Forward

TECHNOLOGY INCORPORATED

XENIX™ SYSTEM

SOFTWARE DEVELOPMENT

VOLUME 2

Catalog no. 9100
Part no. 91F00B

Document no. 8603b-100-00

CONTENTS

CHAPTER 1

INTRODUCTION

One of the primary uses of the XENIX system is as an environment for software development. This manual describes the tools available in this environment and the low level environment itself. Some knowledge of the XENIX system and of the C programming language is presumed.

Nearly all of the XENIX system is written in the C programming language. Therefore, C is the ideal language for creating new XENIX applications. For more information on programming in C, you should refer to Volume I, Programmer's Introduction, for information on the concepts and software that underly the XENIX system, and to Kernighan and Ritchie's book, The C Programming Language, for an excellent tutorial and reference on the language itself. The C Reference portion of this book is contained in Appendix A of this manual.

The tools used to create executable C programs are:

cc	The C compiler.
lint	A C program checker.
ld	The XENIX loader.
as	The XENIX assembler.

Note that **cc** automatically invokes both the loader and the assembler so that use of either is optional. **Lint** is normally used in the early stages of program development to check for illegal and improper usage of the C language.

In addition to the above tools, the program **make** is used to automatically maintain and regenerate the software in medium scale programming projects.

The above tools are used to create executable C programs. These programs are created to run in the XENIX environment. This environment is manifested in the various system calls and libraries that are available to the programmer.

It is worth noting that not all programming projects are best implemented in C, even if they are programs written for XENIX. Often, simple programs can be written in the shell command language much more quickly than they can be in C. For some complicated programs, **lex** and **yacc** may be just what is required. **Lex** is a lexical analyzer that can be used to accept a given input language. **Yacc** is a program designed to compile grammars into a parsing program. Typically, it is used to compile languages that are recognized by **lex**. For this reason, **lex** and **yacc** are often used together, although either can be used separately.

With the above overview of software development in mind, this manual is organized as follows:

CHAPTER 1: INTRODUCTION
The chapter you are now reading contains a word about the develpment of software on the XENIX system with emphasis on how the the software tools discussed in this manual fit together.

CHAPTER 2: BASIC SOFTWARE
This chapter describes each of the basic tools that you are likely to use either directly or indirectly, in creating C programs on the XENIX system.

CHAPTER 3: ENVIRONMENT
This chapter discusses the standard XENIX environment and how this environment can be accessed either from C or from assembly language.

CHAPTER 4: OTHER TOOLS
This chapter describes tools and languages that are useful for special purposes, but that are not as generally useful as the software discussed in chapter 2.

CHAPTER 5: REFERENCE
This chapter contains important information on commands, system calls, subroutines, special files, and file formats. This information is indispensible to the serious programmer.

CHAPTER 2

BASIC DEVELOPMENT TOOLS

This chapter discusses five basic development tools: cc, lint, make, adb, and as. Together, these tools make up a solid software development package, premitting you to create, execute, debug, and maintain software. Each of these tools is discussed in turn in the following sections.

2.1 CC: The C Compiler

Cc is the command used to invoke the XENIX C compiler. Since the entire XENIX system is written in the C language, cc is the fundamental XENIX program development tool. The C language supported by the C compiler is described in _Appendix A, The C Reference Manual_. For more information on programming in C, see _The C Language_, by Kernighan and Ritchie.

This section discusses the compiler used to create executable files from programs written in the C language. The emphasis here is on giving insight into cc's operation and use. Special emphasis is given to input and output files and and to the available compiler options. Throughout, familiarity with compilers and with the C language is assumed.

The fundamental function of the C compiler is to produce executable programs by processing C source files. The word ``processing'' is the key here, since the compilation process involves several distinct phases: These phases are described below:

Preprocessing
: In this phase of compilation, your C source program is examined for macro definitions and include file directives. Any include files are processed at the point of the include statement; then occurrences of macros are expanded throughout the text. Normally, standard include files found in the _/usr/include_ directory are included at the beginning of C programs. These standard include files normally are named with a `.h' extension. For example, the following statement includes the definitions for functions in the standard I/O library:

```
#include <stdio.h>
```

Note that the angle brackets indicate that the file is presumed to exist in _/usr/include_. The effects of preprocessing on a file can be captured in a file by specifying the -P switch on the cc command line. The useful for debugging, when you suspect that an include file or macro is not expanding as desired.

Optimization
: Optimization of generated code can be specified on the cc command line with the -O switch. This

option should be used to increase execution speed or to decrease size of the executing program. Since programs will take longer to compile with this option, you may want to use this option only after you have a working debugged program.

Generation of Assembly Code

The C compiler generates assembly code that is later assembled by the XENIX assembler, **as**. Cc's assembly output can be saved in a file by specifying the -S switch when the compiler is invoked. Assembly output is saved in a file whose name has the `.s' extension.

Assembly

To assemble the generated assembly code, **cc** calls **as** to create a `.o' file. The `.o' file is used in the next step, linking and loading.

Linking and Loading

The final phase in the compilation of a C program is linking and loading. In this phase, your newly created `.o' file is loaded into memory along with any needed `.o' library modules. These modules are then linked together to create a final executable module, whose name by default is a.out. The program responsible for all this is the XENIX loader, **ld**. Loader options can be specified on the **cc** command line. These options are discussed later in the section on the loader.

It is important to realize that all of the above phases can be controlled at the **cc** command level: each does not have to be invoked separately. What normally happens when you execute a **cc** command is that a sequence of programs processes the original C source file. Each program creates a temporary file that is used by the next program in the sequence. The final output is the load image that is loaded into memory when the final executable file is run.

2.1.1 Invocation Switches

A list of switches follows:

-c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

-p Arrange for the compiler to produce code which counts the number of times each routine is called. Also, if loading takes place, replace the standard startup routine by one that automatically calls monitor(3) at the start and arranges to write out a mon.out file at normal termination of execution of the object program. An execution profile can then be generated by use of prof(1).

-O Invoke an object-code optimizer.

-S Compile the named C programs, and leave the assembler-language output on corresponding files suffixed `.s'.

-P Run only the macro preprocessor and place the result for each `.c' file in a corresponding `.i' file. The resultant file has no `#' lines in it.

-o output
Give the final output file the name specified by output. If this option is used the file a.out will be left undisturbed.

-D name=def
Define the name to the preprocessor, as if by `#define'. If no definition is given, the name is defined as 1.

-U name
Remove any initial definition of name.

-I dir
`#include' files whose names do not begin with `/' are always sought first in the directory of the file argument, then in directories named in -I options, then in directories on a standard list.

Other arguments are taken to be either loader option arguments, or C-compatible object programs, typically produced by an earlier cc run, or perhaps libraries of C-compatible routines created with the assembler. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with the name a.out.

Note that some versions of the C compiler support additional switches. These switches and their function are described in the reference section of this manual.

2.1.2 The Loader

As mentioned in the above sections, the XENIX loader, **ld**, plays a fundamental part in the development of any C program. For this reason it is discussed as part of **cc**; it can however, be used as a stand-alone processor of object files. Note that arguments to ld can be given on the cc command line and are a part of the syntax of the **cc** command.

The available loader switches are listed below. except for -l, they should appear before filename arguments.

-s	\`Strip' the output, that is, remove the symbol table and relocation bits to save space (but impair the usefulness of the debugger). This information can also be removed by *strip*(1).
-u	Take the following argument as a symbol and enter it as undefined in the symbol table. This is useful for loading wholly from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.
-l*x*	This option is an abbreviation for the library name /lib/lib*x*.a, where *x* is a string. If that does not exist, *ld* tries /usr/lib/lib*x*.a. A library is searched when its name is encountered, so the placement of a -l is significant.
-x	Do not preserve local (non-.globl) symbols in the output symbol table: enter only external symbols. This option saves some space in the output file.
-X	Save local symbols except for those whose names begin with \`L'. This option is used by *cc* to discard internally generated labels while retaining symbols local to routines.
-n	Arrange that when the output file is executed, the text portion will be read-only and shared among all users executing the file. This involves moving the data areas up to the first possible 4K word boundary following the end of the text.
-i	When the output file is executed, the program text and data areas will live in separate address spaces. The only difference between this option and -n is that here the data starts at location 0.

-o	The name argument after **-o** is used as the name of the ld output file, instead of **a.out**.

For more information on the loader, see **ld** in the reference section of this manual.

2.1.3 Files

The files making up the compiler, as well as those files needed, used, or created by **cc** are listed below:

file.c	input file
file.o	object file
a.out	loaded output
/tmp/ctm?	temporaries for cc
/lib/cpp	preprocessor
/lib/c[01]	compiler for cc
/lib/c2	optional optimizer
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	startoff for profiling
/lib/libc.a	standard library
/usr/include	standard directory for `#include' files
/bin/23fix	processor for large-text programs

2.2 LINT: A C Program Checker

Lint is a program that examines C source programs, detecting a number of bugs and obscurities. It enforces the type rules of C more strictly than the C compilers. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful, or error prone, constructions which nevertheless are, strictly speaking, legal.

The separation of function between lint and the C compilers has both historical and practical rationale. The compilers turn C programs into executable files rapidly and efficiently. This is possible in part because the compilers do not perform sophisticated type checking, especially between separately compiled programs. Lint takes a more global, leisurely view of the program, looking much more carefully at the compatibilities.

This section discusses the use of lint, gives an overview of the implementation, and gives some hints on the writing of machine independent C code.

Suppose there are two Cl source files, filel.c and file2.c, which are ordinarily compiled and loaded together. Then the command

```
lint  filel.c  file2.c
```

produces messages describing inconsistencies and inefficiencies in the programs. The program enforces the typing rules of C more strictly than the C compilers (for both historical and practical reasons) enforce them. The command

```
lint  -p  filel.c  file2.c
```

produces, in addition to the above messages, additional messages that relate to the portability of the programs to other operating systems and machines. Replacing the **-p** by **-h** produces messages about various error-prone or wasteful constructions that, strictly speaking, are not bugs. Saying **-hp** gets the whole works.

The next several sections describe the major messages; the discussion of lint closes with sections discussing the implementation and giving suggestions for writing portable C. The final section gives a summary of lint options.

2.2.1 A Word About Philosophy

Many of the facts that lint needs may be impossible to discover. For example, whether a given function in a program ever gets called may depend on the input data. Deciding whether exit is ever called is equivalent to solving the famous ``halting problem,'' known to be recursively undecidable.

Thus, most of the lint algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, lint assumes it can be called: this is not necessarily so, but in practice it is quite reasonable.

Lint tries to give information with a high degree of relevance. Messages of the form ``xxx might be a bug'' are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, the messages lose their credibility and serve merely to clutter up the output, obscuring the more important messages.

Keeping these issues in mind, we now consider in more detail the classes of messages that lint produces.

2.2.2 Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These ``errors of commission'' rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally serve to discover bugs. If a function does a necessary job, and is never called, something is wrong!

Lint complains about variables and functions that are defined but not otherwise mentioned. An exception is made for variables that are declared through explicit extern statements but are never referenced. Thus, the statement

```
extern  float  sin();
```

will evoke no comment if sin is never used. This agrees with the semantics of the C compiler.

In some cases, these unused external declarations might be of some interest: they can be discovered by adding the **-x** flag to the lint invocation.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The **-v** option is available to suppress the printing of complaints about unused arguments. When **-v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

There is one case where information about unused, or undefined, variables is more distracting than helpful. This is when lint is applied to some, but not all, files out of a collection that are to be loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. The -u flag may be used to suppress the spurious messages that might otherwise appear.

2.2.3 Set/Used Information

Lint attempts to detect cases where a variable is used before it is set. This is very difficult to do well: many algorithms take a good deal of time and space, and still produce messages about perfectly valid programs. Lint detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a ``use,'' since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. It does mean that lint can complain about some legal programs, but these programs would probably be considered bad on stylistic grounds (for example, they might contain at least two goto's). Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables which are used in the expression which first sets them.

The set/used information also permits recognition of those local variables that are set and never used: these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

2.2.4 Flow of Control

Lint attempts to detect unreachable portions of program code. It will complain about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. An attempt is made to detect loops that can never be left at the bottom, detecting the special cases **while(1)** and **for(;;)** as infinite loops. Lint also complains about loops which cannot be entered at the top: some valid programs may have such loops, but at best they are bad style and at worst, bugs.

Lint has an important area of blindness in the flow of control algorithm: it has no way of detecting functions which are called and never return. Thus, a call to exit may cause unreachable code which lint does not detect; the most serious effects of this are in the determination of returned function values, discussed in the next section.

One form of unreachable statement is not usually complained about by lint: a **break** statement that cannot be reached causes no message. Programs generated by yacc2 and especially lex3 may have literally hundreds of unreachable **break** statements. The -O flag in the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance, there is typically nothing the user can do about them, and the resulting messages would clutter up the lint output. If these messages are desired, lint can be invoked with the **-b** option.

2.2.5 Function Values

Sometimes functions return values which are never used; sometimes programs incorrectly use function ``values'' which have never been returned. Lint addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
    return( expr );
```

and

```
     return ;
```

statements is cause for alarm; lint will give the message

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

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

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

Notice that, if a tests false, f will call g and then return with no defined return value; this will trigger a complaint from lint. If g, like exit, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature. It also accounts for a substantial fraction of the ``noise'' messages produced by lint.

On a global scale, lint detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem.

2.2.6 Type Checking

Lint enforces the type checking rules of C more strictly than do the compilers. The additional checking is 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 the operands. The assignment, conditional (?:), and relational operators have this property. The argument of a **return** statement, and expressions used in initialization also suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, unsigned, float, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of _x_'s can be intermixed with pointers to _x_'s.

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

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

With enumerations, checks are made that enumeration variables or members are not mixed with other types, or other enumerations, and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

2.2.7 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_ quite rightly complains. Now, consider the assignment

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

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his intentions. It seems harsh for _lint_ to continue to complain about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The -c flag controls the printing of comments about casts. When -c is in effect, casts are treated as though they were assignments subject to complaint. Otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

2.2.8 Nonportable Character Use

On the PDP-11, characters are signed quantities, with a range from -128 to 127. On most of the other C implementations, characters take on only positive values. Thus, lint flags certain comparisons and assignments as being illegal or nonportable. For example, the fragment

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

works on the PDP-11, but will fail on machines where characters always take on positive values. The real solution is to declare c an integer, since getchar is actually returning integer values. In any case, lint 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 because on some machines bitfields are considered as signed quantities. While it may seem counter-intuitive to consider that a two bit field declared of type **int** cannot hold the value 3, the problem disappears if the bitfield is declared to have type unsigned.

2.2.9 Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which loses accuracy. This may happen in programs which have been incompletely converted to use typedefs. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, losing accuracy. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **-a** flag.

2.2.10 Strange Constructions

Several perfectly legal, but somewhat strange, constructions are flagged by lint. The messages hopefully encourage better code quality, clearer style, and may even point out bugs. The -h flag is used to enable these checks. For example, in the statement

```
     *p++ ;
```

the * does nothing. This provokes the message ``null effect'' from lint. The program fragment

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

is clearly somewhat strange. The test will never succeed. Similarly, the test

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

is equivalent to

```
     if( x != 0 )
```

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

```
     degenerate unsigned comparison
```

If one says

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

lint reports ``constant in conditional context'', since the comparison of 1 with 0 gives a constant result.

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

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

or

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

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

Finally, when the -h flag is in force lint complains about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered by many (including the author) to be bad style, usually unnecessary, and frequently a bug.

2.2.11 Ancient History

There are several forms of older syntax that are discouraged by lint. These fall into two classes, assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =-, . . .) could 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 (+=, -=, etc.) have no such ambiguities. To spur the abandonment of the older forms, lint complains about these old fashioned operators.

A similar issue arises with initialization. The older language allowed

```
     int  x  1 ;
```

to initialize x to 1. This also caused syntactic difficulties. For example

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

looks somewhat like the beginning of a function declaration:

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

and the compiler must read a fair ways past x in order to sure what the declaration really is.. Again, the problem is even more perplexing when the initializer involves a macro.

The current syntax places an equals sign between the variable and the initializer:

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

This is free of any possible syntactic ambiguity.

2.2.12 Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely 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 others, 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 whenever either the -p or -h flags are in effect.

2.2.13 Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines (like the PDP-11) in which the stack runs backwards, function arguments will probably be best evaluated from right-to-left; on machines with a stack running forward, left-to-right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

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

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

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

will draw the complaint:

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

2.2.14 Shutting Lint Up

There are occasions when the programmer is smarter than lint. There may be valid reasons for ``illegal'' type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control information produced by lint often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of communicating with lint, typically to shut it up, is desirable. Therefore, a number of words are recognized by lint when they were embedded in comments. Thus, lint directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the lint directives don't work.

The first directive is concerned with flow of control information. If a particular place in the program cannot be reached, but this is not apparent to lint, this can be asserted at the appropriate spot in the program by the directive:

```
/* NOTREACHED */
```

Similarly, if it is desired 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 flag can be turned on for one function by the directive:

```
/* ARGSUSED */
```

Complaints about 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. This can be done by following the VARARGS keyword immediately with a digit giving the number of arguments that should be checked. Thus:

```
/* VARARGS2 */
```

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

```
/* LINTLIBRARY */
```

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

2.2.15 Library Declaration Files

Lint accepts certain library directives, such as

```
-ly
```

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

```
/* LINTLIBRARY */
```

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

Lint library files are processed almost exactly like ordinary source files. The only difference is that functions that are defined on a library file, but are not used on a source file, draw no complaints. Lint does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine (this is a feature!).

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

2.2.16 Notes

Lint was a difficult program to write, partially because it is closely connected with matters of programming style, and partially because users usually don't notice bugs that cause *lint* to miss errors which it should have caught. (By contrast, if *lint* incorrectly complains about something that is correct, the programmer reports that immediately!)

A number of areas remain to be further developed. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked, and no attempt is made to match up structure and union declarations across files. Some stricter checking of the use of the **typedef** is clearly desirable, but what checking is appropriate, and how to carry it out, is still to be determined.

Lint shares the preprocessor with the C compiler. At some point it may be appropriate for a special version of the preprocessor to be constructed which checks for things such as unused macro definitions, macro arguments which have side effects which are not expanded at all, or are expanded more than once, etc.

The central problem with *lint* is the packaging of the information which it collects. There are many options which serve only to turn off, or slightly modify, certain features. There are pressures to add even more of these options.

In conclusion, it appears that the general notion of having two programs is a good one. The compiler concentrates on quickly and accurately turning the program text into bits which can be run; *lint* concentrates on issues of portability, style, and efficiency. *Lint* can afford to be wrong, since incorrectness and over-conservatism are merely annoying, not fatal. The compiler can be fast since it knows that *lint* will cover its flanks. Finally, the programmer can concentrate at one stage of the programming process solely on the algorithms, data structures, and correctness of the program, and then later retrofit, with the aid of *lint*, the desirable properties of universality and portability.

2.2.17 Current Lint Options

The command currently has the form

```
lint [-options ] files... library-descriptors...
```

The options are

h	Perform heuristic checks
p	Perform portability checks
v	Don't report unused arguments
u	Don't report unused or undefined externals
b	Report unreachable **break** statements.
x	Report unused external declarations
a	Report assignments of **long** to **int** or shorter.
c	Complain about questionable casts
n	No library checking is done
s	Same as h (for historical reasons)

2.3 MAKE: A Program Maintenance Program

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. **Make** provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell **make** the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the **make** command will create the proper files simply, correctly, and with a minimum amount of effort.

The basic operation of **make** is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file defines the graph of dependencies. **Make** does a depth-first search of this graph to determine what work is really necessary.

Make also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator (e.g., Yacc or Lex). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules. Unfortunately, it is very easy for a programmer to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, one may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations will result in a program that will not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

The program described in this report mechanizes many of the activities of program development and maintenance. If the information on inter-file dependences and command sequences is stored in a file, the simple command

```
make
```

is frequently sufficient to update the interesting files, regardless of the number that have been edited since the last ``make''. In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the **make** command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think - edit - make - test . . .

Make is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple source versions or of describing huge programs.

Basic Features The basic operation of **make** is to update a target file by ensuring that all of the files on which it depends exist and are up to date, then creating the target if it has not been modified since its dependents were. **Make** does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example: A program named prog is made by compiling and loading three C-language files x.c, y.c, and z.c with the lS library. By convention, the output of the C compilations is found in files named x.o, y.o, and z.o. Assume that the files x.c and y.c share some declarations in a file named defs, but that z.c does not. That is, x.c and y.c have the line

```
#include "defs"
```

The following text describes the relationships and operations:

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

x.o  y.o :   defs
```

If this information were stored in a file named makefile, the command

```
     make
```

would perform the operations needed to recreate *prog* after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

Make operates using three sources of information: a user-supplied description file (as above), file names and ``last-modified'' times from the file system, and built-in rules to bridge some of the gaps. In our example, the first line says that *prog* depends on three ``.*o*'' files. Once these object files are current, the second line describes how to load them to create *prog*. The third line says that *x.o* and *y.o* depend on the file *defs*. From the file system, make discovers that there are three ``.*c*'' files corresponding to the needed ``.*o*'' files, and uses built-in information on how to generate an object from a source file (*i.e.*, issue a ``cc -c'' command).

The following long-winded description file is equivalent to the one above, but takes no advantage of **make**'s innate knowledge:

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

If none of the source or object files had changed since the last time *prog* was made, all of the files would be current, and the command

```
     make
```

would just announce this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c*) would be recompiled, and then *prog* would be created from the new ``.*o*'' files. If only the file *y.c* had changed, only it would be recompiled, but it would still be necessary to reload *prog*.

If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise the specified targets are made. The command

```
make x.o
```

would recompile x.o if x.c or defs had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise the current time is used. It is often quite useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros. Thus, an entry ``save'' might be included to copy a certain set of files, or an entry ``cleanup'' might be used to throw away unneeded intermediate files. In other cases one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

Make has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign; macro names longer than one character must be parenthesized. 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. $$ is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: $*, $@, $?, and $<. They will be discussed later. The following fragment shows the use:

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

The command

```
make
```

loads the three object files with the lS library. The command

```
    make  "LIBES= -ll -lS"
```

loads them with both the Lex (``-ll'') and the Standard (``-lS'') libraries, since macro definitions on the command line override definitions in the description. (It is necessary to quote arguments with embedded blanks in XENIX commands.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

2.3.1 Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and executable commands. There is also a comment convention: all characters after a sharp (#) are ignored, as is the sharp itself. Blank lines and lines beginning with a sharp are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

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

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

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the **make** command line (see below).

Other lines give information about target files. The general form of an entry is:

```
    target ... :[:] [dependent ...] [; commands] [# ...]
    [(tab) commands] [# ...]
     ...
```

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters ``*'' and ``?'' are expanded.) A command is any string of characters not including a sharp (except in quotes) or newline. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

1. For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed. Otherwise a default creation rule may be invoked.

2. In the double-colon case, a command sequence may be associated with each dependency line. If the target is out of date with any of the files on a particular line, then the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. (The printing is suppressed in silent mode or if the command line begins with an @ sign). **Make** normally stops if any command signals an error by returning a non-zero error code. (Errors are ignored if the ``-i'' flags has been specified on the **make** command line, if the fake target name ``.IGNORE'' appears in the description file, or if the command string in the description file begins with a hyphen. Some XENIX commands return meaningless status). Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (e.g., <u>cd</u> and Shell control commands) that have meaning only within a single Shell process; the results are forgotten before the next line is executed.

Before issuing any command, certain macros are set. $@ is set to the name of the file to be ``made''. $? is set to the string of names that were found to be younger than the target. If the command was generated by an implicit rule (see below), $< is the name of the related file that caused the action, and $* is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name ``.DEFAULT'' are used. If there is no such name, **make** prints a message and stops.

2.3.2 Command Usage

The **make** command takes four kinds of arguments: macro definitions, flags, description file names, and target file names.

```
make [ flags ]  [ macro definitions ]  [ targets ]
```

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are

-i	Ignore error codes returned by invoked commands. This mode is entered if the fake target name ``.IGNORE'' appears in the description file.
-s	Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name ``.SILENT'' appears in the description file.
-r	Do not use the built-in rules.
-n	No execute mode. Print commands, but do not execute them. Even lines beginning with an ``@'' sign are printed.
-t	Touch the target files (causing them to be up to date) rather than issue the usual commands.
-q	Question. The **make** command returns a zero or non-zero status code depending on whether the target file is or is not up to date.
-p	Print out the complete set of macro definitions and target descriptions

-d Debug mode. Print out detailed information on files and times examined.

-f Description file name. The next argument is assumed to be the name of a description file. A file name of ``-'' denotes the standard input. If there are no ``-f'' arguments, the file named makefile or Makefile in the current directory is read. The contents of the description files override the built-in rules if they are present).

