.

## Name

unget – Undoes a previous get of an SCCS file.

## Syntax

unget [– rSID] [– s] [– n] files

## 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 nonSCCS 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 which would normally be removed from the current directory. |

## See Also

delta(CP), get(CP), sact(CP)

## Diagnostics

Use *help*(CP) for explanations.

## Name

val – Validates an SCCS file.

## Syntax

val –

val [– s] [– rSID] [– mname] [– ytype] files

## 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 and also returns a single 8-bit code upon exit as described below.

The options are defined as follows. The effects of any option 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* (*S*CCS *ID*entification String) is an SCCS delta number. A check is made to determine if the *SID* is ambiguous (e. g., r1 is ambiguous because it physically does not exist but implies 1.1, 1.2, etc. which may exist) or invalid (e. g., r1.0 or r1.1.0 are 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, i. e., 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 option

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(CP), delta(CP), get(CP), prs(CP)

## Diagnostics

Use *help*(CP) for explanations.

## Notes

*Val* can process up to 50 files on a single command line.

## Name

xref – Cross-references C programs.

## Syntax

xref [ file ... ]

## Description

*Xref* reads the named *files* or the standard input if no file is specified and prints a cross reference consisting of lines of the form

```
        identifier        filename        line numbers ...
```

Function definition is indicated by a plus sign (+) preceding the line number.

## See Also

cref(CP)

## Name

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, most useful if they are also read-only.

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 which are suffices 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 which contains strings which 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( CP)

## 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 when just placing the longer one there will do.

## Name

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 which 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*(CP) 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(CP)

## 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 filenames are fixed, at most one *yacc* process can be active in a given directory at a time.

# Index

## *Programming Commands* (CP)