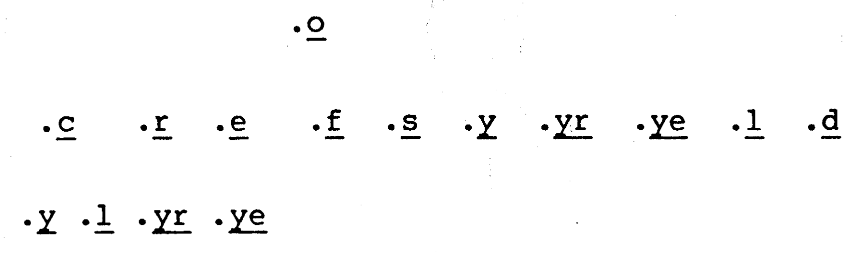
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is ``made''.

2.3.3 Implicit Rules

The make program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (The Appendix describes these tables and means of overriding them.) The default suffix list is:

.o	Object file
.c	C source file
.e	Efl source file
.r	Ratfor source file
.f	Fortran source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.ye	Yacc-Efl source grammar
.l	Lex source grammar

The following diagram summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.



If the file _x.o_ were needed and there were an _x.c_ in the description or directory, it would be compiled. If there were also an _x.l_, that grammar would be run through Lex before compiling the result. However, if there were no _x.c_ but there were an _x.l_, **make** would discard the intermediate C-language file and use the direct link in the graph above.

It is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, RC, EC, YACC, YACCR, YACCE, and LEX. The command

```
make CC=newcc
```

will cause the ``newcc'' command to be used instead of the usual C compiler. The macros CFLAGS, RFLAGS, EFLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus,

```
make "CFLAGS=-O"
```

causes the optimizing C compiler to be used.

2.3.4 Example

As an example of the use of **make**, we will present the description file used to maintain the **make** command itself. The code for **make** is spread over a number of C source files and a Yacc grammar. The description file contains:

```
# Description file for the Make command

P = lpr
FILES = Makefile version.c defs main.c doname.c misc.c files.c
OBJECTS = version.o main.o ... dosys.o gram.o
LIBES= -lS
LINT = lint -p
CFLAGS = -O

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 version.c gram.c
     $(LINT) dosys.c doname.c files.c main.c misc.c version.c gr
     rm gram.c

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

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

```
make
```

in a directory containing only the source and description file:

```
cc  -c version.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  version.o main.o ... dosys.o gram.o -lS -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the description file, make found them using its suffix rules and issued the needed commands. The string of digits results from the ``size make'' command; the printing of the command line itself was suppressed by an @ sign. The @ sign on the size command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The ``print'' entry 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:

```
make print "P = lpr"

     or

make print "P=  cat >zap"
```

2.3.5 Suggestions and Warnings

The most common difficulties arise from **make's** specific meaning of dependency. If file x.c has a ``#include "defs"'' line, 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 discover what **make** would do, the ``-n'' option is very useful. The command

```
make -n
```

orders **make** to print out the commands it would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the ``-t'' (touch) option can save a lot of time: instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the command

```
make -ts
```

(``touch silently'') causes the relevant files to appear up to date. Obvious care is necessary, since this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

The debugging flag (``-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.

2.3.6 Suffixes and Transformation Rules

The **make** program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the ``-r'' flag is used, this table is not used.

The list of suffixes is actually the dependency list for the name ``.SUFFIXES''; **make** looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, **make** acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a ``.r'' file to a ``.o'' file is thus ``.r.o''. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule ``.r.o'' is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro $< is the name of the dependent that caused the action.

The order of the suffix list is significant, since it is scanned from left to right, and the first name that is formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can just add an entry for ``.SUFFIXES'' in his own description file; the dependents will be added to the usual list. A ``.SUFFIXES'' line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed).

The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o :
    $(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
    $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
    $(AS) -o $@ $<
.y.o :
    $(YACC) $(YFLAGS) $<
    $(CC) $(CFLAGS) -c y.tab.c
    rm y.tab.c
    mv y.tab.o $@
.y.c :
    $(YACC) $(YFLAGS) $<
    mv y.tab.c $@
```

2.4 ADB: The XENIX Debugger

ADB is a useful debugging tool for debugging C programs. It provides capabilities to look at ``core'' files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This document provides examples of the more useful features of ADB. The reader is expected to be familiar with the basic commands on XENIX with the C language and able to compile simple C programs.

2.4.1 Invocation

To invoke ADB type:

```
adb objfile corefile
```

where objfile is an executable XENIX file and corefile is a core image file. Many times this will look like:

```
adb a.out core
```

or more simply:

```
adb
```

where the defaults are a.out and core respectively. The filename minus (-) means ignore this argument as in:

```
adb - core
```

ADB has requests for examining locations in either file. The ? request examines the contents of objfile, the / request examines the corefile. The general form of these requests is:

```
address ? format
```

or

```
address / format
```

2.4.2 Current Address

ADB maintains a current address, called dot, similar in function to the current pointer in the XENIX editor, ed. When an address is entered, the current address is set to that location, so that:

```
0126?i
```

sets dot to octal 126 and prints the instruction at that address. The request:

prints 10 decimal numbers starting at dot. Dot ends up refering to the address of the last item printed. When used with the ? or / requests, the current address can be advanced by typing newline; it can be decremented by typing ^.

Addresses are represented by expressions. Expressions are made up from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the operators +, -, *, % (integer division), & (bitwise and), | (bitwise inclusive or), # (round up to the next multiple), and (not). (All arithmetic within ADB is 32 bits.) When typing a symbolic address for a C program, the user can type name or _name; ADB will recognize both forms.

2.4.3 Formats

To print data, a user specifies a collection of letters and characters that describe the format of the printout. Formats are "remembered" in the sense that typing a request without one will cause the new printout to appear in the previous format. The following are the most commonly used format letters.

b	one byte in octal
c	one byte as a character
o	one word in octal
d	one word in decimal
f	two words in floating point
i	PDP 11 instruction
s	a null terminated character string
a	the value of dot
u	one word as unsigned integer
n	print a newline
r	print a blank space
^	backup dot

(Format letters are also available for "long" values, for example, `D' for long decimal, and `F' for double floating point.) For other formats see the ADB manual.

2.4.4 General Request Meanings

The general form of a request is:

```
     address,count command modifier
```

which sets `dot' to address and executes the command count times.

The following table illustrates some general ADB command meanings:

?	Print contents from a.out file
/	Print contents from core file
=	Print value of "dot"
:	Breakpoint control
$	Miscellaneous requests
;	Request separator
!	Escape to shell

ADB catches signals, so a user cannot use a quit signal to exit from ADB. The request $q or $Q (or <CONTROL-D>) must be used to exit from ADB.

2.4.5 Debugging C Programs

2.4.5.1 Debugging A Core Image Consider the C program in Figure 1. The program is used to illustrate a common error made by C programmers. The object of the program is to change the lower case "t" to upper case in the string pointed to by charp and then write the character string to the file indicated by argument 1. The bug shown is that the character "T" is stored in the pointer charp instead of the string pointed to by charp. Executing the program produces a core file because of an out of bounds memory reference.

ADB is invoked by:

```
     adb a.out core
```

The first debugging request:

```
$c
```

is used to give a C backtrace through the subroutines called. As shown in Figure 2 only one function (main) was called and the arguments argc and argv have octal values 02 and 0177762 respectively. Both of these values look reasonable; 02 = two arguments, 0177762 = address on stack of parameter vector.
The next request:

```
$C
```

is used to give a C backtrace plus an interpretation of all the local variables in each function and their values in octal. The value of the variable cc looks incorrect since cc was declared as a character.

The next request:

```
$r
```

prints out the registers including the program counter and an interpretation of the instruction at that location.

The request:

```
$e
```

prints out the values of all external variables.

A map exists for each file handled by ADB. The map for the a.out file is referenced by ? whereas the map for core file is referenced by /. Furthermore, a good rule of thumb is to use ? for instructions and / for data when looking at programs. To print out information about the maps type:

```
$m
```

This produces a report of the contents of the maps. More about these maps later.

In our example, it is useful to see the contents of the string pointed to by charp. This is done by:

```
*charp/s
```

which says use charp as a pointer in the core file and print the information as a character string. This printout clearly shows that the character buffer was incorrectly overwritten and helps identify the error. Printing the locations around charp shows that the buffer is unchanged but that the pointer is destroyed. Using ADB similarly, we could print information about the arguments to a function. The request:

```
     main.argc/d
```

prints the decimal _core_ image value of the argument _argc_ in the function _main_.
The request:

```
     *main.argv,3/o
```

prints the octal values of the three consecutive cells pointed to by _argv_ in the function _main_. Note that these values are the addresses of the arguments to main. Therefore:

```
     0177770/s
```

prints the ASCII value of the first argument. Another way to print this value would have been

```
     *"/s
```

The " means ditto which remembers the last address typed, in this case _main.argc_ ; the * instructs ADB to use the address field of the _core_ file as a pointer.

The request:

prints the current address (not its contents) in octal which has been set to the address of the first argument. The current address, dot, is used by ADB to "remember" its current location. It allows the user to reference locations relative to the current address, for example:

2.4.5.2 _Multiple Functions_ Consider the C program illustrated in Figure 3. This program calls functions _f_,g, and _h_ until the stack is exhausted and a core image is produced.

Again you can enter the debugger via:

```
     adb
```

which assumes the names _a.out_ and _core_ for the executable file and core image file respectively. The request:

```
     $c
```

will fill a page of backtrace references to _f_,g, and _h_. Figure 4 shows an abbreviated list (typing DEL will terminate the output and bring you back to ADB request level).

The request:

```
,5$C
```

prints the five most recent activations.

Notice that each function (_f_,_g_,_h_) has a counter of the number of times it was called.

The request:

```
fcnt/d
```

prints the decimal value of the counter for the function _f_. Similarly _gcnt_ and _hcnt_ could be printed. To print the value of an automatic variable, for example the decimal value of _x_ in the last call of the function _h_, type:

```
h.x/d
```

It is currently not possible in the exported version to print stack frames other than the most recent activation of a function. Therefore, a user can print everything with $C or the occurrence of a variable in the most recent call of a function. It is possible with the $C request, however, to print the stack frame starting at some address as address$C.

2.4.5.3 _Setting Breakpoints_ Consider the C program in Figure 5. This program, which changes tabs into blanks, is adapted from _Software Tools_ by Kernighan and Plauger.

We will run this program under the control of ADB (see Figure 6a) with:

```
adb a.out -
```

Breakpoints are set in the program as:

```
address:b  [request]
```

The requests:

```
     settab+4:b
     fopen+4:b
     getc+4:b
     tabpos+4:b
```

set breakpoints at the start of these functions. C does not generate statement labels. Therefore it is currently not possible to plant breakpoints at locations other than function entry points without a knowledge of the code generated by the C compiler. The above addresses are entered as symbol+4 so that they will appear in any C backtrace since the first instruction of each function is a call to the C save routine (csv). Note that some of the functions are from the C library and that this call to csv is PDP-11 dependent; each machine language requires its own form of procedure initialization.

To print the location of breakpoints one types:

```
     $b
```

The display indicates a count field. A breakpoint is bypassed count-1 times before causing a stop. The command field indicates the ADB requests to be executed each time the breakpoint is encountered. In our example no command fields are present.

By displaying the original instructions at the function settab we see that the breakpoint is set after the jsr to the C save routine. We can display the instructions using the ADB request:

```
     settab,5?ia
```

This request displays five instructions starting at settab with the addresses of each location displayed. Another variation is:

```
     settab,5?i
```

which displays the instructions with only the starting address.

Notice that we accessed the addresses from the a.out file with the ? command. In general when asking for a printout of multiple items, ADB will advance the current address the number of bytes necessary to satisfy the request; in the above example five instructions were displayed and the current address was advanced 18 (decimal) bytes.

To run the program one simply types:

```
:r
```

To delete a breakpoint, for instance the entry to the function settab, one types:

```
settab+4:d
```

To continue execution of the program from the breakpoint type:

```
:c
```

Once the program has stopped (in this case at the breakpoint for fopen), ADB requests can be used to display the contents of memory. For example:

```
$C
```

to display a stack trace, or:

```
tabs,3/8o
```

to print three lines of 8 locations each from the array called tabs. By this time (at location fopen) in the C program, settab has been called and should have set a one in every eighth location of tabs.

2.4.5.4 Advanced Breakpoint Usage We continue execution of the program with:

```
:c
```

See Figure 6b. Getc is called three times and the contents of the variable c in the function main are displayed each time. The single character on the left hand edge is the output from the C program. On the third occurrence of getc the program stops. We can look at the full buffer of characters by typing:

```
ibuf+6/20c
```

When we continue the program with:

```
:c
```

we hit our first breakpoint at tabpos since there is a tab following the "This" word of the data.

Several breakpoints of tabpos will occur until the program has changed the tab into equivalent blanks. Since we feel that tabpos is working, we can remove the breakpoint at that location by:

```
     tabpos+4:d
```

If the program is continued with:

```
     :c
```

it resumes normal execution after ADB prints the message

```
     a.out:running
```

The XENIX quit and interrupt signals act on ADB itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to ADB. The signal is saved by ADB and is passed on to the test program if:

```
     :c
```

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if:

```
     :c  0
```

is typed.

Now let us reset the breakpoint at settab and display the instructions located there when we reach the breakpoint. This is accomplished by:

```
     settab+4:b  settab,5?ia
```

It is also possible to execute the ADB requests for each occurrence of the breakpoint but only stop after the third occurrence by typing:

```
     getc+4,3:b  main.c?C
```

This request will print the local variable c in the function main at each occurrence of the breakpoint. The semicolon is used to separate multiple ADB requests on a single line.

Warning: setting a breakpoint causes the value of dot to be changed; executing the program under ADB does not change dot. Therefore:

```
     settab+4:b  .,5?ia
     fopen+4:b
```

will print the last thing dot was set to (in the example fopen+4) not the current location (settab+4) at which the program is executing.

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

```
     settab+4:b  settab,5?ia; ptab/o  *
```

could be entered after typing the above requests.

Now the display of breakpoints:

```
     $b
```

shows the above request for the settab breakpoint. When the breakpoint at settab is encountered the ADB requests are executed. Note that the location at settab+4 has been changed to plant the breakpoint; all the other locations match their original value.

Using the functions, f,g and h shown in Figure 3, we can follow the execution of each function by planting non-stopping breakpoints. We call ADB with the executable program of Figure 3 as follows:

```
     adb ex3 -
```

Suppose we enter the following breakpoints:

```
     h+4:b      hcnt/d;  h.hi/;  h.hr/
     g+4:b      gcnt/d;  g.gi/;  g.gr/
     f+4:b      fcnt/d;  f.fi/;  f.fr/
     :r
```

Each request line indicates that the variables are printed in decimal (by the specification d). Since the format is not changed, the d can be left off all but the first request.

The output in Figure 7 illustrates two points. First, the ADB requests in the breakpoint line are not examined until the program under test is run. That means any errors in those ADB requests is not detected until run time. At the location of the error ADB stops running the program.

The second point is the way ADB handles register variables. ADB uses the symbol table to address variables. Register variables, like f.fr above, have pointers to uninitialized places on the stack. Therefore the message "symbol not found".

Another way of getting at the data in this example is to print the variables used in the call as:

```
f+4:b     fcnt/d;  f.a/;  f.b/;  f.fi/
g+4:b     gcnt/d;  g.p/;  g.q/;  g.gi/
:c
```

The operator / was used instead of ? to read values from the core file. The output for each function, as shown in Figure 7, has the same format. For the function f, for example, it shows the name and value of the external variable fcnt. It also shows the address on the stack and value of the variables a,b and fi.

Notice that the addresses on the stack will continue to decrease until no address space is left for program execution at which time (after many pages of output) the program under test aborts. A display with names would be produced by requests like the following:

```
f+4:b     fcnt/d;  f.a/"a="d;  f.b/"b="d;  f.fi/"fi="d
```

In this format the quoted string is printed literally and the d produces a decimal display of the variables. The results are shown in Figure 7.

2.4.5.5 Other Breakpoint Facilities

- Arguments and change of standard input and output are passed to a program as:

  ```
  :r  arg1  arg2 ... <infile  >outfile
  ```

  This request kills any existing program under test and starts the a.out afresh.

- The program being debugged can be single stepped by:

  ```
  :s
  ```

  If necessary, this request will start up the program being debugged and stop after executing the first instruction.

- ADB allows a program to be entered at a specific address by typing:

```
address:r
```

- The count field can be used to skip the first n breakpoints as:

  ```
  ,n:r
  ```

  The request:

  ```
  ,n:c
  ```

  may also be used for skipping the first n breakpoints when continuing a program.

- A program can be continued at an address different from the breakpoint with:

  ```
  address:c
  ```

- The program being debugged runs as a separate process and can be killed with:

  ```
  :k
  ```

2.4.6 Maps

XENIX supports several executable file formats. These are used to tell the loader how to load the program file. File type 407 is the most common and is generated by a C compiler invocation such as cc pgm.c. A 410 file is produced by a C compiler command of the form:

```
cc -n pgm.c
```

Whereas a 411 file is produced by cc -i pgm.c. ADB interprets these different file formats and provides access to the different segments through a set of maps (see Figure 8). To print the maps type:

```
$m
```

In 407 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 410 files (shared text), the instructions are separated from data and ?* accesses the data part of the a.out file. The ?* 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 411 files (separated I & D space), the instructions and data are also separated. However, in this case, since data is mapped through a separate set of segmentation registers, the base of the data segment is also relative to address zero. In this case since the addresses overlap it is necessary to use the ?* operator to access the data space of the *a.out* file. In both 410 and 411 files the corresponding core file does not contain the program text.

Figure 9 shows the display of three maps for the same program linked as a 407, 410, 411 respectively. 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 (020 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 a 407 file with mixed text and data this is the same as the length of the header; for 410 and 411 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:

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

A user can access locations by using the ADB defined variables. The $v request prints the variables initialized by ADB:

b	base address of data segment
d	length of the data segment
s	length of the stack
t	length of the text
m	execution type (407,410,411)

In Figure 9 those variables not present are zero. Use can be made of these variables by expressions such as:

```
<b
```

in the address field. Similarly the value of the variable can be changed by an assignment request such as:

```
02000>b
```

that sets b to octal 2000. These variables are useful to know if the file under examination is an executable or _core_ image file.

ADB reads the header of the _core_ image file to find the values for these variables. If the second file specified does not seem to be a _core_ file, or if it is missing then the header of the executable file is used instead.

2.4.7 Advanced Usage

It is possible with ADB to combine formatting requests to provide elaborate displays. Below are several examples.

2.4.7.1 Formatted dump

The line:

```
<b,-1/4o4^8Cn
```

prints 4 octal words followed by their ASCII interpretation from the data space of the core image file. Broken down, the various request pieces mean:

<b	The base address of the data segment.
<b,-1	Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end of file) is detected.

The format 4o4^8Cn is broken down as follows:

4o	Print 4 octal locations.
4^	Backup the current address 4 locations (to the original start of the field).
8C	Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as @ followed by the corresponding character in the range 0140 to 0177. An @ is printed as @@.
n	Print a newline.

The request:

```
<b,<d/4o4^8Cn
```

could have been used instead to allow the printing to stop at the end of the data segment (<d provides the data segment size in bytes).

The formatting requests can be combined with ADB's ability to read in a script to produce a core image dump script. ADB is invoked as:

```
adb a.out core < dump
```

to read in a script file, dump, of requests. An example of such a script is:

```
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,-1/8ona
```

The request **120$w** sets the width of the output to 120 characters (normally, the width is 80 characters). ADB attempts to print addresses as:

```
symbol + offset
```

The request **4095$s** increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The request = can be used to print literal strings. Thus, headings are provided in this dump program with requests of the form:

```
=3n"C Stack Backtrace"
```

that spaces three lines and prints the literal string. The request $v prints all non-zero ADB variables (see Figure 8). The request **0$s** sets the maximum offset for symbol matches to zero thus suppressing the printing of symbolic labels in favor of octal values. Note that this is only done for the printing of the data segment. The request:

```
     <b,-1/8ona
```

prints a dump from the base of the data segment to the end of file with an octal address field and eight octal numbers per line.

Figure 11 shows the results of some formatting requests on the C program of Figure 10.

2.4.7.2 _Directory Dump_ As another illustration (Figure 12) consider a set of requests to dump the contents of a directory (which is made up of an integer _inumber_ followed by a 14 character name):

```
     adb dir -
     =n8t"Inum"8t"Name"
     0,-1? u8t14cn
```

In this example, the u prints the _inumber_ as an unsigned decimal integer, the 8t means that ADB will space to the next multiple of 8 on the output line, and the 14c prints the 14 character file name.

2.4.7.3 _Ilist Dump_ Similarly the contents of the _ilist_ of a file system could be dumped with the following set of requests:

```
     adb /dev/src -
     02000>b
     ?m <b
     <b,-1?"flags"8ton"links,uid,gid"8t3bn",
          size"8tbrdn"addr"8t8un"times"8t2Y2na
```
Last two lines should be entered as one line

In the above example, the value of the base for the map was changed to 02000 (by saying ?m<b) since that is the start of an _ilist_ within a file system. An artifice (brd above) was used to print the 24 bit size field as a byte, a space, and a decimal integer. The last access time and last modify time are printed with the 2Y operator. Figure 12 shows portions of these requests as applied to a directory and file system.

2.4.7.4 Converting values ADB may be used to convert values from one representation to another. For example,

```
     072 = odx
```

will print:

```
     072  58   #3a
```

which is the octal, decimal and hexadecimal representations of 072 (octal). The format is remembered so that typing subsequent numbers will print them in the given formats. Character values may be converted similarly, for example:

prints

```
     a    0141
```

It may also be used to evaluate expressions but be warned that all binary operators have the same precedence which is lower than that for unary operators.

2.4.8 Patching

Patching files with ADB is accomplished with the write, w or W, request (which is not like the ed editor write command). This is often used in conjunction with the locate, l or L request. In general, the request syntax for l and w are similar as follows:

```
     ?l value
```

The request l is used to match on two bytes, L is used for four bytes. The request w is used to write two bytes, whereas W writes four bytes. The value field in either locate or write requests is an expression. Therefore, decimal and octal numbers, or character strings are supported.

In order to modify a file, ADB must be called as:

```
     adb -w file1 file2
```

When called with this option, file1 and file2 are created if necessary and opened for both reading and writing.

For example, consider the C program shown in Figure 10. We can change the word "This" to "The " in the executable file for this program, ex7, by using the following requests:

```
adb -w ex7 -
?l 'Th'
?W 'The '
```

The request ?l starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of ? to write to the a.out file. The form ?* would have been used for a 411 file.

More frequently the request will be typed as:

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

and locates the first occurrence of "Th" and print the entire string. Execution of this ADB request will set dot to the address of the "Th" characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through ADB and the program run. For example:

```
adb a.out -
:s argl arg2
flag/w 1
:c
```

The :s request is normally used to single step through a process or start a process in single step mode. In this case it starts a.out as a subprocess with arguments argl and arg2. If there is a subprocess running ADB writes to it rather than to the file so the w request causes flag to be changed in the memory of the subprocess.

2.4.9 Anomalies

Below is a list of some strange things that users should be aware of.

1. Function calls and arguments are put on the stack by the C save routine. Putting breakpoints at the entry point to routines means that the function appears not to have been called via the ($c or $C command) when the breakpoint occurs.

2. When printing addresses, ADB uses either text or data symbols from the a.out file. This sometimes causes unexpected symbol names to be printed with data (e.g. savr5+022). This does not happen if ? is used for text (instructions) and / for data.

3. ADB cannot handle C register variables in the most recently activated function.

Figure 1: C program with pointer bug

```
struct buf {
    -     int fildes;
          int nleft;
          char *nextp;
          char buff[512];
          }bb;
struct buf *obuf;

char *charp "this is a sentence.";

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

          if(argc < 2) {
                    printf("Input file missing\n");
                    exit(8);
          }

          if((fcreat(argv[1],obuf)) < 0){
                    printf("%s : not found\n", argv[1]);
                    exit(8);
          }
          charp = 'T';
printf("debug 1 %s\n",charp);
          while(cc= *charp++)
                    putc(cc,obuf);
          fflush(obuf);
}
```

Figure 2: ADB output for C program of Figure 1

```
adb a.out core
$c
~main(02,0177762)
$C
~main(02,0177762)
        argc:       02
        argv:       0177762
        cc:         02124
$r
ps      0170010
pc      0204    ~main+0152
sp      0177740
r5      0177752
r4      01
r3      0
r2      0
r1      0
r0      0124
~main+0152:     mov     _obuf,(sp)
$e
savr5:     0
_obuf:     0
_charp:    0124
_errno:    0
_fout:     0
$m
text map    `ex1'
b1 = 0              e1  = 02360                 f1 = 020
b2 = 0              e2  = 02360                 f2 = 020
data map    `core1'
b1 = 0              e1  = 03500                 f1 = 02000
b2 = 0175400        e2  = 0200000               f2 = 05500
*charp/s
0124:           TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTLx        Nh@x&_
~

charp/s
_charp:         T

_charp+02:      this is a sentence.

_charp+026:     Input file missing
main.argc/d
0177756:        2
*main.argv/3o
0177762:        0177770 0177776 0177777
0177770/s
0177770:        a.out
*main.argv/3o
0177762:        0177770 0177776 0177777
*"/s
0177770:        a.out
 .=o
                0177770
 .-10/d         ~
0177756:        2
$q
```

Figure 3: Multiple function C program for stack trace illustration

```
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);
}
```

Figure 4: ADB output for C program of Figure 3

```
adb
$c
~h(04452,04451)
~g(04453,011124)
~f(02,04451)
~h(04450,04447)
~g(04451,011120)
~f(02,04447)
~h(04446,04445)
~g(04447,011114)
~f(02,04445)
~h(04444,04443)
HIT DEL KEY
adb
,5$C
~h(04452,04451)
            x:          04452
            y:          04451
            hi:         ?
~g(04453,011124)
            p:          04453
            q:          011124
            gi:         04451
            gr:         ?
~f(02,04451)
            a:          02
            b:          04451
            fi:         011124
            fr:         04453
~h(04450,04447)
            x:          04450
            y:          04447
            hi:         04451
            hr:         02
~g(04451,011120)
            p:          04451
            q:          011120
            gi:         04447
            gr:         04450
fcnt/d
_fcnt:              1173
gcnt/d
_gcnt:              1173
hcnt/d
_hcnt:              1172
h.x/d
022004:             2346
$q
```

Figure 5: C program to decode tabs

```
#define MAXLINE     80
#define YES         1
#define NO          0
#define TABSP       8

char    input[] "data";
char    ibuf[518];
int     tabs[MAXLINE];

main()
{
        int col, *ptab;
        char c;

        ptab = tabs;
        settab(ptab);       /*Set initial tab stops */
        col = 1;
        if(fopen(input,ibuf) < 0) {
                printf("%s : not found\n",input);
                exit(8);
        }
        while((c = getc(ibuf)) != -1) {
                switch(c) {
                        case '\t': /* TAB */
                                while(tabpos(col) != YES) {
                                        putchar(' ');       /* put BLANK */
                                        col++ ;
                                }
                                break;
                        case '\n':/*NEWLINE */
                                putchar('\n');
                                col = 1;
                                break;
                        default:
                                putchar(c);
                                col++ ;
                }
        }
}

/* Tabpos return YES if col is a tab stop */
tabpos(col)
int col;
{
        if(col > MAXLINE)
                return(YES);
        else
                return(tabs[col]);
}

/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
        int i;

        for(i = 0; i<= MAXLINE; i++)
                (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}
```

Figure 6a: ADB output for C program of Figure 5

```
adb a.out -
settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b
$b
breakpoints
count   bkpt              command
1       ~tabpos+04
1       _getc+04
1       _fopen+04
1       ~settab+04
settab,5?ia
~settab:          jsr     r5,csv
~settab+04:       tst     -(sp)
~settab+06:       clr     0177770(r5)
~settab+012:      cmp     $0120,0177770(r5)
~settab+020:      blt     ~settab+076
~settab+022:
settab,5?i
~settab:          jsr     r5,csv
                  tst     -(sp)
                  clr     0177770(r5)
                  cmp     $0120,0177770(r5)
                  blt     ~settab+076
:r
a.out: running
breakpoint        ~settab+04:     tst       -(sp)
settab+4:d
:c
a.out: running
breakpoint        _fopen+04:      mov       04(r5),nulstr+012
$C
_fopen(02302,02472)
~main(01,0177770)
        col:      01
        c:        0
        ptab:     03500
tabs,3/8o
03500:            01    0     0     0     0     0     0     0
                  01    0     0     0     0     0     0     0
                  01    0     0     0     0     0     0     0
```

Figure 6b: ADB output for C program of Figure 5

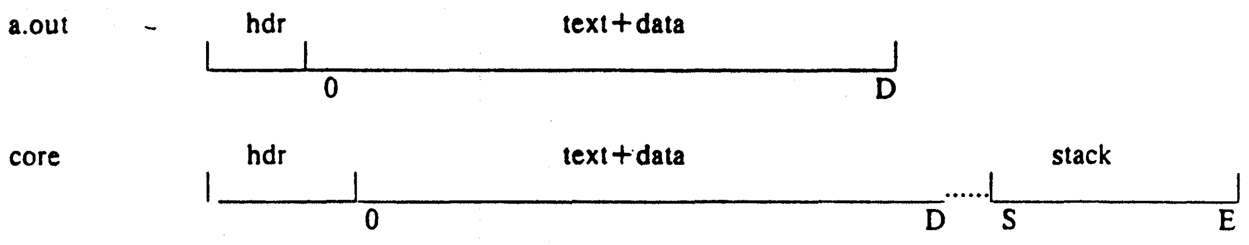
```
:c
a.out: running
breakpoint        _getc+04:       mov     04(r5),r1
ibuf+6/20c
__cleanu+0202:            This    is      a test    of
:c
a.out: running
breakpoint        ~tabpos+04:     cmp     $0120,04(r5)
tabpos+4:d
settab+4:b  settab,5?ia
settab+4:b  settab,5?ia;  0
getc+4,3:b  main.c?C;  0
settab+4:b  settab,5?ia;  ptab/o;  0
$b
breakpoints
count   bkpt              command
1       ~tabpos+04
3       _getc+04          main.c?C;0
1       _fopen+04
1       ~settab+04        settab,5?ia;ptab?o:0
~settab:          jsr     r5,csv
~settab+04:       bpt
~settab+06:       clr     0177770(r5)
~settab+012:      cmp     $0120,0177770(r5)
~settab+020:      blt     ~settab+076
~settab+022:
0177766:          0177770
0177744:          @`
T0177744:         T
h0177744:         h
i0177744:         i
s0177744:         s
```

Figure 7: ADB output for C program with breakpoints

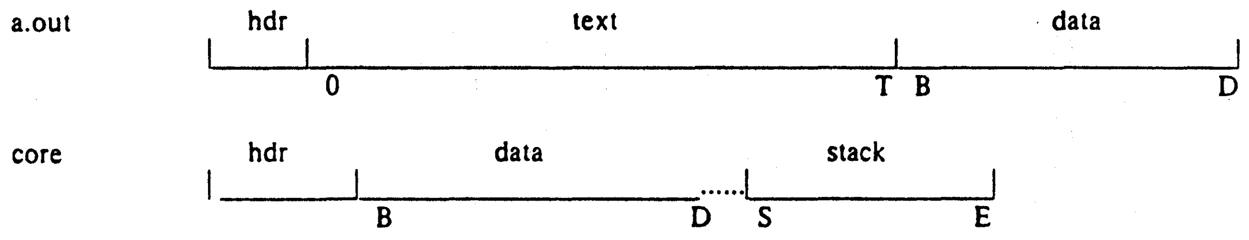
```
adb ex3 -
h+4:b hcnt/d; h.hi/; h.hr/
g+4:b gcnt/d; g.gi/; g.gr/
f+4:b fcnt/d; f.fi/; f.fr/
:r
ex3: running
_fcnt:          0
0177732:        214
symbol not found
f+4:b fcnt/d; f.a/; f.b/; f.fi/
g+4:b gcnt/d; g.p/; g.q/; g.gi/
h+4:b hcnt/d; h.x/; h.y/; h.hi/
:c
ex3: running
_fcnt:          0
0177746:        1
0177750:        1
0177732:        214
_gcnt:          0
0177726:        2
0177730:        3
0177712:        214
_hcnt:          0
0177706:        2
0177710:        1
0177672:        214
_fcnt:          1
0177666:        2
0177670:        3
0177652:        214
_gcnt:          1
0177646:        5
0177650:        8
0177632:        214
 HIT DEL
f+4:b fcnt/d; f.a/"a = "d; f.b/"b = "d; f.fi/"fi = "d
g+4:b gcnt/d; g.p/"p = "d; g.q/"q = "d; g.gi/"gi = "d
h+4:b hcnt/d; h.x/"x = "d; h.y/"h = "d; h.hi/"hi = "d
:r
ex3: running
_fcnt:          0
0177746:        a = 1
0177750:        b = 1
0177732:        fi = 214
_gcnt:          0
0177726:        p = 2
0177730:        q = 3
0177712:        gi = 214
_hcnt:          0
0177706:        x = 2
0177710:        y = 1
0177672:        hi = 214
_fcnt:          1
0177666:        a = 2
0177670:        b = 3
0177652:        fi = 214
 HIT DEL
$q
```

Figure 8: ADB address maps

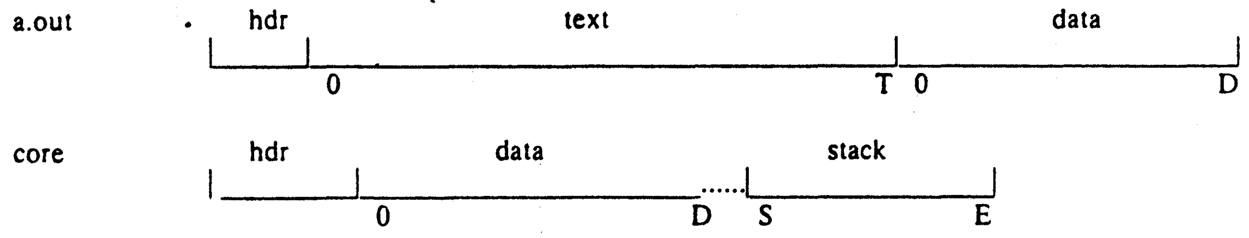
407 files



410 files (shared text)



411 files (separated I and D space)



The following *adb* variables are set.

		407	410	411
b	base of data	0	B	0
d	length of data	D	D–B	D
s	length of stack	S	S	S
t	length of text	0	T	T

Figure 9: ADB output for maps

```
adb map407 core407
$m
text map    `map407'
b1 = 0              e1      = 0256          f1 = 020
b2 = 0              e2      = 0256          f2 = 020
data map    `core407'
b1 = 0              e1      = 0300          f1 = 02000
b2 = 0175400        e2      = 0200000       f2 = 02300
$v
variables
d = 0300
m = 0407
s = 02400
$q


adb map410 core410
$m
text map    `map410'
b1 = 0              e1      = 0200          f1 = 020
b2 = 020000         e2      = 020116   f2 = 0220
data map    `core410'
b1 = 020000         e1      = 020200   f1 = 02000
b2 = 0175400        e2      = 0200000       f2 = 02200
$v
variables
b = 020000
d = 0200
m = 0410
s = 02400
t = 0200
$q


adb map411 core411
$m
text map    `map411'
b1 = 0              e1      = 0200          f1 = 020
b2 = 0              e2      = 0116          f2 = 0220
data map    `core411'
b1 = 0              e1      = 0200          f1 = 02000
b2 = 0175400        e2      = 0200000       f2 = 02200
$v
variables
d = 0200
m = 0411
s = 02400
t = 0200
$q
```

Figure 10: Simple C program for illustrating formatting and patching

```
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;
}
```

Figure 11: ADB output illustrating fancy formats

```
adb map410 core410
<b,-1/8ona
020000:       0      064124   071551   064440   020163   020141   064143   071141

_str1+016: 061541   062564   020162   072163   064562   063556   0     02

_number:
_number:   0710 0   02322040240    0   064124   071551   064440

_str2+06:  020163   064164   020145   062563   067543   062156   061440   060550

_str2+026: 060562   072143   071145   071440   071164   067151   0147 0

savr5+02:  0     0  0     0  0     0  0     0

<b,20/4o4^8Cn
020000:        0    064124   071551   064440   @`@`This i
           020163   020141   064143   071141   s a char
           061541   062564   020162   072163   acter st
           064562   063556   0     02  ring@`@`@b@`

_number:   0710 0   02322040240    H@a@`@`R@d @@
           0     064124   071551   064440   @`@`This i
           020163   064164   020145   062563   s the se
           067543   062156   061440   060550   cond cha
           060562   072143   071145   071440   racter s
           071164   067151   0147 0   tring@`@`@`
           0     0  0     0  @`@`@`@`@`@`@`@`
           0     0  0     0  @`@`@`@`@`@`@`@`
data address not found
<b,20/4o4^8t8cna
020000:        0    064124   071551   064440           This i
_str1+06:  020163   020141   064143   071141           s a char
_str1+016: 061541   062564   020162   072163           acter st
_str1+026: 064562   063556   0     02          ring
_number:
_number:   0710 0   02322040240           HR
_fpt+02:   0     064124   071551   064440          This i
_str2+06:  020163   064164   020145   062563           s the se
_str2+016: 067543   062156   061440   060550           cond cha
_str2+026: 060562   072143   071145   071440           racter s
_str2+036: 071164   067151   0147 0          tring
savr5+02:  0     0  0     0
savr5+012: 0     0  0     0
data address not found
<b,10/2b8t^2cn
020000:       0     0

_str1:     0124 0150    Th
           0151 0163    is
           040  0151     i
           0163 040     s
           0141 040     a
           0143 0150    ch
           0141 0162    ar
           0141 0143    ac
           0164 0145    te
$Q
```

Figure 12: Directory and inode dumps

```
adb dir -
=nt"Inode"t-Name"
0,-1?ut14cn

          Inode     Name
0:        652  .
          82   ..
          5971 cap.c
          5323 cap
          0    pp
```

```
adb /dev/src -
02000>b
?m<b
new map     `/dev/src'
b1 = 02000        e1    = 0100000000   f1 = 0
b2 = 0            e2    = 0            f2 = 0
$v
variables
b = 02000
<b,-1?"flags"8ton"links,uid,gid"8t3bn"size"8tbrdn"addr"8t8un"times"8t2Y2na
02000:         flags 073145
          links,uid,gid     0163 0164 0141
          size  0162 10356
          addr  28770   8236 25956    27766    25455    8236 25956    25206
          times1976 Feb 5 08:34:56  1975 Dec 28 10:55:15

02040:         flags 024555
          links,uid,gid     012  0163 0164
          size  0162 25461
          addr  8308 30050    8294 25130    15216    26890    29806    10784
          times1976 Aug 17 12:16:51 1976 Aug 17 12:16:51

02100:         flags 05173
          links,uid,gid     011  0162 0145
          size  0147 29545
          addr  25972    8306 28265    8308 25642    15216    2314 25970
          times1977 Apr 2 08:58:01  1977 Feb 5 10:21:44
```

ADB Summary

Command Summary

a. formatted printing

? _format_ print from _a.out_ file according to _format_

/ _format_ print from _core_ file according to _format_

= _format_ print the value of _dot_

?w expr write expression into _a.out_ file

/w expr write expression into _core_ file

?l expr locate expression in _a.out_ file

b. Breakpoint and program control

:b	set breakpoint at _dot_
:c	continue running program
:d	delete breakpoint
:k	kill the program being debugged
:r	run _a.out_ file under ADB control
:s	single step

c. Miscellaneous printing

$b	print current breakpoints
$c	C stack trace
$e	external variables
$f	floating registers
$m	print ADB segment maps
$q	exit from ADB
$r	general registers
$s	set offset for symbol match
$v	print ADB variables
$w	set output line width

d. Calling the shell

! call _shell_ to read rest of line

e. Assignment to variables

>_name_ assign dot to variable or register _name_

Format Summary

a	the value of dot
b	one byte in octal
c	one byte as a character
d	one word in decimal
f	two words in floating point
i	PDP 11 instruction
o	one word in octal
n	print a newline
r	print a blank space
s	a null terminated character string
_n_t	move to next _n_ space tab
u	one word as unsigned integer
x	hexadecimal
Y	date
^	backup dot
"..."	print string

Expression Summary

Expression components

decimal integer	e.g. 256
octal integer	e.g. 0277
hexadecimal	e.g. #ff
symbols	e.g. flag _main main.argc
variables	e.g. <b
registers	e.g. <pc <r0
(expression)	expression grouping

a. Dyadic operators

+	add
-	subtract
*	multiply
%	integer division
&	bitwise and
|	bitwise or
#	round up to the next multiple

b. Monadic operators

	not
*	contents of location
-	integer negate

2.5 AS: The XENIX Assembler

This document describes the usage and input syntax of the XENIX 8086 assembler **as**. **As** is an assembler that produces an output file containing relocation information and a complete symbol table. The output is acceptable to the XENIX loader **ld**, which may be used to combine the outputs of several assembler runs and to obtain object programs from libraries. The output format has been designed so that if a program contains no unresolved references to external symbols, it is executable without further processing.

2.5.1 Usage

As is invoked as follows:

```
as [ -l ] [ -o output ] file
```

If the optional `-l' argument is given, an assembly listing is produced which includes the source, the assembled (binary) code, and any assembly errors.

The output of the assembler is by default placed on the file a86.out in the current directory; The `-o' flag causes the output to be placed on the named file.

2.5.2 Lexical conventions

Assembler tokens include identifiers (alternatively, ``symbols'' or ``names''), constants, and operators.

2.5.2.1 Identifiers An identifier consists of a sequence of alphanumeric characters (including period ``.''and underscore ``_'' as alphanumeric) of which the first may not be numeric. Only the first eight characters are significant. The case of alphabetics in identifiers is significant.

2.5.2.2 Constants A hex constant consists of a sequence of digits and the letters `a', `b', `c', `d', `e', and `f' (any of which may be capitalized), preceeded by the character `/'. The letters are interpreted with the following values:

HEX	DECIMAL
A	10
B	11
C	12
D	13
E	14
F	15

An octal constant consists of a series of digits, preceded by the tilde character `` ''. The digits must be in the range from _0_ to _7_.

A decimal constant consists simply of a sequence of digits. The magnitude of the constant should be representable in 15 bits; i.e., be less than _32,768_.

2.5.2.3 _Blanks_ Blank and tab characters may be freely interspersed between tokens, but may not be used within tokens (except in character constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

2.5.2.4 _Comments_ The character ``|'' introduces a comment, which extends through the end of the line on which it appears. Comments are ignored by the assembler.

2.5.3 _Segments_

Assembled code and data fall into three segments: the **text** segment, the **data** segment, and the bss segment. The text segment is the one in which the assembly begins, and it is the one into which instructions are typically placed. The XENIX system will, if desired, enforce the purity of the text segment of programs by trapping write operations into it. Object programs produced by the assembler must be processed by the link-editor _ld_ (using its `-_i_' flag) if the text segment is to be write-protected. A single copy of the text segment is shared among all processes executing such a program.

The data segment is available for placing data or instructions which will be modified during execution. Anything which may go in the text segment may be put into the data segment. In programs with write-protected, sharable text segments, the data segment contains the initialized but variable parts of a program. If the text segment is not pure, the data segment begins immediately after the text segment. If the text segment is pure, the data segment is in an address space of its own, starting at location zero (0).

The bss segment may not contain any explicitly initialized code or data. The length of the bss segment (like that of text or data) is determined by the high-water mark of the location counter within it. The bss segment is actually an extension of the data segment and begins immediately after it. At the start of execution of a program, the bss segment is set to 0. The advantage in using the bss segment for storage that starts off empty is that the initialization information need not be stored in the output file. See also _location counter_ and _assignment statements_ below.

2.5.4 The location counter

The special symbol, ``.'', is the location counter. Its value at any time is the offset within the appropriate segment from the start of the statement in which it appears. The location counter may be assigned to, with the restriction that the current segment may not change; furthermore, the value of ``.'' may not decrease. If the effect of the assignment is to increase the value of ``.'', the required number of null bytes are generated (but see _Segments_ above).

2.5.5 Statements

A source program is composed of a sequence of _statements_. Statements are separated by new-lines. There are four kinds of statements: null statements, expression statements, assignment statements, and keyword statements.

The format for most 8086 assembly language source statements is:

[<_label field_>]
op-code [<_operand field_>] [<_comment_>]

Any kind of statement may be preceded by one or more labels.

2.5.5.1 Labels

There are two kinds of labels: name labels and numeric labels. A name label consists of a identifier followed by a colon (:). The effect of a name label is to assign the current value and type of the location counter ``.'' to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the ``.'' value assigned changes the definition of the label.

A numeric label consists of a string of digits 0 to 9 and dollar-sign ($) followed by a colon (:). Such a label serves to define local symbols of the form ``n$'', where n is the digit of the label. The scope of the numeric label is the labelled block in which it appears. As an example, the label 9$ is defined only between the lables foobar and foo:

```
foobar:
9$:     .byte 0
        .
        .
        .
foo:    .word a
```

As in the case of name labels, a numeric label assigns the current value and type of ``.'' to the symbol.

2.5.5.2 Null statements A null statement is an empty statement (which may, however, have labels and a comment). A null statement is ignored by the assembler. Common examples of null statements are empty lines or lines containing only a label.

2.5.5.3 Expression statements An expression statement consists of an arithmetic expression not beginning with a keyword. The assembler computes its value and places it in the output stream, together with the appropriate relocation bits.

2.5.5.4 Assignment statements An assignment statement consists of an identifier, an equal sign (=), and an expression. The value and type of the expression are assigned to the identifier. It is not required that the type or value be the same in pass 2 as in pass 1, nor is it an error to redefine any symbol by assignment.

Any external attribute of the expression is lost across an assignment. This means that it is not possible to declare a global symbol by assigning to it, and that it is impossible to define a symbol to be offset from a non-locally defined global symbol.

As mentioned, it is permissible to assign to the location counter ``.''. It is required, however, that the type of the expression assigned be of the same type as ``.'', and it is forbidden to decrease the value of ``.''. In practice, the most common assignment to ``.'' has the form ``.=.+n'' for some number n; this has the effect of generating n null bytes.

2.5.5.5 Keyword statements Keyword statements are numerically the most common type, since most machine instructions are of this sort. A keyword statement begins with one of the many predefined keywords of the assembler; the syntax of the remainder depends on the keyword. All the keywords are listed below with the syntax they require.

2.5.6 Expressions

An expression is a sequence of symbols representing a value. Its constituents are identifiers, constants, and operators. Each expression has a type.

Arithmetic is two's complement. All operators have equal precedence, and expressions are evaluated strictly left to right.

2.5.6.1 Expression operators The operators are:

Operator	Description
(blank)	same as +
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Logical OR
&	Logical AND
!	Logical NOT
>	Right Shift
<	Left Shift

2.5.6.2 Types The assembler deals with expressions, each of which may be of a different type. Most types are attached to the keywords and are used to select the routine which treats that keyword. The types likely to be met explicitly are:

undefined

Upon first encounter, each symbol is undefined. It may become undefined if it is assigned an undefined expression.

undefined external

A symbol which is declared **.globl** but not defined in the current assembly is an undefined external. If such a symbol is declared, the link editor _ld_ must be used to load the assembler's output with another routine that defines the undefined reference.

absolute

An absolute symbol is defined ultimately from a constant. Its value is unaffected by any possible future applications of the link-editor to the output file.

text

The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value since the program need not be the first in the link editor's output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of ``.'' is text 0.

data

The value of a data symbol is measured with respect to the origin of the data segment of a program. Like text symbols, the value of a data symbol may change during a subsequent link-editor run since previously loaded programs may have data segments. After the first .data statement, the value of ``.'' is data 0.

bss

The value of a bss symbol is measured from the beginning of the bss segment of a program. Like text and data symbols, the value of a bss symbol may change during a subsequent link-editor run, since previously loaded programs may have bss segments. After the first .bss statement, the value of ``.'' is bss 0.

external absolute, text, data, or bss

Symbols declared .globl but defined within an assembly as absolute, text, data, or bss symbols may be used exactly as if they were not declared .globl; however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

other types
: Each keyword known to the assembler has a type which is used to select the routine which processes the associated keyword statement. The behavior of such symbols when not used as keywords is the same as if they were absolute.

2.5.6.3 <u>Type propagation in expressions</u> When operands are combined by expression operators, the result has a type which depends on the types of the operands and on the operator. The rules involved are complex to state but were intended to be sensible and predictable. For purposes of expression evaluation the important types are

undefined
absolute
text
data
bss
undefined external
other

The combination rules are then: If one of the operands is undefined, the result is undefined. If both operands are absolute, the result is absolute. If an absolute is combined with one of the `other types'mentioned above, the result has the other type. If two operands of `other type' are combined, the result has the numerically larger type. An `other type' combined with an explicitly discussed type other than absolute acts like an absolute.

Further rules applying to particular operators are:

+ If one operand is text-, data-, or bss-segment relocatable, or is an undefined external, the result has the postulated type and the other operand must be absolute.

- If the first operand is a relocatable text-, data-, or bss-segment symbol, the second operand may be absolute (in which case the result has the type of the first operand); or the second operand may have the same type as the first (in which case the result is absolute). If the first operand is external undefined, the second must be absolute. All other combinations are illegal.

others
: It is illegal to apply these operators to any but absolute symbols.

2.5.7 Pseudo-operations

The keywords listed below introduce statements that influence the later operations of the assembler. The metanotation

```
     [ stuff ] ...
```

means that 0 or more instances of the given stuff may appear. Also, boldface tokens are literals, italic words are substitutable.

2.5.7.1 .even

If the location counter ``.'' is odd, it is advanced by one so the next statement will be assembled at a word boundary. This is useful for forcing storage allocation to be on a word boundary after a .byte or .ascii directive.

2.5.7.2 .float, .double

```
     .float    31459E4
```

The .float psuedo operation accepts as its operand an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, them an optional `e' or `E', followed by an optionally signed integer. The string is interpreted as a floating point number. The difference between .float and .double is in the number of bytes for the result; .float sets aside four bytes, while .double sets aside eight bytes.

2.5.7.3 .B .globl

```
     .globl name [ , name ] ...
```

This statement makes the names external. If they are otherwise defined (by assignment or appearance as a label) they act within the assembly exactly as if the .globl statement were not given; however, the link editor ld may be used to combine this routine with other routines that refer to these symbols.

Conversely, if the given symbols are not defined within the current assembly, the link editor can combine the output of this assembly with that of others which define the symbols. It is possible to force the assembler to make all otherwise undefined symbols external.

2.5.7.4 _.text, .data, .bss_ These three pseudo-operations cause the assembler to begin assembling into the text, data, or bss segment respectively. Assembly starts in the text segment. It is forbidden to assemble any code or data into the bss segment, but symbols may be defined and ``.'' moved about by assignment.

2.5.7.5 _.comm_ The format of the .comm is:

```
     .comm     ARRAY
```

Provided the _name_ is not defined elsewhere, this statement is equivalent to _.globl_. That is, the type of _name_ is ``undefined external'', and its size is _expression_. In fact the _name_ behaves in the current assembly just like an _undefined_ external. However, the link-editor _ld_ has been special-cased so that all external symbols which are not otherwise defined, and which have a non-zero value, are defined to lie in the bss segment, and enough space is left after the symbol to hold _expression_ bytes. All symbols which become defined in this way are located before all the explicitly defined bss-segment locations.

2.5.7.6 _.insrt_ The format of a .insrt is:

```
     .insrt "filename"
```

where _filename_ is any valid XENIX filename. Note that the _filename_ must be enclosed within double quotes.

The assembler will attempt to open this file for input. If it succeeds, source lines will be read from it until the end of file is reached. If _as_ was unable to open the file, a Cannot open insert file error message will be generated.

This statement is useful for including a standard set of comments or symbol assignments at the beginning of a program. It is also useful for breaking up a large source program into easily managable pieces.

A maximum depth of 10 (ten) files may be .insrted at any one time.

System call names are not predefined. They may be found in the file /_usr_/_include_/_sys_._s_.

2.5.7.7 .ascii, .asciz The .ascii directive translates character strings into their 7-bit ascii (represented as 8-bit bytes) equivalents for use in the source program. The format of the .ascii directive is as follows:

```
     .ascii     /character string/
```

where

character string contains any character valid in a character constant. Obviously, a <newline> must not appear within the character string. (It can be represented by the escape sequence \en).

/ and / are delimiter characters, which may be any character not appearing in character string

Several examples follow:

```
   Hex Code Generated:              Statement:
  22 68 65  6C 6C 6F 20 74      .ascii  /"hello there"/
  68 65 72 65 22
  77 61 72 6E 69 6E 67 20       .ascii "Warning-\007\007 \n"
  2D 07 07 20 0A
  61 62 63 64 65 66 67          .ascii *abcdefg*
```

The .asciz directive is equivalent to the .ascii directive with a zero (null) byte automatically inserted as the final character of the string. Thus, when a list or text string is printed, a search for the null character can terminate the string. Null terminated strings are used as arguments to some XENIX system calls.

2.5.7.8 .list, .nlist These pseudo-directives control the assembler output listing. These, in effect, temporarily override the `-l' switch to the assembler. This is useful when certain portions of the assembly output is not necessarily desired on a printed listing.

.list	turns the listing on
.nlist	turns the listing off

2.5.7.9 .blkb, .blkw The .blkb and .blkw directives are used to reserve blocks of storage: .blkb reserves bytes, .blkw reserves words.

The format is:

```
.blkb       [expression]
.blkw       [expression]
```

where *expression* is the number of bytes or words to reserve. If no argument is given a value of 1 is assumed. The expression must be absolute, and defined during pass 1.

This is equivalent to the statement ``.=.+*expression*'', but has a much more transparent meaning.

2.5.7.10 .byte, .word The .byte and .word directives are used to reserve bytes and words and to initialize them with certain values.

The format is:

```
.byte       [expression]
.word       [expression]
```

The .byte directive reserves one byte for each expression in the operand field and initializes the value of the byte to be the low-order byte of the corresponding expression.

For example,

```
.byte 0
                    reserves an byte, with a value
                    of zero.
state: .byte 0
                    reserves a byte  with  a  zero
                    value called state.
```

The semantics for .word are identical, except that 16-bit words are reserved and initialized.

2.5.7.11 .end The .end directive indicates the physical end of the source program. The format is:

```
.end      [expression]
```

where *expression* is an optional argument which, if present, indicates the entry point of the program, i.e. the starting point for execution. If the entry point of a program is not specified during assembly, it defaults to zero.

Every source program must be terminated with a .end statement. Inserted files which contain a .end statement will terminate assembly of the entire program, not just the inserted portion.

2.5.8 Machine

The 8086 instructions treat different types of operands uniformly. Nearly every instruction can operate on either byte or word data. In the table that follows, with some notable execeptions, an instruction that operates on a byte operand will have a b suffix on the opcode.

The 8086 instruction mnemonics which follow are implemented by the Microsoft 8086 assembler desribed in this document. Some of the opcodes are not found in any other 8086 manual.

For example, this document describes branch instructions not found in any 8086 manual. The branch instructions expand into a jump on the inverse of the condition specified, followed by an an unconditional intra-segment jump. The operand field format for the branch opcodes is the same as the operand field for the jump long opcodes. The opcodes which are implemented only in this assembler will be annotated by an asterisk, and will be fully defined and described in this document.

8086 Assembler Opcodes

Opcode	Description
aaa	ascii adjust for addition
aad	ascii adjust for division
aam	ascii adjust for multiply
aas	ascii adjust for subtraction
adc	add with carry
adcb	add with carry
add	add
addb	add
and	logical AND
andb	logical AND
*beq	long branch equal
*bge	long branch grt or equal
*bgt	long branch grt
*bhi	long branch on high
*bhis	long branch high or same
*ble	long branch les or equal
*blo	long branch on low
*blos	long branch low or same
*blt	long branch less than
*bne	long branch not equal
*br	long branch
call	intra segment call
calli	inter segment call
cbw	convert byte to word
clc	clear carry flag
cld	clear direction flag
cli	clear interrupt flag
cmc	complement carry flag
cmp	compare
cmpb	compare
cmps	compare string
cmpsb	compare string
cwd	covert word to double word
daa	decimal adjust for addition
das	decimal adjust for subtraction
dec	decrement by one
decb	decrement by one
div	divison unsigned
divb	divison unsigned
hlt	halt
idiv	integer division
idivb	integer division
imul	integer multiplication
imulb	integer multiplication
in	input byte
inc	increment by one
incb	increment by one
int	interrupt

into	interrupt if overflow
inw	input word
iret	interrupt return
j	short jump
ja	short jump if above
jae	short jump if above or equal
jb	short jump if below
jbe	short jump if below or equal
jcxz	short jump if CX is zero
je	short jump on equal
jg	short jump on greater than
jge	short jump greater than or equal
jl	short jump on less than
jle	short jump on less than or equal
jmp	jump
jmpi	inter segment jump
jna	short jump not above
jnae	short jump not above or equal
jnb	short jump not below
jnbe	short jump not below or equal
jne	short jump not equal
jng	short jump not greater
jnge	short jump not greater or equal
jnl	short jump not less
jnle	short jump not less or equal
jno	short jump not overflow
jnp	short jump not parity
jns	short jump not sign
jnz	short jump not zero
jo	short jump on overflow
jp	short jump if parity
jpe	short jump if parity even
jpo	short jump if parity odd
js	short jump if signed
jz	short jump if zero
lahf	load AH from flags
lds	load pointer using DS
lea	load effective address
les	load pointer using ES
lock	lock bus
lodb	load string byte
lodw	load string word
loop	loop short label
loope	loop if equal
loopne	loop if not equal
loopnz	loop is not zero
loopz	loop if zero
mov	move
movb	move byte
movs	move string
movsb	move string byte

mul	multipication unsigned
mulb	multipication unsigned
neg	negate
negb	negate
nop	no op
not	logical NOT
notb	logical NOT
or	logical OR
orb	logical OR
out	output byte
outw	output word
pop	pop from stack
popf	pop flag from stack
push	push onto stack
pushf	push flags onto stack
rcl	rotate left through carry
rclb	rotate left through carry
rcr	rotate right throuch carry
rcrb	rotate right throuch carry
rep	repeat string operation
repnz	repeat string operation not zero
repz	repeat string operation while zero
ret	return from procedure
reti	return from intersegment procedure
rol	rotate left
rolb	rotate left
ror	rotate right
rorb	rotate right
sahf	store AH into flagsno operands
sal	shift arithmetic left
salb	shift arithmetic left
sar	shift arithmetic right
sarb	shift arithmetic right
sbb	subtract with borrow
sbbb	subtract with borrow
scab	scan string
shl	shift logical left
shlb	shift logical left
shr	shidr logical right
shrb	shidr logical right
stc	set carry flag
std	set direction flag
sti	set interrupt enable flag
stob	store byte string
stow	store word string
sub	subtraction
subb	subtraction
test	test
testb	test
wait	wait while TEST pin
xchg	exchange

xchgb	exchange
xlat	translate
xor	xclusive OR
xorb	xclusive OR

2.5.9 Addressing Modes

The 8086 assembler provides many different ways to access instruction operands. Operands may be contained in registers, within the instruction itself, in memory, or in I/O ports. In addition, the addresses of memory and I/O port operands can be calculated in several different ways.

2.5.9.1 Register Operands

Instructions that specify only register operands are generally the most compact and fastest executing of all the instruction forms. This is because the register `addresses' are encoded in the instructions with just a few bits, and because these operations are performed entirely within the CPU. Registers may serve as source operands, destination operands, or both.

EXAMPLES OF REGISTER ADDRESSING

```
sub      cx,di
mv       ax,/3*4
mv       /3*4/,ax
mov      ax,*1
```

2.5.9.2 Immediate Operands

Immediate operands are constant data contained in an instruction. The data may be either 8 or 16 bits in length. Immediate operands can be accessed quickly because they are available directly from the instruction queue; it is possible that no bus cycles will be needed to obtain an immediate operand. An immediate operand is always a constant value and can only be used as a source operand.

The assembler can accept both 8 and 16 bit operands. It does not perform any checking on the operand size, but determines the size of the operand by the following symbols:

*expr	an 8 bit immediate
#expr	a 16 bit immediate

EXAMPLES OF IMMEDIATE ADDRESSING

```
mov       cx,*PAGSIZ/2
mov       cx,#PAGSIZ/2
mov       map,#PAGSIZ/2
mov       map,*PAGSIZ/2
```

2.5.10 Memory Addressing Modes

When reading or writing a memory operand, a value called the offset is required. This offset value, also called the effective address is the operand's distance in bytes from the beginning of the segment in which it resides.

2.5.10.1 Direct Addressing

Direct addressing is the simplest memory addressing mode since no registers are involved. The effective address is taken directly from the displacement field of the instruction. It is typically used to access simple (scalar) variables.

EXAMPLES OF DIRECT ADDRESSING

```
push      *6(bp)
mov       cx,#256
add       si,*4
```

2.5.10.2 Register Indirect Addressing

The effective address of a memory operand may be taken from a base or index register. One instruction can operate on many different memory locations if the value in the base or index register is updated appropriately. Indirect addressing is denoted by an ampersand @ preceding the operand.

EXAMPLES OF INDIRECT ADDRESSING

```
popl      rr0,@r15
calli     @moncall
```

2.5.10.3 Based Addressing

In based addressing, the effective address is the sum of a displacement value and the content of register bx or bp. Based addressing also provides a straightforward way to address structures which may be located in different places in memory. A base register can be pointed at the base of the structure and elements of the structure addressed by their displacements from the base. Different copies of the same structure can be accessed by simply changing the base register.

EXAMPLE OF BASED ADDRESSING

```
mov       *2(si),#/1000
```

2.5.10.4 Indexed Addressing In indexed addressing, the effective address is calculated from the sum of a displacement plus the content of an index register. Indexed addressing often is used to access elements in an array. The displacement locates the beginnning of the array, and the value of the index register selects one element. Since all array elements are the same length, simple arithmetic on the index register will select any element.

EXAMPLE OF INDEXED ADDRESSING

```
mov       #_cat,(bx)
```

2.5.10.5 Based Indexed Addressing Based indexed addressing generates an effective address that is the sum of a base register, an index register, and a displacement. Based indexed addressing is a very flexible mode because two address components can be varied at execution time.

Based indexed addressing provides a convenient way for a procedure to address an array allocated on a stack. Register bp can contain the offset of a reference point on the stack, typically the top of the stack after the procedure has saved registers and allocated local storage. The offset of the beginning of the array from the reference point can be expressed by a displacement value, and an index register can be used to access individual array elements.

EXAMPLES OF BASED INDEXED ADDRESSING

```
mov       (bx)(dx),_sym
mov       *2(bx)(dx),_sym
mov       #2(bx)(dx),_sym
```

2.5.11 Diagnostics

When syntactic errors occur, the line number and the file in which they occur is displayed. Errors in pass 1 cause cancellation of pass 2.

```
***ERROR*** syntax error, line xx
file: yy errors
```

where *xx* represents the line number(s) in error, and *yy* represents the total number of errors.

CHAPTER 3

ENVIRONMENT

Although the C programming language is a fine language, it is designed to be used in a computing environment. From within some C programs, you may want to execute other programs, or to make calls to perform system functions. Also, you may want to write assembly language routines that interface to programs. Before you can perform any of these programming tasks, you must have a knowledge of the XENIX environment. In the case of the XENIX system, this environment includes low level system calls, available C libraries, and compiler calling conventions. The rest of this chapter explains the various parts of the XENIX environment.

3.1 THE C INTERFACE TO THE XENIX SYSTEM

This section shows how to interface C programs to the XENIX system, either directly or through the standard I/O library. The topics discussed include:

- Handling command arguments
- Rudimentary I/O
- The standard input and output
- The standard I/O library
- File system access
- Low-level I/O: open, read, write, close, seek
- Processes: exec, fork, pipes
- Signals and interrupts

3.1.1 Basics

3.1.1.1 Program Arguments When a C program is run as a command, the arguments on the command line are made available to the function main as an argument count argc and an array argv of pointers to character strings that contain the arguments. By convention, argv[0] is the command name itself, so argc is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal. (This is essentially the echo command.)

```
main(argc, argv)          /* echo arguments */
int argc;
char *argv[];
{
        int i;

        for (i = 1; i < argc; i++)
                printf("%s%c", argv[i], (i<argc-1) ? ' ' : '0);
}
```

argv is a pointer to an array whose individual elements are pointers to arrays of characters; each is terminated by \0, so they can be treated as strings. The program starts by printing argv[1] and loops until it has printed them all.

The argument count and the arguments are parameters to main. If you want to keep them around so other routines can get at them, you must copy them to external variables.

3.1.1.2 The ``Standard Input'' and ``Standard Output'' The simplest input mechanism is to read the ``standard input,'' which is generally the user's terminal. The function getchar returns the next input character each time it is called. A file may be substituted for the terminal by using the < convention: if prog uses getchar, then the command line:

```
     prog <file
```

causes prog to read file instead of the terminal. prog itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the pipe mechanism. For example

```
     otherprog | prog
```

provides the standard input for prog from the standard output of otherprog.

Getchar returns the value EOF when it encounters the end of file (or an error) on whatever you are reading. The value of EOF is normally defined to be -1, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, putchar(c) puts the character c on the ``standard output,'' which is also by default the terminal. The output can be captured on a file by using >. If prog uses putchar,

```
     prog >outfile
```

writes the standard output on outfile instead of the terminal. Outfile is created if it doesn't exist; if it already exists, its previous contents are overwritten.

The function printf, which formats output in various ways, uses the same mechanism as putchar does, so calls to printf and putchar may be intermixed in any order: the output appears in the order of the calls.

Similarly, the function scanf provides for formatted input conversion; it reads the standard input and breaks it up into strings, numbers, etc., as desired. Scanf uses the same mechanism as getchar, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with getchar, putchar, scanf, and printf may be entirely adequate, and it is almost always enough to get started. This is particularly true if the XENIX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ASCII control characters from its input (except for new-line and tab).

```
#include <stdio.h>

main()  /* ccstrip: strip non-graphic characters */
{
        int c;
        while ((c = getchar()) != EOF)
                if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n'
                        putchar(c);
        exit(0);
}
```

The line

```
     #include <stdio.h>
```

should appear at the beginning of each source file. It causes the C compiler to read a file (/usr/include/stdio.h) of standard routines and symbols that includes the definition of EOF.

If it is necessary to treat multiple files, you can use cat to collect the files for you:

```
     cat file1 file2 ... | ccstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to exit at the end is not necessary to make the program work properly, but it assures that any caller of the program will see a normal termination status (conventionally 0) from the program when it completes. Status returns are discussed later in more detail.

3.1.2 The Standard I/O Library

The Standard I/O Library is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

In this section, we will discuss the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

3.1.2.1 File Access The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. The next step is to write a program that accesses a file that is not already connected to the program. One simple example is wc, which counts the lines, words and characters in a set of files. For instance, the command

```
     wc x.c y.c
```

prints the number of lines, words and characters in x.c and y.c and the totals.

The question is how to arrange for the named files to be read-that is, how to connect the file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be opened by the standard library function fopen. fopen takes an external name (like x.c or y.c), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a file pointer, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including stdio.h is a structure definition called FILE. The only declaration needed for a file pointer is exemplified by

```
     FILE    *fp, *fopen();
```

This says that fp is a pointer to a FILE, and fopen returns a pointer to a FILE. FILE(is a type name, like int, not a structure tag.

The actual call to fopen in a program is

```
     fp = fopen(name, mode);
```

The first argument of fopen is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The only allowable modes are read (r), write (w), or append (a).

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, fopen returns the null pointer value NULL (which is defined as zero in stdio.h).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which getc and putc are the simplest. Getc returns the next character from a file. It needs the file pointer to tell it what file. Thus:

```
c = getc(fp)
```

places in c the next character from the file referred to by fp; it returns EOF when it reaches end of file. Putc is the inverse of getc. For example

```
putc(c, fp)
```

puts the character c on the file fp and returns c. Getc and putc return EOF on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called stdin, stdout, and stderr. Normally these are all connected to the terminal, but may be redirected to files or pipes. Stdin, stdout and stderr are pre-defined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type FILE * can be. They are constants, however, not variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write wc. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order; if there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv)          /* wc: count lines, words, chars */
int argc;
char *argv[];
{
        int c, i, inword;
        FILE *fp, *fopen();
        long linect, wordct, charct;
        long tlinect = 0, twordct = 0, tcharct = 0;

        i = 1;
        fp = stdin;
        do {
                if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
                        fprintf(stderr, "wc: can't open %s\n", argv[i]);
                        continue;
                }
                linect = wordct = charct = inword = 0;
                while ((c = getc(fp)) != EOF) {
                        charct++;
                        if (c == '\n')
                                linect++;
                        if (c == ' ' || c == '\t' || c == '\n')
                                inword = 0;
                        else if (inword == 0) {
                                inword = 1;
                                wordct++;
                        }
                }
                printf("%7ld %7ld %7ld", linect, wordct, charct);
                printf(argc > 1 ? " %s\n" : "\n", argv[i]);
                fclose(fp);
                tlinect += linect;
                twordct += wordct;
                tcharct += charct;
        } while (++i < argc);
        if (argc > 2)
                printf("%7ld %7ld %7ld total\n", tlinect, twordct,
                        tcharct);
        exit(0);
}
```

The function fprintf is identical to printf, save that the first argument is a file pointer that specifies the file to be written.

The function fclose is the inverse of fopen; it breaks the connection between the file pointer and the external name that was established by fopen, freeing the file pointer for another file. Since there is a limit on the number of files that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason to call fclose on an output file-it flushes the buffer in which putc is collecting output. fclose(is called automatically for each open file when a program terminates normally.)

3.1.2.2 Error Handling-Stderr and Exit Stderr is assigned to a program in the same way that stdin and stdout are. Output written on stderr appears on the user's terminal even if the standard output is redirected. wc writes its diagnostics on stderr instead of stdout so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function exit to terminate program execution. The argument of exit is available to whatever process called it, so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations.

exit itself calls fclose for each open output file, to flush out any buffered output, then calls a routine named _exit. The function _exit causes immediate termination without any buffer flushing; it may be called directly if desired.

3.1.2.3 Miscellaneous I/O Functions The standard I/O library provides several other I/O functions besides those we have illustrated above.

Normally output with putc, etc., is buffered (except to stderr); to force it out immediately, use fflush(fp).

fscanf is identical to scanf, except that its first argument is a file pointer (as with fprintf) that specifies the file from which the input comes; it returns EOF at end of file.

The functions sscanf and sprintf are identical to fscanf and fprintf, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for sscanf and into it for sprintf.

fgets(buf, size, fp) copies the next line from fp, up to and including a new-line, into buf; at most size-1 characters are copied; it returns NULL at end of file. fputs(buf, fp) writes the string in buf onto file fp.

The function ungetc(c, fp) ``pushes back'' the character c onto the input stream fp; a subsequent call to getc, fscanf, etc., will encounter c. Only one character of push-back per file is permitted.

3.1.3 Low-Level I/O

This section describes the bottom level of I/O on the XENIX system. The lowest level of I/O in XENIX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

3.1.3.1 File Descriptors In the XENIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called ``opening'' the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a filedescriptor. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of and in Fortran.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

File pointers are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the ``shell'') runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

If I/O is redirected to and from files with < and >, as in

```
     prog <infile >outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

3.1.3.2 Read and Write All input and output is done by two functions called *read* and *write*. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
     n_read = read(fd, buf, n);

     n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than *n* bytes remained to be read. (When the file is a terminal, *read* normally reads only up to the next new-line, which is generally less than what was requested.) A return value of zero bytes implies end of file, and -1 indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time (``unbuffered''), and 512, which corresponds to a physical block size on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define BUFSIZE 512     /* best size for PDP-11 UNIX */

main()  /* copy input to output */
{
        char    buf[BUFSIZE];
        int     n;

        while ((n = read(0, buf, BUFSIZE)) > 0)
                write(1, buf, n);
        exit(0);
}
```

If the file size is not a multiple of BUFSIZE, some read will return a smaller number of bytes to be written by write; the next call to read after that will return zero.

It is instructive to see how read and write can be used to construct higher level routines like getchar, putchar, etc. For example, here is a version of getchar which does unbuffered input.

```
#define CMASK   0377    /* for making char's > 0 */

getchar()       /* unbuffered single character input */
{
        char c;

        return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

c must be declared char, because read accepts a character pointer. The character being returned must be masked with 0377 to ensure that it is positive; otherwise sign extension may make it negative. (The constant 0377 is appropriate for the but not necessarily for other machines.)

The second version of getchar does input in big chunks, and hands out the characters one at a time.

```
#define CMASK   0377     /* for making char's > 0 */
#define BUFSIZE 512

getchar()       /* buffered version */
{
        static char     buf[BUFSIZE];
        static char     *bufp = buf;
        static int      n = 0;

        if (n == 0) {   /* buffer is empty */
                n = read(0, buf, BUFSIZE);
                bufp = buf;
        }
        return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

3.1.3.3 Open, Creat, Close, Unlink Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, open and creat [sic].

open is rather like the fopen discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an int.

```
     int fd;

     fd = open(name, rwmode);
```

As with fopen, the name argument is a character string corresponding to the external file name. The access mode argument is different, however: rwmode is 0 for read, 1 for write, and 2 for read and write access. open returns -1 if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to open a file that does not exist. The entry point creat is provided to create new files, or to re-write old ones.

```
     fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called name, and -1 if not. If the file already exists, creat will truncate it to zero length; it is not an error to creat a file that already exists.

If the file is brand new, creat creates it with the protectionmode specified by the pmode argument. In the XENIX file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the XENIX utility **cp**, a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv)         /* cp: copy fl to f2 */
int argc;
char *argv[];
{
        int     fl, f2, n;
        char    buf[BUFSIZE];

        if (argc != 3)
                error("Usage: cp from to", NULL);
        if ((fl = open(argv[1], 0)) == -1)
                error("cp: can't open %s", argv[1]);
        if ((f2 = creat(argv[2], PMODE)) == -1)
                error("cp: can't create %s", argv[2]);

        while ((n = read(fl, buf, BUFSIZE)) > 0)
                if (write(f2, buf, n) != n)
                        error("cp: write error", NULL);
        exit(0);
}

error(sl, s2)   /* print error message and die */
char *sl, *s2;
{
        printf(sl, s2);
        printf("0);
        exit(1);
}
```

As we said earlier, there is a limit (typically 15-25) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine close breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via _exit_ or return from the main program closes all open files.

The function _unlink_(_filename_) removes the file _filename_ from the file system.

3.1.3.4 _Random Access-Seek and Lseek_ File I/O is normally sequential: each _read_ or _write_ takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call _lseek_ provides a way to move around in a file without actually reading or writing:

```
     lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is _fd_ to move to position _offset_, which is taken relative to the location specified by _origin_. Subsequent reading or writing will begin at that position. _offset_ is a _long_; _fd_ and _origin_ are _int_'s. _origin_ can be 0, 1, or 2 to specify that _offset_ is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
     lseek(fd, 0L, 2);
```

To get back to the beginning (``rewind''),

```
     lseek(fd, 0L, 0);
```

Notice the _0L_ argument; it could also be written as (_long_) _0_.

With _lseek_, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
     get(fd, pos, buf, n) /* read n bytes from position pos */
     int fd, n;
     long pos;
     char *buf;
     {
             lseek(fd, pos, 0);      /* get to pos */
             return(read(fd, buf, n));
     }
```

Before Version 7, the basic entry point to the XENIX I/O system was called _seek_. _seek_ is identical to _lseek_, except that its _offset_ argument is an _int_ rather than a _long_. Accordingly, since integers have only 16 bits, the _offset_ specified for _seek_ is limited to 65,535; for this reason, _origin_ values of 3, 4, 5 cause _seek_ to multiply the given offset by 512 (the number of bytes in one physical block) and then interpret _origin_ as if it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large file requires two seeks, first one which selects the block, then one which has _origin_ equal to 1 and moves to the desired byte within the block.

3.1.3.5 _Error Processing_ The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of -1. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell _errno_. The meanings of the various error numbers are listed in the introduction to Section II of the _XENIX Programmer's Manual_, so your program can, for example, determine if an attempt to open a file failed because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine _perror_ will print a message associated with the value of _errno_; more generally, _sys errno_ is an array of character strings which can be indexed by _errno_ and printed by your program.

3.1.4 Processes

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

3.1.4.1 _The ``System'' Function_ The easiest way to execute a program from another is to use the standard library routine _system_. _system_ takes one argument, a command string exactly as typed at the terminal (except for the new-line at the end) and executes it. For instance, to time-stamp the output of a program,

```
main()
{
        system("date");
        /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of sprintf may be useful.

Remember than getc and putc normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use fflush; for input, see setbuf in the appendix.

3.1.4.2 Low-Level Process Creation-Execl and Execv If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's system routine is based on.

The most basic operation is to execute another program without returning, by using the routine execl. To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to execl is the filename of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this; the end of the list is marked by a NULL argument.

The execl call overlays the existing program with the new one, runs that, then exits. There is no return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an execl call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where date is located, say

```
     execl("/bin/date", "date", NULL);
     execl("/usr/bin/date", "date", NULL);
     fprintf(stderr, "Someone stole 'date'0);
```

A variant of _execl_ called _execv_ is useful when you don't know in advance how many arguments there are going to be. The call is

```
     execv(filename, argp);
```

where _argp_ is an array of pointers to the arguments; the last pointer in the array must be _NULL_ so _execv_ can tell where the list ends. As with _execl_, _filename_ is the file in which the program is found, and _argp[0]_ is the name of the program. (This arrangement is identical to the _argv_ array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories-you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like <, >, *, ?, and [] in the argument list. If you want these, use _execl_ to invoke the shell _sh_, which then does all the work. Construct a string _commandline_ that contains the complete command as it would have been typed at the terminal, then say

```
     execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, /_bin_/_sh_. Its argument -_c_ says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in _commandline_.

3.1.4.3 _Control of Processes-Fork and Wait_ So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with _execl_ or _execv_. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called _fork_:

```
     proc_id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of _proc id_, the ``process id.'' In one of these processes (the ``child''), proc id is zero. In the other (the ``parent''), proc id is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
        execl("/bin/sh", "sh", "-c", cmd, NULL);          /* in child
```

And in fact, except for handling errors, this is sufficient. The fork makes two copies of the program. In the child, the value returned by fork is zero, so it calls execl which does the command and then dies. In the parent, fork returns non-zero so it skips the execl. (If there is any error, fork returns -1).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function wait:

```
     int status;

     if (fork() == 0)
             execl( ... );
     wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the execl or fork, or the possibility that there might be more than one child running simultaneously. (The wait returns the process id of the terminated child, if you want to check it against the value returned by fork.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in status). Still, these three lines are the heart of the standard library's system routine, which we'll show in a moment.

The status returned by wait encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to exit which caused a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither fork nor the exec calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the execl. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

3.1.4.4 Pipes A pipe is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
ls | pr
```

which connects the standard output of ls to the standard input of pr. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call pipe creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int     fd[2];

stat = pipe(fd);
if (stat == -1)
        /* there was an error ... */
```

fd is an array of two file descriptors, where fd[0] is the read side of the pipe and fd[1] is for writing. These may be used in read, write and close calls just like any other file descriptors.

If a process reads a pipe which is empty, it will wait until data arrives; if a process writes into a pipe which is too full, it will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent read will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called popen(cmd, mode), which creates a process cmd (just as system does), and returns a file descriptor that will either read or write that process, according to mode. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the pr command; subsequent write calls using the file descriptor fout will send their data to that process through the pipe.

popen first creates the the pipe with a pipe system call; it then forks to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via execl) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define READ    0
#define WRITE   1
#define tst(a, b)       (mode == READ ? (b) : (a))
static  int     popen_pid;

popen(cmd, mode)
char    *cmd;
int     mode;
{
        int p[2];

        if (pipe(p) < 0)
                return(NULL);
        if ((popen_pid = fork()) == 0) {
                close(tst(p[WRITE], p[READ]));
                close(tst(0, 1));
                dup(tst(p[READ], p[WRITE]));
                close(tst(p[READ], p[WRITE]));
                execl("/bin/sh", "sh", "-c", cmd, 0);
                _exit(1);       /* disaster has occurred if we ge
        }
        if (popen_pid == -1)
                return(NULL);
        close(tst(p[READ], p[WRITE]));
        return(tst(p[WRITE], p[READ]));
}
```

The sequence of closes in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first close closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The close closes file descriptor 0, that is, the standard input. dup is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the dup is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. (Yes, this is a bit tricky, but it's a standard idiom.) Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function pclose to close the pipe created by popen. The main reason for using a separate function rather than close is that it is desirable to wait for the termination of the child process. First, the return value from pclose indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the wait lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd)      /* close pipe fd */
int fd;
{
        register r, (*hstat)(), (*istat)(), (*qstat)();
        int      status;
        extern int popen_pid;

        close(fd);
        istat = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        hstat = signal(SIGHUP, SIG_IGN);
        while ((r = wait(&status)) != popen_pid && r != -1);
        if (r == -1)
                status = -1;
        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        signal(SIGHUP, hstat);
        return(status);
}
```

The calls to signal make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable popen pid; it really should be an array indexed by file descriptor. A popen function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

3.1.5 Signals and Interrupts

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals: interrupt, which is sent when the character is typed; quit, generated by the character; hangup, caused by hanging up the phone; and terminate, generated by the kill command. When one of these events occurs, the signal is sent to all processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the quit case, a core image file is written for debugging purposes.

The routine which alters the default action is called signal. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address is either a function, or a somewhat strange code that requests that the signal either be ignored, or that it be given the default action. The include file signal.h gives names for the various arguments, and should always be included when signals are used. Thus

```
     #include <signal.h>
       ...
     signal(SIGINT, SIG_IGN);
```

causes interrupts to be ignored, while

```
     signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, signal returns the previous value of the signal. The second argument to signal may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main()
{
        int onintr();

        if (signal(SIGINT, SIG_IGN) != SIG_IGN)
                signal(SIGINT, onintr);

        /* Process ... */

        exit(0);
}

onintr()
{
        unlink(tempfile);
        exit(1);
}
```

Why the test and the double call to <u>signal</u>? Recall that signals like interrupt are sent to <u>all</u> processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by &), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the <u>onintr</u> routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that <u>signal</u> returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf sjbuf;

main()
{
        int (*istat)(), onintr();

        istat = signal(SIGINT, SIG_IGN);        /* save original sta
        setjmp(sjbuf);  /* save current stack position */
        if (istat != SIG_IGN)
                signal(SIGINT, onintr);

        /* main processing loop */
}

    onintr()
    {
            printf("Onterrupt0);
            longjmp(sjbuf); /* return to saved state */
    }
```

The include file setjmp.h declares the type jmp buf an object in which the state can be saved. sjbuf is such an object; it is an array of some sort. The setjmp routine then saves the state of things. When an interrupt occurs, a call is forced to the onintr routine, which can print a message, set flags, or whatever. longjmp takes as argument an object stored into by setjmp, and restores control to the location after the call to setjmp, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling exit or longjmp, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that ``execution resumes at the exact point it was interrupted,'' the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for ``errors'' which are caused by interrupted system calls. (The ones to watch out for are reads from a terminal, wait, and pause.) A program whose onintr program just sets intflag, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
     if (getchar() == EOF)
             if (intflag)
                     /* EOF caused by interrupt */
             else
                     /* true end-of-file */
```

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like ``!'' in the editor) whereby other programs can be executed. Then the code should look something like this:

```
if (fork() == 0)
        execl( ... );
signal(SIGINT, SIG_IGN);        /* ignore interrupts */
wait(&status);  /* until the child is done */
signal(SIGINT, onintr); /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function system:

```
#include <signal.h>

system(s)       /* run command string s */
char *s;
{
        int status, pid, w;
        register int (*istat)(), (*qstat)();

        if ((pid = fork()) == 0) {
                execl("/bin/sh", "sh", "-c", s, 0);
                _exit(127);
        }
        istat = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        while ((w = wait(&status)) != pid && w != -1)
                ;
        if (w == -1)
                status = -1;
        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        return(status);
}
```

As an aside on declarations, the function signal obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values SIG IGN and SIG DFL have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the PDP-11; the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

```
    #define SIG_DFL (int (*)())0
    #define SIG_IGN (int (*)())1
```

3.2 THE C LIBRARY

A knowledge of the C library is invaluable to the C programmer, since it defines a common set of macros, types, and functions that can be used in almost any programming project. The most imporant functions and macros are declared in the standard I/O library, discussed below.

3.2.1 The Standard I/O Library

The standard I/O library was designed with the following goals in mind.

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.

2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.

3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines other than the PDP-11 running a version of XENIX .

3.2.2 General Usage

Each program using the library must have the line

```
     #include <stdio.h>
```

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore _ to reduce the possibility of conflict with other names created by the user. The names intended to be visible outside the package are:

stdin	The name of the standard input file
stdout	The name of the standard output file
stderr	The name of the standard error file
EOF	is actually -1, and is the value returned by the read routines on end-of-file or error.
NULL	is a notation for the null pointer, returned by pointer-valued functions to indicate an error
FILE	expands to *struct _iob* and is a useful shorthand when declaring pointers to streams.

BUFSIZ is a number (viz. 512) of the size suitable for an I/O buffer supplied by the user. See setbuf, below.

Getc,getchar,putc, putchar,feof,ferror, and fileno are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions. Thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names stdin, stdout, and stderr are in effect constants and may not be assigned to. Stdio.h contains the definitions of NULL , EOF , FILE , and BUFSIZ . The standard input file (stdin), standard output file (stdout), and standard error file (stderr) are also defined here. These definitions are incorporated into a program with the following statement:

```
#include <stdio.h>
```

The file ctype.h provides the macro definitions for the character classifications that are now possible. Any program using those facilities must contain the line:

```
#include <ctype.h>
```

The functions that handle signals need to include the signal definitions. This can be done with the line:

```
#include <signal.h>
```

Some function names have changed in order to follow the established convention. To insure that the uniqueness of function names is preserved even if truncation occurs on some systems, those functions dealing with entire strings are named str...; those functions that consider only the first n characters of a string are named strn....

Listed below are some common C library functions that you should study, most of these belong to the standard I/O library- although other libraries are represented here as well.

3.2.3 File access

fclose

```
#include <stdio.h>
int fclose(stream)
FILE *stream;
```

Fclose closes a file that was opened by fopen, frees any buffers after emptying them, and returns zero on success, non-zero on error. Exit calls fclose for all open files as part of its processing.

fdopen

```
#include <stdio.h>
FILE *fdopen (fildes, type)
int fildes;
char *type;
```

Fdopen is used strictly on XENIX systems and therefore is not a portable function. Its value is in providing a bridge between the low-level input-output (I/O) facilities of XENIX and the standard I/O functions. Fdopen associates a stream with a valid file descriptor obtained from a XENIX system call (e.g., open). Type is the same mode (r , w , a , r+ , w+ , a+) that was used in the original creation of a file identified by fildes . Fdopen returns a pointer to the associated stream, or NULL if unsuccessful.

Example:

```
int fd;
char *name = "myfile";
FILE *strm;

fd = open(name,0);

        .
        .
        .
        if((strm = fdopen(fd,"r")) == NULL)
                fprintf(stderr,"Error on %d0,fd);
```

fileno

```
#include <stdio.h>
int fileno (stream)
FILE *stream;
```

Implemented as a macro on XENIX, (and contained in the file stdio.h), fileno returns an integer file descriptor associated with a valid stream . Any existing non-XENIX implementations may have different meanings for the integer which is returned. Fileno is used by many other standard functions in the C library.

fopen

```
#include <stdio.h>
FILE *fopen (filename, type)
char *filename, *type;
```

Fopen opens a file named filename and returns a pointer to a structure (hereafter referred to as stream), containing the data necessary to handle a stream of data. Type is one of the following character strings:

r	used to open for reading.
w	used to open for writing, which truncates an existing file to zero length or creates a new file.
a	used to append, that is, open for writing at the end of a file, or create a new file.
r+	update reading, which means open for reading and allow writing, positions the file pointer at the beginning of the file.
w+	update writing, which means open for writing and allow reading, truncates an existing file to zero length or creates a new file.
a+	update appending, which means open for writing, positions to the end of the file, and allows for subsequent reads and writes. If the file does not exist, it will be created.

For the update options, fseek or rewind can be used to trigger the change from reading to writing, or vice versa. (Reaching EOF on input will also permit writing without further formality.) Fopen returns a NULL pointer if filename cannot be opened. The update functions are particularly applicable to stream I/O and allow for the possibility of creating temporary files for both reading and writing.

Example:

```
FILE *fp;
char *file;

        if((fp = fopen(file,"r")) == NULL)
                fprintf(stderr, "Cannot open %s0,file);
```

freopen

```
#include <stdio.h>
FILE *freopen (newfile, type, stream)
char *newfile, *type;
FILE *stream;
```

Freopen accepts a pointer, stream , to a previously opened file; the old file is closed, and then the new file is opened. The principal motivation for freopen is the desire to attach the names stdin, stdout, and stderr to specified files. On a successful freopen, the stream pointer is returned; otherwise NULL is returned, indicating that, while the file closing took place, the reopening failed. Freopen is of limited portability; it can not be implemented in all environments.

Example:

```
char *newfile;
FILE *nfile;

        if((nfile = freopen(newfile,"r",stdout)) == NULL)
                fprintf(stderr,"Cannot reopen %s0,newfile);
```

fseek

```
#include <stdio.h>
int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;
```

Fseek positions a stream to a location offset distance from the beginning, current position or end of a file, depending on the values 0, 1, 2 respectively for ptrname . On XENIX the offset unit is bytes; other implementations are not necessarily the same. The return values are 0 on success and EOF on failure. Both buffered and unbuffered files may make use of fseek.

Example:

To position to the end of a file:

```
FILE *stream;

fseek(stream,0L,2);
```

pclose

```
#include <stdio.h>
int pclose (stream)
FILE *stream;
```

Pclose closes a stream opened by popen. It returns the exit status of the command that was issued as the first argument of its corresponding popen, or -1 if the stream was not opened by popen. The function name pclose means an entirely different thing in the OS/370 environment.

popen

```
#include <stdio.h>
FILE *popen (command, type)
char *command, *type;
```

Popen is used to create a pipe between the calling process and a command to be executed. The first argument is a shell command line; type is the I/O mode for the pipe, and may be either r for reading or w for writing. The function returns a stream pointer to be used for I/O on the standard input or output of the command. A NULL pointer is returned if an error occurs.

Example:

```
FILE *pstrm;

if((pstrm=popen("tr mvp MVP","w"))== NULL)
        fprintf(stderr,"popen error0);
fprintf(pstrm,"a message via the pipe...0);
if(pclose(pstrm) == -1)
        fprintf(stderr,"Pclose error0);
```

results in:

```
a Message Via the PiPe
```

rewind

```
#include <stdio.h>
int rewind(stream)
FILE *stream;
```

Rewind sets the position of the next operation at the beginning of the file associated with stream , retaining the current mode of the file. It is the equivalent of fseek (stream,0L,0);.

setbuf

```
#include <stdio.h>
setbuf (stream, buf)
FILE *stream;
char *buf;
```

This function allows the user to choose his own buffer for I/O or to choose to have no buffering at all. Use it after opening and before reading or writing. The function is often used to eliminate the single character writes to a file that result from the execution of putc to standard output that is not redirected. The choice to buffer I/O brings with it the responsibility for flushing any data that may remain in a last, partially-filled buffer. Fflush or fclose perform this task. The constant BUFSIZ in stdio.h tells how big the character array buf is. It is well-chosen for the machine on which UNIX is running. When buf is set to NULL , the I/O is completely unbuffered.

Example:

```
setbuf (stdout, malloc(BUFSIZ));
```

3.2.4 File Status

clearerr

```
#include <stdio.h>
clearerr(stream)
FILE *stream;
```

Clearerr is used to reset the error condition on stream . The need for clearerr arises on XENIX implementations where the error indicator is not reset after a query.

feof

```
#include <stdio.h>
int feof (stream)
```

```
FILE *stream;
```

Feof, which is implemented as a macro on UNIX, returns non-zero if an input operation on stream has reached end of file; otherwise a zero is returned. _Feof_ should be used in conjunction with any I/O function whose return value is not a clear indicator of an end-of-file condition. Such functions are _fread_ and _getw_.

Example:

```
int *x;
FILE *stream;

do
        *x++ = getw(stream);
while(!feof(stream));
```

ferror

```
#include <stdio.h>
int ferror (stream)
FILE *stream;
```

Ferror tests for an indication of error on stream . It returns a non-zero value (true) when an error is found, and a zero otherwise. Calls to _ferror_ do not clear the error condition, hence the _clearerr_ function is needed for that purpose. The user should be aware that, after an error, further use of the file may cause strange results. On XENIX _ferror_ is implemented as a macro.

Example:

```
FILE *stream;
int *x;

while(!ferror(stream))
        putw(*x++,stream);
```

ftell

```
#include <stdio.h>
long ftell (stream)
FILE *stream;
```

Ftell is used to determine the current offset relative to the beginning of the file associated with stream . It returns the current value of the offset; in XENIX it returns the offset value in bytes. On error, a value of -1 is returned. This function is useful in obtaining an offset for subsequent fseek calls.

3.2.5 Input Function

fgetc

```
#include <stdio.h>
int fgetc (stream)
FILE *stream;
```

This is the function version of the macro getc and acts identically to getc. Because fgetc is a function and not a macro,it can be used in debugging to set breakpoints on fgetc and when the side effects of macro processing of the argument is a problem. Furthermore, it can be passed as an argument.

fgets

```
#include <stdio.h>
char *fgets (s,n,stream)
char *s;
int n;
FILE *stream;
```

Fgets reads from stream into the area pointed to by s either n-1 characters or an entire string including its new-line terminator, whichever comes first. A final null character is affixed to the data read. It returns the pointer s on success, and NULL on end-of-file or error. Fgets differs from the function gets in that it can read from other than stdin, and that it appends the new-line at the end of input when the size of the string is longer than or equal to n . More importantly, it provides control over the size of the string to be read that is not available with gets.

Example:

```
char msg[MAX];
FILE *myfile;

        while(fgets(msg,MAX,myfile) != NULL)
                printf("%s0,msg);
```

fread

```
#include <stdio.h>
int fread((char *)ptr, sizeof (*ptr), nitems, stream)
FILE *stream;
```

This function reads from stream the next nitems whose size is the same as the size of the item pointed to by ptr , into a sufficiently large area starting at ptr . It returns the number of items read. In XENIX, fread makes use of the function getc. It is often used in combination with feof and ferror to obtain a clear indication of the file status.

Example:

```
FILE *pstm;
char mesg[100];

        while(fread((char *)mesg,sizeof(*mesg),1,pstm) ==
                printf("%s0,mesg);
```

fscanf

```
#include <stdio.h>
int fscanf (stream, format[, argptr]...)
char *format;
FILE *stream;
```

Fscanf accepts input from the file associated with stream , and deposits it into the storage area pointed to by the respective argument pointers according to the specified formats. Format specifications are those that appear in Attachment D. Fscanf differs from scanf in that it can read from other than stdin. The function returns the number of successfully handled input arguments, or EOF on end-of-input.

Example:

```
FILE *file;
long pay;
char name[15];
char pan[7];

        fscanf(file,"%6s%14s%ld0,pan,name,&pay);
        if(pay<50000)
                printf("$%ld raise for %s.0,pay/10,name);
```

If the input data is:

```
020202MaryJones 15000
```

the resulting output is:

```
$1500 raise for MaryJones.
```

getc

```
#include <stdio.h>
int getc (stream)
FILE *stream;
```

Getc returns the next character from the named stream . It is implemented as a macro to avoid the overhead of a function call. On error or end-of-file it returns an EOF . _Fgetc_ should be used when it is necessary to avoid the side effects of argument processing by the macro _getc_.

getchar

```
#include <stdio.h>
int getchar()
```

This is identical to _getc_ (_stdin_).

gets

```
#include <stdio.h>
char *gets(s)
char *s;
```

Gets reads a string of characters up to a new-line from _stdin_ and places them in the area pointed to by s . The new-line character which ended the string is replaced by the null character. The return values are s on success, NULL on error or end-of-file. The simple example below presumes the size of the string read into msg will not exceed SIZE in length. If used in conjunction with _strlen_, a dangerous overflow can be detected, though not prevented.

Example:

```
char msg[SIZE];
char *s;
        s = msg;
        while (gets(s) != NULL)
                printf("%s0,s);
```

getw

```
#include <stdio.h>
int getw (stream)
FILE *stream;
```

Getw reads the next word from the file associated with stream . On success it returns the word; on error or end of file, it returns EOF . However, because EOF could be a valid word, this function is best used with feof and ferror.

Example:

```
FILE *stream;
int *x;
        do
                *x++ = getw(stream);
        while (!feof(stream));
```

scanf

```
#include <stdio.h>
int scanf (format[, argptr]...)
char *format;
```

Scanf reads input from stdin, delivers the input according to the specified formats, and deposits the input in the storage area pointed to by the respective argument pointers. The correct format specifications can be found in Attachment D. For input from other streams than stdin use fscanf; for input from a character array use sscanf. The return values are the number of successfully handled input arguments, or EOF on end-of-input.

Example:

```
long number;

        scanf("%ld",&number);
        (printf(number%2?"%ld is odd":"%ld is even",number)
```

sscanf

```
#include <stdio.h>
sscanf (s, format [, pointer]...)
char *s;
char *format;
```

Sscanf accepts input from a character string s , delivers the input according to the specified formats, and deposits it into the storage area pointed to by the respective argument pointers. Format specifications appear in Attachment D. This function returns the number of successfully handled input arguments.

Example:

```
char datestr[] = {"THU MAR 29 11:04:40 EST 1979"};
char month[4];
char year[5];

        sscanf(datestr,"%*3s%3s%*2s%*8s%*3s%4s",month,year);
        printf("%s, %s0,month,year);
```

The result is:

```
MAR, 1979
```

ungetc

```
#include <stdio.h>
int ungetc (c, stream)
int c;
FILE *stream;
```

Ungetc puts the character c back on the file associated with stream . One character (but never EOF) is assured of being put back. If successful, the function returns c , otherwise EOF .

Example:

```
while(isspace (c = getc(stdin)))
        ;
ungetc(c,stdin);
```

This code puts the first character that is not white space back onto the standard input stream.

3.2.6 Output Functions

fflush

```
#include <stdio.h>
int fflush (stream)
FILE *stream;
```

Fflush takes action to guarantee that any data contained in file buffers and not yet written out will be written. It is used by fclose to flush a stream. No action is taken on files not open for writing. The return values are zero for success, EOF on error.

fprintf

```
#include <stdio.h>
int fprintf (stream, format[, arg ]...)
FILE *stream;
char *format;
```

Fprintf provides formatted output to a named stream. The function printf may be used if the destination is stdout. Specifications for formats are available in Attachment C. On error, fprintf returns non-zero, otherwise zero. In later releases of the C library, fprintf will return the number of characters transmitted, or a negative value on error.

Example:

```
int *filename;
int c;

        if(c==EOF)
                fprintf(stderr,"EOF on %s0,filename);
```

fputc

```
#include <stdio.h>
int fputc (c,stream)
int c;
FILE *stream;
```

Fputc performs the same task as putc; that is, it writes the character c to the file associated with stream , but is implemented as a function rather than a macro. It is preferred to putc when the side effects of macro processing of arguments are a problem. On success, it returns the character written; on failure it returns EOF .

Example:

```
FILE *in, *out;
int c;

        while ((c = fgetc(in)) != EOF)
                fputc(c,out);
```

fputs

```
#include <stdio.h>
int fputs(s,stream)
char *s;
FILE *stream;
```

Fputs copies a string to the output file associated with stream . It uses the function putc to do this. It is different from puts in two ways: it allows any output stream to be specified, and it does not affix a new-line to the output. For an example, see puts.

fwrite

```
#include <stdio.h>
int fwrite ((char *)ptr, sizeof (*ptr),nitems,stream)
FILE *stream;
```

Beginning at ptr , this function writes up to nitems of data of the type pointed to by ptr into output stream . It returns the number of items actually written. Like fread this function should be used in conjunction with ferror to detect the error condition.

Example:

```
char mesg[] ={"My message to write out0};
FILE *pstrm;

        if(fwrite(mesg,(sizeof(*mesg)-1),1,pstrm) != 1)
                fprintf(stderr,"Output error0);
```

printf

```
#include <stdio.h>
int printf(format[, arg]...)
char *format;
```

Printf provides formatted output on stdout. The many format specifications are available in Attachment C. Fprintf and sprintf are related functions that write output onto other than the standard output device. In case of error, implementations are not consistent in their output. On error, printf returns non-zero, otherwise zero. In later releases of the C library, printf returns the number of characters transmitted, or a negative value on error.

Example:

```
int num = 10;
char msg[] = {"ten"};
printf("%d - %o - %s0, num, num, msg);
```

results in the line:

```
10 - 12 - ten;
```

putc

```
#include <stdio.h>
int putc (c,stream)
int c;
FILE *stream;
```

Putc writes the character c to the file associated with stream. On success, it returns the character written; on error it returns EOF . Because it is implemented as a macro, side effects may result from argument processing. In such cases, the equivalent function fputc should be used.

Example:

```
#define PROMPT()         putc('7',stderr)          /* BEL */
```

putchar

```
#include <stdio.h>
int putchar(c)
int c;
```

Putchar is defined as putc (c, stdout). It returns the character written on success, or EOF on error.

Example:

```
char *cp;
char x[SIZE];

        for(cp=x;cp<(x+SIZE);cp++)
                putchar(*cp);
```

puts

```
#include <stdio.h>
int puts(s)
char *s;
```

The function copies the string pointed to by s without its terminating null character to stdout.

A new-line character is appended. XENIX uses the macro <u>putchar</u> (which calls <u>putc</u>).

<u>Example</u>:

```
puts("I will append a new-line");
fputs(" some more data ", stdout);
puts("and now a new-line");
```

The resulting output is:

```
I will append a new-line
        some more data and now a new-line
```

putw

```
#include <stdio.h>
int putw(w,stream)
FILE *stream;
int w;
```

<u>Putw</u> appends word w to the output stream . As with <u>getw</u>, the proper way to check for an error or end-of-file is to use the <u>feof</u> and <u>ferror</u> functions.

<u>Example</u>:

```
int info;

while(!feof(stream))
        putw(info,stream);
```

sprintf

```
#include <stdio.h>
int sprintf(s, format, [, arg]...)
char *s;
char *format;
```

<u>Sprintf</u> allows for formatted output to be placed in a character array pointed to by s . <u>Sprintf</u> adds a null at the end of the formatted output. See Attachment C for the specification of formats. It is the user's responsibility to provide an array of sufficient length. Other related functions <u>printf</u> and <u>fprintf</u> handle similar kinds of formatted output. <u>Sprintf</u> can be used to build formatted arrays in memory, to be changed dynamically before output, or to be used to call other routines. The comparable input function is <u>sscanf</u>. On error, <u>sprintf</u> returns non-zero, otherwise zero. In later

releases of the C library, sprintf returns the number of characters transmitted, or a negative value on error.

Example:

```
char cmd[100];
char *doc = "/usr/src/cmd/cp.c"
int width = 50;
int length = 60;

        sprintf(cmd,"pr -w%d -l%d %s0,width,length,doc);
        system(cmd);
```

The above code executes the pr command to print the source of the cp command.

3.2.7 String Functions

strcat

```
char *strcat(dst,src)
char *dst, *src;
```

Strcat appends characters in the string pointed to by src to the end of the string pointed to by dst , and places a null character after the last character copied. It returns a pointer to dst . To concatenate strings up to a maximum number of characters, use strncat.

Example:

```
char *myfile;
char dir[L_cuserid+5] = "/usr/";
        myfile = (strcat(dir,cuserid(0)));
```

The result is the concatenation of the login name onto the end of the string dir .

strcmp

```
char *strcmp(sl,s2)
char *sl, *s2;
```

Strcmp compares the characters in the string sl and s2 . It returns an integer value, greater than, equal to, or less than zero, depending on whether sl is lexicographically greater than, equal to, or less than s2 .

Example:

```
#define EQ(x,y)  !strcmp(x,y)
```

strcpy

```
char *strcpy(dst, src)
char *dst, *src;
```

Strcpy copies the characters (including the null terminator) from the string pointed to by src into the string pointed to by dst . A pointer to dst is returned.

Example:

```
char dst[] = "UPPER CASE";
char src[] = "this is lower case";

        printf("%s0,strcpy(dst,src+8));
```

results in:

```
lower case
```

strlen

```
int strlen(s)
char *s;
```

Strlen counts the number of characters starting at the character pointed to by s up to, but not including, the first null character. It returns the integer count.

Example:

```
char nextitem[SIZE];
char series[MAX];

if(strlen(series)) strcat(series,",");
strcat(series,nextitem);
```

strncat

```
char *strncat(dst, src, n)
char *dst, *src;
int n;
```

Strncat appends a maximum of n characters of the string pointed to by src and then a null character to the string pointed to by dst . It returns a pointer to dst .

Example:

```
char dst[] = "cover";
char src[] = "letter";

        printf("%s0,strncat(dst,src,3));
```

The output is:

```
coverlet
```

strncmp

```
int strncmp(sl,s2,n)
char *sl, *s2;
int n;
```

Strncmp compares two strings for at most n characters and returns an integer greater than, equal to, or less than zero as sl is lexicographically greater than, equal to or less than s2 .

Example:

```
char filename [] = "/dev/ttyx";

if(strncmp (filename+5, "tty",3) == 0)
        printf("success0);
```

strncpy

```
char *strncpy(dst,src,n)
char *dst, *src;
int n;
```

Strncpy copies n characters of the string pointed to by src into the string pointed to by dst . Null padding or truncation of src occurs as necessary. A pointer to dst is returned.

Example:

```
char buf [MAX];
char date [29] = {"Fri Jun 29 09:35:44 EDT 1979"};
char *day = buf;

        strncpy(day,date,3);
```

After executing this code, day points to the string Fri .

3.2.8 Character Classification

isalnum

```
#include <ctype.h>
int isalnum(c)
int c;
```

This macro determines whether or not the character c is an alphanumeric character ([A-Za-z0-9]). It returns zero for false and non-zero for true.

isalpha

```
#include <ctype.h>
int isalpha(c)
int c;
```

This macro determines whether or not the character c is an alphabetic character ([A-Za-z]). It returns zero for false and non-zero for true.

isascii

```
#include <ctype.h>
int isascii(c)
int c;
```

This macro determines whether or not the integer value supplied is an ASCII character; that is, a character whose octal value ranges from 000 to 177. It returns zero for false and non-zero for true.

iscntrl

```
#include <ctype.h>
int iscntrl(c
int c;
```

This macro determines whether or not the character c when mapped to ASCII is a control character (that is, octal 177 or 000-037). It returns zero for false and non-zero for true.

isdigit

```
#include <ctype.h>
int isdigit(c)
int c;
```

This macro determines whether or not the character c is a digit. It returns zero for false and non-zero for true. (that is, is an

ASCII code between octal 041 and 176 inclusive).

islower

```
#include <ctype.h>
int islower(c)
int c;
```

This macro determines whether or not the character c is a lower-case letter. It returns zero for false and non-zero for true.

isprint

```
#include <ctype.h>
int isprint(c)
int c;
```

This macro determines whether or not the character c is a printable character. (This includes spaces.) It returns zero for false and non-zero for true.

ispunct

```
#include <ctype.h>
int ispunct(c)
int c;
```

This macro determines whether or not the character c is a punctuation character (neither a control character nor an alphanumeric). It returns zero for false and non-zero for true.

isspace

```
#include <ctype.h>
int isspace(c)
int c;
```

This macro determines whether or not the character c is a form of white space (that is, a blank, horizontal or vertical tab, carriage return, form-feed or new-line). It returns zero for false and non-zero for true.

isupper

```
#include <ctype.h>
int isupper(c)
int c;
```

This macro determines whether or not the character c is an upper-case letter. It returns zero for false and non-zero for true.

3.2.9 Character Translation

toascii

```
#include <ctype.h>
int toascii (c)
int c;
```

The macro toascii usually does nothing: its purpose is to map the input character into its ASCII equivalent.

Example:

```
FILE *oddstrm;

        if(!isdigit (toascii(getw(oddstrm))))
                fprintf(stderr,"bad data0);
```

tolower

```
#include <ctype.h>
int tolower (c)
int c;
```

If the argument c passed to the function tolower is an upper-case letter, the lower-case representation of c is returned, otherwise c is returned unchanged. For a faster routine, use _tolower, which is implemented as a macro; however, the argument must already be an upper-case letter.

Example:

```
if(tolower(getchar()) != 'y')
        exit(0);
```

toupper

```
#include <ctype.h>
int toupper (c)
int c;
```

If the argument c passed to the function toupper is a lower-case letter, the upper-case representation of c is returned, otherwise c is returned unchanged. For a faster routine, use _toupper, however, the argument must already be a lower-case letter.

Example:

```
if(toupper (getchar()) != 'Y')
        exit(0);
```

3.2.10 Space Allocation

calloc

```
char *calloc(n, size)
unsigned n, size;
```

Calloc allocates enough storage for an array of n items aligned for any use, each of size bytes. The space is initialized to zero. Calloc returns a pointer to the beginning of the allocated space, or a NULL pointer on failure.

Example:

```
char *t;
int n;
unsigned size;

        if(t=calloc((unsigned)n, size) == NULL)
                fprintf(stderr,"Out of space.0);
```

free

```
free(ptr)
char *ptr;
```

Free is used in conjunction with the space allocating functions malloc, calloc, or realloc. Ptr is a pointer supplied by one of these routines. The effect is to free the space previously allocated.

malloc

```
char *malloc(size)
unsigned size;
```

Malloc allocates size bytes of storage beginning on a word boundary. It returns a pointer to the beginning of the allocated space, or a NULL pointer on failure to acquire space. For space initialized to zero, see calloc.

Example:

```
int n;
char *t;
unsigned size;

        if(t=malloc((unsigned)n) == NULL)
                fprintf(stderr,"Out of space.0);
```

realloc

```
char *realloc (ptr, size)
char *ptr;
unsigned size;
```

Given ptr which was supplied by a call to _malloc_ or _calloc_, and a new byte size, size , _realloc_ returns a pointer to the block of space of size bytes. This function is useful to do storage compacting along with _malloc_ and _free_.

The following pages contain the contents of the three most important include files: ctype.h, stdio.h, and signal.s. These files are well worth some study, just to see how these all these definitions help to create a powerful interface to the internals of the XENIX system.

3.2.10.1 ctype.h

```
#define _U      01
#define _L      02
#define _N      04
#define _S     010
#define _P     020
#define _C     040
#define _B    0100

extern  char    _ctype_[];

#define isalpha(c)      ((_ctype_+1)[c]&(_U|_L))
#define isupper(c)      ((_ctype_+1)[c]&_U)
#define islower(c)      ((_ctype_+1)[c]&_L)
#define isdigit(c)      ((_ctype_+1)[c]&_N)
#define isspace(c)      ((_ctype_+1)[c]&(_S|_B))
#define ispunct(c)      ((_ctype_+1)[c]&_P)
#define isalnum(c)      ((_ctype_+1)[c]&(_U|_L|_N))
#define isprint(c)      ((_ctype_+1)[c]&(_P|_U|_L|_N|_B))
#define iscntrl(c)      ((_ctype_+1)[c]&_C)
#define isascii(c)      ((unsigned)(c)<=0177)
#define _toupper(c)     ((c)-'a'+'A')
#define _tolower(c)     ((c)-'A'+'a')
#define toascii(c)      ((c)&0177)
```

3.2.10.2 signal.h

```
#define     NSIG        16

#define     SIGHUP       1    /* hangup */
#define     SIGINT       2    /* interrupt */
#define     SIGQUIT      3    /* quit */
#define     SIGILL       4    /* illegal instruction (not reset when caugh
#define     SIGTRAP      5    /* trace trap (not reset when caught) */
#define     SIGIOT       6    /* IOT instruction */
#define     SIGEMT       7    /* EMT instruction */
#define     SIGFPE       8    /* floating point exception */
#define     SIGKILL      9    /* kill (cannot be caught or ignored) */
#define     SIGBUS      10    /* bus error */
#define     SIGSEGV     11    /* segmentation violation */
#define     SIGSYS      12    /* bad argument to system call */
#define     SIGPIPE     13    /* write on a pipe with no one to read it */
#define     SIGALRM     14    /* alarm clock */
#define     SIGTERM     15    /* software termination signal from kill */

int         (*signal())();
#define     SIG_DFL     (int (*)())0
#define     SIG_IGN     (int (*)())1
```

3.2.10.3 stdio.h

```
#define         BUFSIZ          512
#define         _NFILE           20
# ifndef FILE
extern  struct  _iobuf {
        char    *_ptr;
        int     _cnt;
        char    *_base;
        char    _flag;
        char    _file;
} _iob[_NFILE];
# endif

#define         _IOREAD          01
#define         _IOWRT           02
#define         _IONBF           04
#define         _IOMYBUF        010
#define         _IOEOF          020
#define         _IOERR          040
#define         _IOSTRG        0100
#define         _IORW          0200

#define         NULL              0
#define         FILE           struct  _iobuf
#define         EOF            (-1)

#define         L_ctermid         9
#define         L_cuserid         9
#define         L_tmpnam         19

#define         stdin          (&_iob[0])
#define         stdout         (&_iob[1])
#define         stderr         (&_iob[2])
#define         getc(p)        (--(p)->_cnt>=0?
#define         getchar()      getc(stdin)
#define         putc(x,p)      (--(p)->_cnt>=0?
_flsbuf((unsigned)(x),p))
#define         putchar(x)     putc(x,stdout)
#define         feof(p)        (((p)->_flag&_IOEOF)!=0)
#define         ferror(p)      (((p)->_flag&_IOERR)!=0)
#define         fileno(p)      p->_file

FILE    *fopen();
FILE    *freopen();
FILE    *fdopen();
long    ftell();
char    *fgets();
```

3.3 THE XENIX ASSEMBLY LANGUAGE INTERFACE

The XENIX system is designed so that there should be little need to program in assembly language. Occasionally, however, the need does arise, and you may need to know the conventions for storing words in memory, for accessing parameters on the stack in a way compatible with the C runtime environment. Remember, however, that programming in assembly language is highly machine dependent, and that you sacrifice portability whenever you forsake C for whatever low-level advantages you might gain.

3.3.1 Memory Format

With the 8086, words are stored as followed:

```
<addr+1>        <high order byte>
<addr+0>        <low  order byte>
```

The words of a long are stored 'backwards' to this convention, the high order word comes first in memory:

```
<addr+3>        <high order byte of low  order word>
<addr+2>        <low  order byte of low  order word>
<addr+1>        <high order byte of high order word>
<addr+0>        <low  order byte of high order word>
```

The floating point format is currently Microsoft format, but will definitely change to an IEEE compatible format in the future.

Pascal 32-bit integers are stored as follows:

```
<addr+3>        <high order byte of low  order word>
<addr+2>        <low  order byte of low  order word>
<addr+1>        <high order byte of high order word>
<addr+0>        <low  order byte of high order word>
```

3.3.2 Calling Sequence

Arguments are pushed last first, and are in fact evaluated in that order. In C, the order of evaluation of arguments is undefined, Arguments are pushed by value, in a choice of 4 sizes: chars, ints, and unsigned ints are pushed in one 16-bit word. Longs are pushed as two 16-bit words, low order word first so the order in memory is preserved. Floats and doubles as four 16-bit words, again order preserving. Note that chars and floats are extended to the size of int or double respectively. Structures, which are allocated rounded up to the next even byte size, are pushed so that their memory order is preserved. This means that the last word is pushed first.

3.3.3 Procedure Entry and Exit

Th bp, sp, si, and di registers must be restored upon procedure exit if they have been modified. The following sequence does this, and is what the compiler uses:

```
     entry:
             push    bp
             mov     sp,bp
             push    di
             push    si

             <body>

     return:
             jmp     cret      | cret cleans up,
                               | including any modifications
                               | that may have been made to sp.
                               | Ax, bx, cx, and dx
                               | are preserved, as
                               | well as segment registers.
                               | Flags are not preserved.
                               | Cret does a ret instruction,
                               | so there is no need
                               | for the user to do it.
```

Note that with this mechanism, the first argument (the last pushed) will be at 4(bp), with subsequent words at 6(bp), 8(bp) and so forth. Where the various arguments are is based on the size of arguments pushed.

We recommend that this sequence always be used, even if the registers SI and DI will not be modified. Use of this sequence allows backtracing by ADB in the case of a program crash.

3.3.4 Return Values

Int and char return values are left in the ax register. Long return values are left in ax-dx, high order in dx. (Note that this corresponds to what the cwl instruction does, so it should be easy to remember.) Structures are returned by having ax point to a static area of memory, which contains the return value; floats are returned the same way.

3.3.5 System calls

In order to issue system calls, it is necessary for the user to use the library functions discussed in chapter 3. Assembly language programmers need to make a proper C-compatible call to these routines, as shown above.

CHAPTER 4

OTHER TOOLS

This chapter discusses other tools and languages available to the software developer. These tools and languages can be used to complement the basic tools of chapter 4 or they can in some instances be substituted for them.

The tools described here include a macro processor called _m4_, a lexical analyzer named _lex_, and a compiler of compilers named _YACC_. _Lex_ and _YACC_ have been used to create a number of compilers, and _m4_ has been field tested as the front end to a variety of processors.

The languages discussed in this chapter are the calculating languages _dc_ and _bc_. These languages can be used to perform reasonably complex mathematical operations with a high degree of precision. These languages are similar to the languages understood by hand-held calculators.

4.1 The M4 Macro Processor

A macro processor is a useful way to enhance a programming language, to make it more palatable or more readable, or to tailor it to a particular application. The _#define_ statement in C and the analogous _define_ in Ratfor are examples of the basic facility provided by any macro processor -- replacement of text by other text.

M4 is a suitable front end for Ratfor and C, and has also been used successfully with Cobol. Besides the straightforward replacement of one string of text by another, it provides macros with arguments, conditional macro expansion, arithmetic, file manipulation, and some specialized string processing functions.

The basic operation of M4 is to copy its input to its output. As the input is read, however, each alphanumeric ``token'' (that is, string of letters and digits) is checked. If it is the name of a macro, then the name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before it is rescanned.

M4 provides a collection of about twenty built-in macros which perform various useful operations; in addition, the user can define new macros. Built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

4.1.1 _Usage_

To invoke M4, type:

```
     m4 [files]
```

Each argument file is processed in order. If there are no arguments, or if an argument is `-', the standard input is read at that point. The processed text is written on the standard output, which may be captured for subsequent processing with

```
     m4 [files] >outputfile
```

4.1.2 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 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 `(', it is assumed to have no arguments. This is the situation for _N_ above; it is actually a macro with no arguments, and thus when it is used there need be no (...) following it.

You should also notice that a macro name is only recognized as such if it appears surrounded by non-alphanumerics. For example, in

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

the variable _NNN_ is absolutely unrelated to the defined macro _N_, even though it contains a lot of _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'll always get the value of N at that time (because the M will be replaced by N which will be replaced by 100).

4.1.3 Quoting

The more general solution is to delay the expansion of the arguments of define by quoting them. Any text surrounded by the single quotes ` and ' is not expanded immediately, but has the quotes stripped off. If you say

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

the quotes around the N are stripped off as the argument is being collected, but they have served their purpose, and M is defined as the string N, not 100. The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If 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 ` and ' are not convenient for some reason, the quote characters can be changed with the built-in _changequote_. For example:

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

makes the new quote characters the left and right brackets. You can restore the original characters with just

```
changequote
```

There are two additional built-ins related to _define_. _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 quotes 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)
```

4.1.4 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_, although that is less commonly used.) Arguments that are not supplied are replaced by null strings, so we can define a macro _cat_ which simply concatenates its arguments, like this:

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

Thus

```
     cat(x, y, z)
```

is equivalent to

```
     xyz
```

$_4_ through $_9_ are null, since no corresponding arguments were provided.

Leading unquoted 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.

4.1.5 Arithmetic Built-ins

M4 provides two built-in functions for doing arithmetic on integers (only). The simplest is incr, which increments its numeric argument by 1. Thus, to handle the common programming situation where you want a variable to be defined as ``one more than N'', write

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

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
es the 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.

4.1.6 File Manipulation

You can include a new file in the input at any time by the built-in function include:

```
     include(filename)
```

inserts the contents of filename in place of the include command. The contents of the file is often a set of definitions. The value of include (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in include cannot be accessed. To get some control over this situation, the alternate form sinclude can be used; sinclude (``silent include'') says nothing and continues if it can't access the file.

It is also possible to divert the output of M4 to temporary files during processing, and output the collected material upon command. M4 maintains nine of these diversions, numbered 1 through 9. If you say

```
     divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as n. Diverting to this file is stopped by another divert command; in particular, divert or divert(0) resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

```
     undivert
```

brings back all diversions in numeric order, and _undivert_ with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of _undivert_ is _not_ the diverted 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.

4.1.7 System Command

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.

4.1.8 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 quotes, which prevent too-early evaluation of _ifelse_.

If the fourth argument is missing, it is treated as empty.

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

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

if the string _a_ matches the string _b_, the result is _c_. Otherwise, if _d_ is the same as _e_, the result is _f_. Otherwise the result is _g_. If the final argument is omitted, the result is null, so

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

is _c_ if _a_ matches _b_, and null otherwise.

4.1.9 String Manipulation

The built-in _len_ returns the length of the string that makes up its argument. Thus

```
     len(abcdef)
```

is 6, and _len_((_a_,_b_)) is 5.

The built-in _substr_ can be used to produce substrings of strings. _substr_(_s_, _i_, _n_) returns the substring of _s_ that starts at the _i_th position (origin zero), and is _n_ characters long. If _n_ is omitted, the rest of the string is returned, so

```
     substr(`now is the time', 1)
```

is

```
     ow is the time
```

If _i_ or _n_ are out of range, various sensible things happen.

index(_s1_, _s2_) returns the index (position) in _s1_ where the string _s2_ occurs, or -1 if it doesn't occur. As with _substr_, the origin for strings is 0.

The built-in _translit_ performs character transliteration.

```
     translit(s, f, t)
```

modifies _s_ by replacing any character found in _f_ by the corresponding character of _t_. That is,

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

replaces the vowels by the corresponding digits. If _t_ is shorter than _f_, characters which don't have an entry in _t_ are deleted; as a limiting case, if _t_ is not present at all, characters 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
```

4.1.10 _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 which dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget the quotes.

4.1.11 Summary of Built-ins

```
changequote(L, R)
define(name, replacement)
divert(number)
divnum
dnl
dumpdef(`name', `name', ...)
errprint(s, s, ...)
eval(numeric expression)
ifdef(`name', this if true, this if false)
ifelse(a, b, c, d)
include(file)
incr(number)
index(s1, s2)
len(string)
maketemp(...XXXXX...)
sinclude(file)
substr(string, position, number)
syscmd(s)
translit(str, from, to)
undefine(`name')
undivert(number,number,...)
```

4.2 Lex

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial look%ahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. Lex is designed to simplify interfacing with Yacc, the XENIX compiler-compiler.

4.2.1 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 program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the bound%aries 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 beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called ``host languages.'' Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C.

Lex turns the user's expressions and actions (called <u>source</u> in this section) into the host general-purpose language; the generated program is named <u>yylex</u>. The <u>yylex</u> program will recognize expressions in a stream (called <u>input</u> here) and perform the specified actions for each expression as it is detected.

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

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

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C 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 ...''; and the $ indicates ``end of line.'' No action is specified, so the program generated by

Lex (yylex) 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 will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule 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 particularly easy to interface Lex and Yacc. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. 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 <u>yylex</u> 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 re%scanning. 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 <u>actions</u> 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 which can be interpreted on the basis of one character look%ahead. For example, if there are two rules, one looking for ab and another for abcdefg, and the input stream is abcdefh, Lex will recognize ab and leave the input pointer just before cd. . Such backup is more costly than the processing of simpler languages.

4.2.2 Lex Source

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 which copies the input to the output unchanged.

In the outline of Lex programs shown above, the rules represent the user's control decisions; they are 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 an individual rule might appear

```
     integerprintf("found keyword INT");
```

to look for the string integer in the input stream and print the message ``found keyword INT'' whenever it appears. In this example the host procedural language is C and the C library function printf is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
     colourprintf("color");
     mechaniseprintf("mechanize");
     petrolprintf("gas");
```

would be a start. These rules are not quite enough, since the word petroleum would become gaseum; a way of dealing with this will be described later.

4.2.3 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.

Operators. The operator characters are

```
     " \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
     xyz"++"
```

matches the string xyz++ when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
     "xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

<u>Character classes</u>. Classes of characters can be specified using the operator pair []. The construction [<u>abc</u>] matches a single character, which may be <u>a</u>, <u>b</u>, or <u>c</u>. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ - and ^. The - character indicates ranges. For example,

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

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using - between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0-z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character - in a character class, it should be first or last; thus

```
[-+0-9]
```

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the 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 \ character provides the usual escapes within character class brackets.

Arbitrary character. To match almost any character, the operator character

```
.
```

is the class of all characters except newline. Escaping into octal is possible although non-portable:

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional expressions. The operator ? indicates an optional element of an expression. Thus

```
ab?c
```

matches either _ac_ or _abc_.

Repeated expressions. Repetitions of classes are indicated by the operators * and +.

```
a*
```

is any number of consecutive _a_ characters, including zero; while

```
a+
```

is one or more instances of _a_. For example,

```
[a-z]+
```

is all strings of lower case letters. And

```
[A-Za-z][A-Za-z0-9]*
```

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping. The operator | indicates alternation:

```
     (ab|cd)
```

matches either ab or cd. Note that parentheses are used for grouping, although they are not necessary on the outside level;

```
     ab|cd
```

would have sufficed. Parentheses can be used for more complex expressions:

```
     (ab|cd+)?(ef)*
```

matches such strings as abefef, efefef, cdef, or cddd; but not abc, abcd, or abcdef.

Context sensitivity. Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be 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 ^, complementation of character classes, since that only applies within the [] operators. If the very last character is $, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

```
     ab/cd
```

matches the string ab, but only if followed by cd. Thus

```
     ab$
```

is the same as

```
     ab/\n
```

Left context is handled in Lex by start conditions as explained in section 10. If a rule is only to be executed when the Lex automaton interpreter is in start condition x, the rule should be prefixed by

```
     <x>
```

using the angle bracket operator characters. If we considered ``being at the beginning of a line'' to be start condition ONE, then the ^ operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

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

```
{digit}
```

looks for a predefined string named digit and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

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

looks for 1 to 5 occurrences of a.

Finally, initial % is special, being the separator for Lex source segments.

4.2.4 Lex Actions

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex 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. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, ; as an action 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 the 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 different style. The quotes around \n and \t are not required.

In more complex actions, the user will 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);
```

will print the string in _yytext_. The C function _printf_ accepts a format argument and data to be printed; in this case, the format is ``print string'' (% indicating data conversion, and _s_ indicating string type), and the data are the characters in _yytext_. So this just places the matched string on the output. This action is so common that it may be written as ECHO:

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

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches _read_ it 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 _yyleng_ of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

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

which accumulates in _chars_ the number of characters in the words recognized. The last character in the string matched can be accessed by

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

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, yymore() can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in yytext. Second, yyless (n) may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument n indicates the number of characters in yytext to be retained. Further characters previously matched are returned to the input. This provides the same sort of look%ahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

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

which will, when faced with a string such as "abc\"def" first match the five characters "abc\; then the call to yymore() will cause the next part of the string, "def, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled ``normal processing''.

The function yyless() might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of ``=-a''. Suppose it is desired to treat this as ``=- a'' but print a message. A rule might be

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

which prints a message, returns the letter after the operator to the input stream, and treats the operator as ``=-''. Alternatively it might be desired to treat this as

``= -a''. To do this, just return the minus sign as well as the letter to the input:

```
=-[a-zA-Z]{
  printf("Operator (=-) ambiguous\n");
  yyless(yyleng-2);
  ... action for = ...
  }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
=-/[A-Za-z]
```

in the first case and

```
=/-[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of ``=-3'', however, makes

```
=-/[^ \t\n]
```

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

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

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

3. unput(c) pushes the character c back onto the input stream to be read later by input().

By default these routines are provided as macro definitions, but the 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 Lex look%ahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in + * ? or $ or containing / implies look%ahead. Look%ahead 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 the user will sometimes want to redefine is yywrap() which is called whenever Lex reaches an end-of-file. If yywrap returns a 1, Lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a yywrap which arranges for new input and returns 0. This instructs Lex to continue processing. The default yywrap always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through yywrap. In fact, unless a private version of input() is supplied a file containing nulls cannot be handled, since a value of 0 returned by input is taken to be end-of-file.

4.2.5 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 which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integerkeyword action ...;
[a-z]+identifier action ...;
```

to be given in that order. If the input is integers, it is taken as an identifier, because [a-z]+ matches 8 characters while integer matches only 7. If the input is integer, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. int) will not match the expression integer and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example,

```
     '.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

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

the above expression will match

```
     '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, will stop after 'first'. The consequences of errors like this are mitigated by the fact that the . operator will not match newline. Thus expressions like .* stop on the current line. Don't try to defeat this with expressions like [.\n]+ or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both she and he in an input text. Some Lex rules to do this might be

```
     shes++;
     heh++;
     \n|
     . ;
```

where the last two rules ignore everything besides he and she. Remember that . does not include newline. Since she includes he, Lex will normally not recognize the instances of he included in she, since once it has passed a she those characters are gone.

Sometimes the 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 a priori to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+{ ... ; REJECT;}
a[cd]+{ ... ; REJECT;}
```

If the input is ab, only the first rule matches, and on ad only the second matches. The input string accb matches the first rule for four characters and then the second rule for three characters. In contrast, the input accd agrees with the second rule for four characters and 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.

4.2.6 Lex Source Definitions

Remember the format of the Lex source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by Lex. These can go either in the definitions section or in the rules section.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

1. Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter 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 which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2. Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3. Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

   Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and begining 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]+/"."EQprintf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs. These possibilities are discussed below under ``Summary of Source Format.''

4.2.7 Usage

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

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

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

The resulting program is placed on the usual file a.out for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so. If private versions of input, output and unput are given, the library can be avoided.

4.2.8 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 -lln
```

The Yacc library (-ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

4.2.9 Examples

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

```
%%
  int k;
[0-9]+{
  k = atoi(yytext);
  if (k%7 == 0)
       printf("%d", k+3);
  else
       printf("%d",k);
  }
```

to do just that. The rule [0-9]+ recognizes strings of digits; atoi converts the digits to binary and stores the result in k. The operator % (remainder) is used to check whether k is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as 49.63 or X7. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as 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 ``.'' 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 histograms the lengths of words, 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 by N. L. Schryer to convert double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```
a [aA]
b [bB]
c [cC]
...
z [zZ]
```

An additional class recognizes white space:

```
W [ \t]*
```

The first rule changes ``double precision'' to ``real'', or ``DOUBLE PRECISION'' to ``REAL''.

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
     printf(yytext[0]=='d'? "real" : "REAL");
     }
```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword.

The next rule copies continuation card indications to avoid confusing them with constants:

```
^"     "[^ 0]ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as ``beginning of line, then five blanks, then anything but blank or zero.'' Note the two different meanings of ^. There follow some rules to change double precision constants to ordinary floating constants.

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

After the floating point constant is recognized, it is scanned by the <u>for</u> loop to find the letter <u>d</u> or <u>D</u>. The program than adds <u>'e'-'d'</u>, 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 <u>d</u>. By using the array <u>yytext</u> 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 <u>d</u> changed to initial <u>a</u>:

```
{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 <u>d</u> changed to initial <u>r</u>:

```
{d}l{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.

4.2.10 Left Context Sensitivity

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as $ recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of start conditions on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the 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 which began with the letter a, changing magic to second on every line which began with the letter b, and changing magic to third on every line which 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

```
        %Startname1 name2 ...
```

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

```
        <name1>expression
```

is a rule which is only recognized when Lex is in the start condition name1. To enter a start condition, execute the action statement

```
        BEGIN name1;
```

which changes the start condition to name1. To resume the normal state,

```
     BEGIN 0;
```

resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions:

```
     <name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
     %START AA BB CC
     %%
     ^a{ECHO; BEGIN AA;}
     ^b{ECHO; BEGIN BB;}
     ^c{ECHO; BEGIN CC;}
     \n{ECHO; BEGIN 0;}
     <AA>magicprintf("first");
     <BB>magicprintf("second");
     <CC>magicprintf("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.

4.2.11 Character Set

The programs generated by Lex handle character I/O only through the routines _input_, _output_, and _unput_. Thus the character representation provided in these routines is accepted by Lex and employed to return values in _yytext_. For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter _a_ is represented 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.

Thus the next example

```
%T
 1Aa
 2Bb
...
26Zz
27\n
28+
29-
300
311
...
399
%T
```

Sample character table.

maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and - into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

4.2.12 Summary of 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:

LetterParameter	
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

4.2.13 Notes

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input. Instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used unput to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

4.3 YACC: Yet Another Compiler-Compiler

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 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.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) 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 (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 quotes; 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 ``,'' must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

```
     date  :  month '/' day '/' year    ;
```

allowing

```
     7 / 4 / 1776
```

as a synonym for

```
     July 4, 1776
```

In most cases, this new rule could be ``slipped in'' to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

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

The next several sections describe:

- The 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.
- gives some suggestions which should improve the style and efficiency of the specifications.

4.3.1 Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed 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 `%' 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 multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A  :  BODY  ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ``.'', underscore ``_'', and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes ``'''. As in C, the backslash ``\'' is an escape character within literals, and all the C escapes are recognized. Thus

```
'\n'newline
'\r'return
'\''single quote ``'''
'\\'backslash ``\''
'\t'tab
'\b'backspace
'\f'form feed
'\xxx'``xxx'' in octal
```

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar ``|'' can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

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

can be given to Yacc as

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

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :   ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token   name1  name2 . . .
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

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

```
%start   symbol
```

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

It is the job of the 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 ``end-of-file'' or ``end-of-record''.

4.3.2 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 symbol ``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 <u>expr</u> 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 <u>x</u> to 1, and <u>y</u> 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 Section 10.

4.3.3 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, 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; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the Lex program developed by Mike Lesk8 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.

4.3.4 How the Parser Works

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

The parser produced by Yacc consists of a finite state machine with a stack. The parser 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:

```
     IFshift 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

refers to grammar rule 18, while the action

```
     IFshift 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.

It is time for an example! Consider the specification

```
%token  DING  DONG  DELL
%%
rhyme:sound  place
  ;
sound:DING  DONG
  ;
place:DELL
  ;
```

When Yacc is invoked with the **-v** option, a file called _y.output_ is produced, with a human-readable description of the parser. The _y.output_ file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
  $accept  :  _rhyme  $end

  DING  shift 3
  .  error

  rhyme  goto 1
  sound  goto 2

state 1
  $accept  :   rhyme_$end

  $end  accept
  .  error

state 2
  rhyme  :   sound_place

  DELL  shift 5
  .  error

  place   goto 4

state 3
  sound   :   DING_DONG

  DONG  shift 6
  .  error

state 4
  rhyme  :   sound  place_    (1)

  .   reduce  1

state 5
  place  :   DELL_    (3)

  .   reduce  3

state 6
  sound   :   DING  DONG_    (2)

  .   reduce  2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

```
     DING  DONG  DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, DING, is read, becoming the lookahead token. The action in state 0 on DING is 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,

```
     soundgoto 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.

4.3.5 Ambiguity and Conflicts

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

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

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not 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 preceding ``un-ELSE'd'' IF. In this example, consider the situation where the parser has seen

```
     IF  (  C1  )  IF  (  C2  )  S1
```

and is looking at the ELSE. It can immediately reduce by the simple-if rule to get

```
     IF  (  C1  )  stat
```

and then read the remaining input,

```
     ELSE  S2
```

and reduce

```
     IF  (  C1  )  stat  ELSE  S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the ELSE may be shifted, S2 read, and then the right hand portion of

```
     IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be reduced by the if-else rule to get

```
     IF  (  C1  )  stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually 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.

4.3.6 Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of _precedence_ levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
     expr  :  expr  OP  expr
```

and

```
     expr  :  UNARY  expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the 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. %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.

4.3.7 Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found. It is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser ``restarted'' after an error. A general class of algorithms to 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 last line: " );  }  input
       {$$  =  $4;  }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
     yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

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

As mentioned above, the token seen immediately after the ``error'' symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
     yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the 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.

4.3.8 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 (due to local file system conventions, the names may differ from installation to installation). 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 (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) _yyparse_ returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, _yyparse_ returns the value 0.

The 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.

4.3.9 Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of ``knowing who to blame when things go wrong.''

b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

c. 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.

d. 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.

e. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The examples in the text of this section follow this style (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion The algorithm used by the Yacc parser encourages so called ``left recursive'' grammar rules: rules of the form

```
name:name  rest_of_rule  ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list:item
  |list  ','  item
  ;
```

and

```
seq:item
  |seq  item
  ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
     seq:item
       |item  seq
       ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
     seq:/* empty */
       |seq  item
       ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
  int dflag;
%}
  ...  other declarations ...

%%

prog:decls  stats
  ;

decls:/* empty */
  {dflag = 1;  }
  |decls  declaration
  ;

stats:/* empty */
  {dflag = 0;  }
  |stats  statement
  ;

    ...  other rules ...
```

The flag dflag is now 0 when reading statements, and 1 when reading declarations, exceptforthefirsttokenin This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of ``backdoor'' approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words Some programming languages permit the 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, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be reserved; that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway.

4.3.10 Advanced Topics

This section discusses a number of advanced features of Yacc.

Simulating Error and Accept in Actions The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; *yyerror* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules. An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
    sent:adj  noun  verb  adj  noun
      {  look at the sentence . . .  }
      ;

    adj:THE{$$ = THE;  }
      |YOUNG{$$ = YOUNG;  }
      . . .
      ;

    noun:DOG
      {$$ = DOG;  }
      |CRONE
      {if( $0 == YOUNG ){
      printf( "what?\n" );
      }
      $$ = CRONE;
      }
      ;
      . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) 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 Lint5 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 can not 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 _a priori_ type. Similarly, reference to left context values (such as $0 - see the previous subsection) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
     rule:aaa  {  $<intval>$  =  3;  } bbb
       {fun( $<intval>2, $<other>0 );  }
       ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in 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.

4.3.11 A Simple Example

This example gives the complete Yacc specification for a small desk calculator: the desk calculator has 26 registers, labeled ``a'' through ``z'', and accepts arithmetic expressions made up of the operators +, -, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

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

int  regs[26];
int  base;

%}

%start  list

%token  DIGIT  LETTER

%left  '|'
%left  '&'
%left  '+'  '-'
%left  '*'  '/'  '%'
%left  UMINUS      /*  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  lower  case  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  );
     }
```

4.3.12 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 IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
            /*  grammar  for  the  input  to  Yacc  */

      /*  basic  entities  */
%token      IDENTIFIER  /*  includes  identifiers  and literals  */
%token      C_IDENTIFIER      /*    identifier  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  ';'
        ;
```

4.3.13 An Advanced Example

This section gives an example of a grammar using some of the advanced features discussed in earlier sections. The desk calculator example is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, -, *, /, unary -, and = (assignment), and has 26 floating point variables, ``a'' through ``z''. Moreover, it also understands _intervals_, written

```
( x , y )
```

where _x_ is less than or equal to _y_. There are 26 interval valued variables ``A'' through ``Z'' that may also be used. Assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as _double_'s. This structure is given a type name, INTERVAL, by using _typedef_. The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (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 ``,'' is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine <u>atof</u> 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'*'  '/'
%leftUMINUS        /*  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;

   while(  (c=getchar())  ==  ' '  ){  /*  skip  over  blanks  */  }

   if(  isupper(  c  )  ){
   yylval.ival  =  c  -  'A';
   return(  VREG  );
   }
   if(  islower(  c  )  ){
   yylval.ival  =  c  -  'a';
   return(  DREG  );
   }

   if(  isdigit(  c  )  ||  c=='.'  ){
   /*  gobble  up  digits,  points,  exponents  */

   char  buf[BSZ+1],  *cp  =  buf;
   int  dot  =  0,  exp  =  0;

   for(  ;  (cp-buf)<BSZ  ;  ++cp,c=getchar()  ){

   *cp  =  c;
   if(  isdigit(  c  )  )  continue;
   if(  c  ==  '.'  ){
   if(  dot++  ||  exp  )  return(  '.'  );    /*  will  cause  syntax
   continue;
   }

   if(  c  ==  'e'  ){
```

```
  if( exp++ ) return( 'e' );   /* will cause syntax error */
  continue;
  }

  /* end of number */
  break;
  }
  *cp = '\0';
  if( (cp-buf) >= BSZ ) printf( "constant too long: truncated\n"
  else ungetc( c, stdin );   /* push back last char read */
  yylval.dval = atof( buf );
  return( CONST );
  }
  return( c );
  }

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
  /* returns the smallest interval containing a, b, c, and d */
  /* used by *, / routines */
  INTERVAL v;

  if( a>b ) { v.hi = a;   v.lo = b; }
  else { v.hi = b;   v.lo = a; }

  if( c>d ) {
  if( c>v.hi ) v.hi = c;
  if( d<v.lo ) v.lo = d;
  }
  else {
  if( d>v.hi ) v.hi = d;
  if( c<v.lo ) v.lo = c;
  }
  return( v );
  }

INTERVAL vmul( a, b, v ) double a, b;  INTERVAL v; {
  return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
  }

dcheck( v ) INTERVAL v; {
  if( v.hi >= 0. && v.lo <= 0. ){
  printf( "divisor interval contains 0.\n" );
  return( 1 );
  }
  return( 0 );
  }

INTERVAL vdiv( a, b, v ) double a, b;  INTERVAL v; {
  return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
  }
```

4.3.14 Old Features Supported but not Encouraged

This section mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes ``"''.

2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals. The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job that must be actually done by the lexical analyzer.

3. Most places where % is legal, backslash ``\'' may be used. In particular, \\ is the same as %%, \left the same as %left, etc.

4. There are a number of other synonyms:

   ```
   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec
   ```

5. Actions may also have the form

   ```
   ={ . . . }
   ```

   and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

4.4 DC: An Interactive Desk Calculator

DC is an interactive desk calculator program designed to perform arbitrary-precision integer arithmetic. It has provision for manipulating scaled fixed-point numbers and for input and output in bases other than decimal. The size of numbers that can be manipulated is limited only by available core storage. On typical implementations of XENIX, the size of numbers that can be handled varies from several hundred digits on the smallest systems to several thousand on the largest.

DC works like a stacking calculator using reverse Polish notation. Ordinarily DC operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained.

A language called BC has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles output which is interpreted by DC. Some of the commands described below were designed for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into DC are put on a push-down stack. DC commands work by taking the top number or two off the stack, performing the desired operation, and pushing the result on the stack. If an argument is given, input is taken from that file until its end, then from the standard input.

4.4.1 Synopsis

Here we describe the more commonly used DC commands. The additional commands that are intended to be invoked by compiled output are described in the detailed description.

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

number	The value of the number is pushed onto the main stack. A number is an unbroken string of the digits 0-9 and the capital letters A-F which are treated as digits with values 10-15 respectively. The number may be preceded by an underscore to input a negative number. Numbers may contain decimal points.

+ - * % ^
: The top two values on the stack are added (+), subtracted (-), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. The result of a division is an integer truncated toward zero. See the detailed description below for the treatment of numbers with decimal points. An exponent must not have any digits after the decimal point.

s_x_
: The top of the main stack is popped and stored into a register named _x_, where _x_ may be any character. If the s is capitalized, _x_ is treated as a stack and the value is pushed onto it. Any character, even blank or new-line, is a valid register name.

l_x_
: The value in register _x_ is pushed onto the stack. The register _x_ is not altered. If the l is capitalized, register _x_ is treated as a stack and its top value is popped onto the main stack.

All registers start with empty value which is treated as a zero by the command l and is treated as an error by the command L.

d
: The top value on the stack is duplicated.

p
: The top value on the stack is printed. The top value remains unchanged.

f
: All values on the stack and in registers are printed.

x
: treats the top element of the stack as a character string, removes it from the stack, and executes it as a string of DC commands.

[...]
: puts the bracketed character string onto the top of the stack.

q
: exits the program. If executing a string, the recursion level is popped by two. If q is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

B<_x_ >_x_ =_x_ !<_x_ !>_x_ !=_x_
: The top two elements of the stack are popped and

compared. Register x is executed if they obey the stated relation. Exclamation point is negation.

v	replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer. For the treatment of numbers with decimal points, see the detailed description below.
!	interprets the rest of the line as a XENIX command. Control returns to DC when the XENIX command terminates.
c	All values on the stack are popped; the stack becomes empty.
i	The top value on the stack is popped and used as the number radix for further input. If i is capitalized, the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.
o	The top value on the stack is popped and used as the number radix for further output. If o is capitalized, the value of the output base is pushed onto the stack.
k	The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If k is capitalized, the value of the scale factor is pushed onto the stack.
z	The value of the stack level is pushed onto the stack.
?	A line of input is taken from the input source (usually the console) and executed.

4.4.2 Internal Representation of Numbers

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0-99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100's complement notation, which is analogous to two's complement notation for binary numbers. The high order digit of a negative number is always -1 and all other digits are in the range 0-99. The digit preceding the high order -1 digit is never a 99. The representation of -157 is 43,98,-1. We shall call this the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when that is convenient, as it sometimes is.

An additional byte is stored with each number beyond the high order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,3 where the scale has been italicized to emphasize the fact that it is not the high order digit. The value of this extra byte is called the scale factor of the number.

4.4.3 The Allocator

DC uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is done through the allocator. Associated with each string in the allocator is a four-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and DC is done via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of 2. When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Left-over strings are put on the free list. If there are no larger strings, the allocator tries to coalesce smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that is next to it in core and, if free, can be combined with it to make a string twice as long.

Failing to find a string of the proper length after coalescing, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in DC. If at any time in the process of trying to allocate a string, the allocator runs out of headers, it also asks the system for more space.

There are routines in the allocator for reading, writing, copying, rewinding, forward-spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end-of-string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

4.4.4 Internal Arithmetic

All arithmetic operations are done on integers. The operands (or operand) needed for the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called **scale** plays a part in the results of most arithmetic operations. **scale** is the bound on the number of decimal places retained in arithmetic computations. **scale** may be set to the number on the top of the stack truncated to an integer with the **k** command. **K** may be used to push the value of **scale** on the stack. **scale** must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations will include the exact effect of **scale** on the computations.

4.4.5 Addition and Subtraction

The scales of the two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

Finally, the addition is performed digit by digit from the low order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers replacing the high-order configuration 99,-1 by the digit -1. In any case, digits which are not in the range 0-99 must be brought into that range, propagating any carries or borrows that result.

4.4.6 Multiplication

The scales are removed from the two operands and saved. The operands are both made positive. Then multiplication is performed in a digit by digit manner that exactly mimics the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low order digit. The intermediate products are accumulated into a partial sum which becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register **scale** and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

4.4.7 Division

The scales are removed from the two operands. Zeros are appended or digits removed from the dividend to make the scale of the result of the integer division equal to the internal quantity scale. The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. It may turn out be one unit too low, but if it is, the next trial quotient will be larger than 99 and this will be adjusted at the end of the process. The trial digit is multiplied by the divisor and the result subtracted from the dividend and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form, with propagation of carry as needed. The sign is set from the sign of the operands.

4.4.8 Remainder

The division routine is called and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

4.4.9 Square Root

The scale is stripped from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity scale and the scale of the operand.

The method used to compute sqrt(y) is Newton's method with The initial guess is found by taking the integer square root of the top two digits.

4.4.10 Exponentiation

Only exponents with zero scale factor are handled. If the exponent is zero, then the result is 1. If the exponent is negative, then it is made positive and the base is divided into one. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent. Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

4.4.11 Input Conversion and Base

Numbers are converted to the internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with a _. The hexadecimal digits A-F correspond to the numbers 10-15 regardless of input base. The i command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input. The input base is initialized to 10 but may, for example be changed to 8 or 16 to do octal or hexadecimal to decimal conversions. The command I will push the value of the input base on the stack.

4.4.12 Output Commands

The command p causes the top of the stack to be printed. It does not remove the top of the stack. All of the stack and internal registers can be output by typing the command f. The o command can be used to change the output base. This command uses the top of the stack, truncated to an integer as the base for all further output. The output base in initialized to 10. It will work correctly for any base. The command O pushes the value of the output base on the stack.

4.4.13 Output Format and Base

The input and output bases only affect the interpretation of numbers on input and output; they have no effect on arithmetic computations. Large numbers are output with 70 characters per line; a \ indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 can be used for decimal-octal or decimal-hexadecimal conversions.

4.4.14 Internal Registers

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands s and l. The command s<u>x</u> pops the top of the stack and stores the result in register x. <u>x</u> can be any character. l<u>x</u> puts the contents of register x on the top of the stack. The l command has no effect on the contents of register <u>x</u>. The s command, however, is destructive.

4.4.15 Stack Commands

The command c clears the stack. The command **d** pushes a duplicate of the number on the top of the stack on the stack. The command **z** pushes the stack size on the stack. The command **X** replaces the number on the top of the stack with its scale factor. The command **Z** replaces the top of the stack with its length.

4.4.16 Subroutine Definitions and Calls

Enclosing a string in [] pushes the ascii string on the stack. The q command quits or in executing a string, pops the recursion levels by two.

4.4.17 Internal Registers - Programming DC

The load and store commands together with [] to store strings, x to execute and the testing commands `<', `>', `=', `!<', `!>', `!=' can be used to program DC. The **x** command assumes the top of the stack is an string of DC commands and executes it. The testing commands compare the top two elements on the stack and if the relation holds, execute the register that follows the relation. For example, to print the numbers 0-9,

```
[lipl+  si  li10>a]sa
0si  lax
```

4.4.18 Push-Down Registers and Arrays

These commands were designed for used by a compiler, not by people. They involve push-down registers and arrays. In addition to the stack that commands work on, DC can be thought of as having individual stacks for each register. These registers are operated on by the commands S and L. S_x_ pushes the top value of the main stack onto the stack for the register _x_. L_x_ pops the stack for register _x_ and puts the result on the main stack. The commands s and l also work on registers but not as push-down stacks. l doesn't effect the top of the register stack, and s destroys what was there before.

The commands to work on arrays are : and ;. :_x_ pops the stack and uses this value as an index into the array _x_. The next element on the stack is stored at this index in _x_. An index must be greater than or equal to 0 and less than 2048. ;_x_ is the command to load the main stack from the array _x_.

The value on the top of the stack is the index into the array _x_ of the value to be loaded.

4.4.19 Miscellaneous Commands

The command ! interprets the rest of the line as a XENIX command and passes it to XENIX to execute. One other compiler command is Q. This command uses the top of the stack as the number of levels of recursion to skip.

4.5 BC: A Desk-Calculator Language

BC is a language and a compiler for doing arbitrary precision arithmetic. The output of the compiler is interpreted and executed by a collection of routines which can input, output, and do arithmetic on indefinitely large integers and on scaled fixed-point numbers.

These routines are themselves based on a dynamic storage allocator. Overflow does not occur until all available core storage is exhausted.

The language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution.

Two 500-digit numbers can be multiplied to give a 1000-digit result in about ten seconds.

A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

Some of the uses of this compiler are

- To perform computation with large integers,
- To perform computations accurate to many decimal places,
- To convert numbers from one base to another base.

The compiler was written to make conveniently available a collection of routines (called DC [5]) which are capable of doing arithmetic on integers of arbitrary size. The compiler is by no means intended to provide a complete programming language. It is a minimal language facility.

There is a scaling provision that permits the use of decimal point notation. Provision is made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base to equal 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine. Manipulation of numbers with many hundreds of digits is possible even on the smallest versions of XENIX .

The syntax of BC has been deliberately selected to agree substantially with the C language [2]. Those who are familiar with C will find few surprises in this language.

4.5.1 Simple Computations with Integers

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the line:

```
142857 + 285714
```

the program responds immediately with the line

```
428571
```

The operators -, *, /, %, and ^ can also be used; they indicate subtraction, multiplication, division, remaindering, and exponentiation, respectively. Division of integers produces an integer result truncated toward zero. Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the `unary' minus sign). The expression

```
7+-3
```

is interpreted to mean that -3 is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in Fortran, with ^ having the greatest binding power, then * and % and /, and finally + and -. Contents of parentheses are evaluated before material outside the parentheses. Exponentiations are performed from right to left and the other operators from left to right. The two expressions

```
a^b^c  and  a^(b^c)
```

are equivalent, as are the two expressions

```
a*b*c  and  (a*b)*c
```

BC shares with Fortran and C the convention that

```
a/b*c  is equivalent to  (a/b)*c
```

Internal storage registers to hold numbers have single lower-case letter names. The value of an expression can be assigned to a register in the usual way. The statement

```
     x = x + 3
```

has the effect of increasing by three the value of the contents of the register named x. When, as in this case, the outermost operator is an =, the assignment is performed but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (but see scaling below). The lines

```
     x = sqrt(191)
     x
```

produce the printed result

```
     13
```

4.5.2 Bases

There are special internal quantities, called \`ibase' and \`obase'. The contents of \`ibase', initially set to 10, determines the base used for interpreting numbers read in. For example, the lines

```
     ibase = 8
     11
```

will produce the output line

```
     9
```

and you are all set up to do octal to decimal conversions. Beware, however of trying to change the input base back to decimal by typing

```
     ibase = 10
```

Because the number 10 is interpreted as octal, this statement will have no effect. For those who deal in hexadecimal notation, the characters A-F are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values 10-15 respectively. The statement

```
     ibase = A
```

will change you back to decimal input base no matter what the current input base is. Negative and large positive input bases are permitted but useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The contents of `obase', initially set to 10, are used as the base for output numbers. The lines

```
obase = 16
1000
```

will produce the output line

```
3E8
```

which is to be interpreted as a 3-digit hexadecimal number. Very large output bases are permitted, and they are sometimes useful. For example, large numbers can be output in groups of five digits by setting `obase' to 100000. Strange (i.e. 1, 0, or negative) output bases are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with \. Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Non-decimal output conversion of a one hundred digit number takes about three seconds.

It is best to remember that `ibase' and `obase' have no effect whatever on the course of internal computation or on the evaluation of expressions, but only affect input and output conversion, respectively.

4.5.3 Scaling

A third special internal quantity called `scale' is used to determine the scale of calculated quantities. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations. We refer to the number of digits after the decimal point of a number as its scale.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules. For addition and subtraction, the scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result. For multiplications, the scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands and, subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity `scale'. The scale of a quotient is the contents of the internal quantity `scale'. The scale of a remainder is the sum of the scales of the quotient and the divisor. The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer. The scale of a square root is set to the maximum of the scale of the argument and the contents of `scale'.

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case where digits are discarded, truncation and not rounding is performed.

The contents of `scale' must be no greater than 99 and no less than 0. It is initially set to 0. In case you need more than 99 fraction digits, you may arrange your own scaling.

The internal quantities `scale', `ibase', and `obase' can be used in expressions just like other variables. The line

```
     scale = scale + 1
```

increases the value of `scale' by one, and the line

```
     scale
```

causes the current value of `scale' to be printed.

The value of `scale' retains its meaning as a number of decimal digits to be retained in internal computation even when `ibase' or `obase' are not equal to 10. The internal computations (which are still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

4.5.4 Functions

The name of a function is a single lower-case letter. Function names are permitted to collide with simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line

```
       define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace }.

Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return statement can take either of the two forms

```
return
return(x)
```

In the first case, the value of the function is 0, and in the second, the value of the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one `auto' statement in a function and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition is

```
define a(x,y){
auto z
z = x*y
return(z)
}
```

The value of this function, when called, will be the product of its two arguments.

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them: b().

If the function a above has been defined, then the line

```
a(7,3.14)
```

would cause the result 21.98 to be printed and the line

```
     x = a(a(3,4),5)
```

would cause the value of x to become 60.

4.5.5 Subscripted Variables

A single lower-case letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name and the expression in brackets is called the subscript. Only one-dimensional arrays are permitted. The names of arrays are permitted to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables may be freely used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function, or may be declared as automatic in a function definition by the use of empty brackets:

```
     f(a[])
     define f(a[])
     auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function, and thrown away on exit from the function. Array names which refer to whole arrays cannot be used in any other contexts.

4.5.6 Control Statements

The `if', the `while', and the `for' statements may be used to alter the flow within programs or to cause iteration. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following way

```
     if(relation) statement
     while(relation) statement
     for(expression1; relation; expression2) statement
```

or

```
if(relation) {statements}
while(relation) {statements}
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form

```
x>y
```

where two expressions are related by one of the six relational operators <, >, <=, >=, ==, or !=. The relation == stands for `equal to' and != stands for `not equal to'. The meaning of the remaining relational operators is clear.

BEWARE of using = instead of == in a relational. Unfortunately, both of them are legal, so you will not get a diagnostic message, but = really will not do a comparison.

The `if' statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in sequence.

The `while' statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range and if the relation is false, control passes to the next statement beyond the range of the while.

The `for' statement begins by executing `expression1'. Then the relation is tested and, if true, the statements in the range of the `for' are executed. Then `expression2' is executed. The relation is tested, and so on. The typical use of the `for' statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10. Here are some examples of the use of the control statements.

```
define f(n){
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
```

The line

```
f(a)
```

will print _a_ factorial if _a_ is a positive integer. Here is the definition of a function which will compute values of the binomial coefficient (m and n are assumed to be positive integers).

```
define b(n,m){
auto x, j
x=1
for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
  scale = 20
  define e(x){
    auto a, b, c, d, n
    a = 1
    b = 1
    c = 1
    d = 0
    n = 1
    while(1==1){
    a = a*x
    b = b*n
    c = c + a/b
    n = n + 1
    if(c==d) return(c)
    d = c
    }
}
```

4.5.7 Some Details

There are some language features that every user should know about even if he will not use them.

Normally statements are typed one to a line. It is also permissible to type several statements on a line separated by semicolons.

If an assignment statement is parenthesized, it then has a value and it can be used anywhere that an expression can. For example, the line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

Here is an example of a use of the value of an assignment statement even when it is not parenthesized.

```
x = a[i=i+1]
```

causes a value to be assigned to x and also increments i before it is used as a subscript.

The following constructs work in BC in exactly the same manner as they do in the C language.

```
x=y=z  is the same asx=(y=z)
x =+ y                x = x+y
x =- y                x = x-y
x =* y                x = x*y
x =/ y                x = x/y
x =% y                x = x%y
x =^ y                x = x^y
x++                   (x=x+1)-1
x--                   (x=x-1)+1
++x                   x = x+1
--x                   x = x-1
```

Even if you don't intend to use the constructs, if you type one inadvertently, something correct but unexpected may happen.

WARNING! In some of these constructions, spaces are significant. There is a real difference between x=-y and x= -y. The first replaces x by x-y and the second by -y.

4.5.8 Three Important Things

1. To exit a BC program, type `quit'.

2. There is a comment convention identical to that of C. Comments begin with `/*' and end with `*/'.

3. There is a library of math functions which may be obtained by typing at command level

   ```
   bc -l
   ```

   This command will load a set of library functions which, at the time of writing, consists of sine (named `s'), cosine (`c'), arctangent (`a'), natural logarithm (`l'), exponential (`e') and Bessel functions of integer order (`j(n,x)'). Doubtless more functions will be added in time. The library sets the scale to 20. You can reset it to something else if you like. The design of these mathematical library routines is discussed elsewhere.

If you type

```
bc file ...
```

BC will read and execute the named file or files before accepting commands from the keyboard. In this way, you may load your favorite programs and function definitions.

4.5.9 Notation

In the following pages syntactic categories are in *italics*; literals are in **bold**; items in brackets [] is optional.

4.5.9.1 Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs or comments. Newline characters or semicolons separate statements.

Comments Comments are introduced by the characters /* and terminated by */.

Identifiers There are three kinds of identifiers - ordinary identifiers, array identifiers and function identifiers. All three types consist of single lower-case letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict; a program can have a variable named **x**, an array named **x** and a function named **x**, all of which are separate and distinct.

Keywords The following are reserved keywords:

```
    ibaseif
    obasebreak
    scaledefine
    sqrt auto
    lengthreturn
    whilequit
    for
```

Constants Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits **A-F** are also recognized as digits with values 10-15, respectively.

4.5.9.2 Expressions The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here, with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

Primitive expressions

Named expressions Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

4.5.9.3 identifiers Simple identifiers are named expressions. They have an initial value of zero.

4.5.9.4 array-name[expression] Array elements are named expressions. They have an initial value of zero.

4.5.9.5 **scale, ibase** and **obase** The internal registers **scale, ibase** and **obase** are all named expressions. **scale** is the number of digits after the decimal point to be retained in arithmetic operations. **scale** has an initial value of zero. **ibase** and **obase** are the input and output number radix respectively. Both **ibase** and **obase** have initial values of 10.

Function calls

4.5.9.6 function-name([expression[,expression...]]) A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement or is zero if no expression is provided or if there is no return statement.

4.5.9.7 sqrt(expression) The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of scale whichever is larger.

4.5.9.8 length(expression) The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

4.5.9.9 scale(expression) The result is the scale of the expression. The scale of the result is zero.

Constants Constants are primitive expressions.

Parentheses An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

Unary operators The unary operators bind right to left.

-expression The result is the negative of the expression.

++named-expression The named expression is incremented by one. The result is the value of the named expression after incrementing.

--named-expression The named expression is decremented by one. The result is the value of the named expression after decrementing.

named-expression++ The named expression is incremented by one. The result is the value of the named expression before incrementing.

named-expression-- The named expression is decremented by one. The result is the value of the named expression before decrementing.

Exponentiation operator The exponentiation operator binds right to left.

<u>expression ^ expression</u> The result is the first expression raised to the power of the second expression. The second expression must be an integer. If <u>a</u> is the scale of the left expression and <u>b</u> is the absolute value of the right expression, then the scale of the result is:

min(<u>a</u>x<u>b</u>,max(scale,<u>a</u>))

<u>Multiplicative operators</u> The operators *, /, % bind left to right.

<u>expression * expression</u> The result is the product of the two expressions. If <u>a</u> and <u>b</u> are the scales of the two expressions, then the scale of the result is:

min(<u>a</u>+<u>b</u>,max(scale,<u>a</u>,<u>b</u>))

<u>expression / expression</u> The result is the quotient of the two expressions. The scale of the result is the value of scale.

<u>expression % expression</u> The % operator produces the remainder of the division of the two expressions. More precisely, <u>a</u>%<u>b</u> is <u>a</u>-<u>a</u>/<u>b</u>*<u>b</u>.

The scale of the result is the sum of the scale of the divisor and the value of scale.

<u>Additive operators</u> The additive operators bind left to right.

<u>expression + expression</u> The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.

<u>expression - expression</u> The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

assignment operators The assignment operators bind right to left.

named-expression = expression This expression results in assigning the value of the expression on the right to the named expression on the left.

named-expression =+ expression

named-expression =- expression

named-expression =* expression

named-expression =/ expression

named-expression =% expression

named-expression =^ expression The result of the above expressions is equivalent to ``named expression = named expression OP expression'', where OP is the operator after the = sign.

4.5.9.10 Relations Unlike all other operators, the relational operators are only valid as the object of an if, while, or inside a for statement.

expression < expression

expression > expression

expression <= expression

expression >= expression

expression == expression

expression != expression

4.5.9.11 Storage classes There are only two storage classes in BC, global and automatic (local). Only identifiers that are to be local to a function need be declared with the auto command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as auto are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. auto arrays are specified by the array name followed by empty square brackets.

Automatic variables in BC do not work in exactly the same way as in either C or PL/I. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

4.5.9.12 Statements Statements must be separated by semicolon or newline. Except where altered by control statements, execution is sequential.

Expression statements When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

Compound statements Statements may be grouped together and used when one statement is expected by surrounding them with { }.

Quoted string statements "any string"
This statement prints the string inside the quotes.

If statements

if(relation)statement

The substatement is executed if the relation is true.

While statements

while(relation)statement

The statement is executed while the relation is true. The test occurs before each execution of the statement.

For statements

for(expression; relation; expression)statement

The for statement is the same as

```
     first-expression
     while(relation) {
          statement
          last-expression
     }
```

All three expressions must be present.

Break statements

break

break causes termination of a for or while statement.

Auto statements

auto identifier[,identifier]

The auto statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The auto statement must be the first statement in a function definition.

Define statements

define([*parameter*[,*parameter*...]]){
     statements}

The define statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

Return statements

return

return(*expression*)

The return statement causes termination of a function, popping of its auto variables, and specifies the result of the function. The first form is equivalent to return(0). The result of the function is the result of the expression in parentheses.

Quit The quit statement stops execution of a BC program and returns control to XENIX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an if, for, or while statement.

NAME

intro - introduction to commands

DESCRIPTION

This section describes publicly accessible commands in alphabetic order. Certain distinctions of purpose are made in the headings:

(1) Commands of general utility.

(1C) Commands for communication with other systems.

(1G) Commands used primarily for graphics and computer-aided design.

(1M) Commands used primarily for system maintenance.

The word `local' at the foot of a page means that the command is not intended for general distribution.

SEE ALSO

DIAGNOSTICS

Section (6) for computer games.

How to get started, in the Introduction.

DIAGNOSTICS

Upon termination each command returns two bytes of status, one supplied by the system giving the cause for termination, and (in the case of `normal' termination) one supplied by the program, see *wait* and *exit*(2). The former byte is 0 for normal termination, the latter is customarily 0 for successful execution, nonzero to indicate troubles such as erroneous parameters, bad or inaccessible data, or other inability to cope with the task at hand. It is called variously `exit code', `exit status' or `return code', and is described only where special conventions are involved.

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; INTERRUPT 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 #; 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 utter this name to disinguish it from internal or hidden variables of a program.

routine.*name*
The address of the variable *name* in the specified C routine. Both *routine* and *name* are *symbols*. If *name* is omitted the value is the address of the most recently activated C stack frame corresponding to *routine*.

(*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.

_el_+_e2_	Integer addition.
_el_-_e2_	Integer subtraction.
_el_*_e2_	Integer multiplication.
_el_%_e2_	Integer division.
_el_&_e2_	Bitwise conjunction.
_el_|_e2_	Bitwise disjunction.
_el_#_e2_	_El_ 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<*address*<*e1* => *file address*=*address*+*f1*-*b1*, otherwise,

b2<*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

ptrace(2), a.out(5), core(5)

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.

NOTES

A breakpoint set at the entry point is not effective on initial entry to the program.
When single stepping, system calls do not count as an executed instruction.
Local variables whose names are the same as an external

variable may foul up the accessing of the external.

NAME

as - assembler

SYNOPSIS

as [-l] [-o objfile] file

DESCRIPTION

_As_ assembles the named file. If the optional first argument _-l_ is used, an assembly listing is produced and written to _file.L_. This includes the source, the assembled (binary) code, and any assembly errors.

The output of the assembly is left on the file _objfile_; if that is omitted, _file.o_ is used. It is executable if no errors occurred during the assembly.

FILES

/tmp/AS*	temporary
a.out	object

SEE ALSO

ld(1), nm(1), adb(1), a.out(5)
8086 Assembler Manual   Martin Katz

NAME

at - execute commands at a later time

SYNTAX

at time [day] [file]

DESCRIPTION

At squirrels away a copy of the named *file* (standard input default) to be used as input to *sh*(1) at a specified later time. A *cd*(1) command to the current directory is inserted at the beginning, followed by assignments to all environment variables. When the script is run, it uses the user and group ID of the creator of the copy file.

The *time* is 1 to 4 digits, with an optional following `A', `P', `N' or `M' for AM, PM, noon or midnight. One and two digit numbers are taken to be hours, three and four digits to be hours and minutes. If no letters follow the digits, a 24 hour clock time is understood.

The optional *day* is either (1) a month name followed by a day number, or (2) a day of the week; if the word `week' follows invocation is moved seven days further off. Names of months and days may be recognizably truncated. Examples of legitimate commands are

```
at 8am jan 24
at 1530 fr week
```

At programs are executed by periodic execution of the command */usr/lib/atrun* from *cron*(8). The granularity of *at* depends upon how often *atrun* is executed.

Standard output or error output is lost unless redirected.

FILES

/usr/spool/at/yy.ddd.hhhh.uu
activity to be performed at hour *hhhh* of year day *ddd* of year *yy*. *uu* is a unique number.
/usr/spool/at/lasttimedone contains *hhhh* for last hour of activity.
/usr/spool/at/past directory of activities now in progress
/usr/lib/atrun program that executes activities that are due
pwd(1)

SEE ALSO

calendar(1), cron(8)

DIAGNOSTICS

Complains about various syntax errors and times out of range.

NOTES

Due to the granularity of the execution of /*usr*/*lib*/*atrun*, there may be bugs in scheduling things almost exactly 24 hours into the future.

NAME

awk - pattern scanning and processing language

SYNTAX

awk [-F*c*] [prog] [file] ...

DESCRIPTION

Awk scans each input *file* for lines that match any of a set of patterns specified in *prog*. With each pattern in *prog* there can be an associated action that will be performed when a line of a *file* matches the pattern. The set of patterns may appear literally as *prog*, or in a file specified as -f *file*.

Files are read in order; if there are no files, the standard input is read. The file name `-' means the standard input. Each line is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern.

An input line is made up of fields separated by white space. (This default can be changed by using FS, *vide infra*.) The fields are denoted $1, $2, ... ; $0 refers to the entire line.

A pattern-action statement has the form

```
pattern { action }
```

A missing { action } means print the line; a missing pattern always matches.

An action is a sequence of statements. A statement can be one of the following:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
break
continue
{ [ statement ] ... }
variable = expression
print [ expression-list ] [ >expression ]
printf format [ , expression-list ] [ >expression ]
next # skip remaining patterns on this input line
exit # skip the rest of the input
```

Statements are terminated by semicolons, newlines or right braces. An empty expression-list stands for the whole line. Expressions take on string or numeric values as appropriate, and are built using the operators +, -, *, /, %, and concatenation (indicated by a blank). The C operators ++, --,

+=, -=, *=, /=, and %= are also available in expressions. Variables may be scalars, array elements (denoted x[i]) or fields. Variables are initialized to the null string. Array subscripts may be any string, not necessarily numeric; this allows for a form of associative memory. String constants are quoted "...".

The _print_ statement prints its arguments on the standard output (or on a file if >_file_ is present), separated by the current output _field separator_, and terminated by the output record separator. The _printf_ statement formats its expression list according to the format (see _printf_(3)).

The built-in function _length_ returns the length of its argument taken as a string, or of the whole line if no argument. There are also built-in functions _exp_, _log_, _sqrt_, and _int_. The last truncates its argument to an integer. _substr_(_s_, _m_, _n_) returns the _n_-character substring of _s_ that begins at position _m_. The function _sprintf_(_fmt_, _expr_, _expr_, ...) formats the expressions according to the _printf_(3) format given by _fmt_ and returns the resulting string.

Patterns are arbitrary Boolean combinations (!, ||, &&, and parentheses) of regular expressions and relational expressions. Regular expressions must be surrounded by slashes and are as in _egrep_. Isolated regular expressions in a pattern apply to the entire line. Regular expressions may also occur in relational expressions.

A pattern may consist of two patterns separated by a comma; in this case, the action is performed for all lines between an occurrence of the first pattern and the next occurrence of the second.

A relational expression is one of the following:

```
expression matchop regular-expression
expression relop expression
```

where a relop is any of the six relational operators in C, and a matchop is either ~ (for contains) or !~ (for does not contain). A conditional is an arithmetic expression, a relational expression, or a Boolean combination of these.

The special patterns BEGIN and END may be used to capture control before the first input line is read and after the last. BEGIN must be the first pattern, END the last.

A single character _c_ may be used to separate the fields by starting the program with

```
BEGIN { FS = "c" }
```

or by using the -F<u>c</u> option.

Other variable names with special meanings include NF, the number of fields in the current record; NR, the ordinal number of the current record; FILENAME, the name of the current input file; OFS, the output field separator (default blank); ORS, the output record separator (default newline); and OFMT, the output format for numbers (default "%.6g").

EXAMPLES

Print lines longer than 72 characters:

```
length > 72
```

Print first two fields in opposite order:

```
{ print $2, $1 }
```

Add up first column, print sum and average:

```
    { s += $1 }
END { print "sum is", s, " average is", s/NR }
```

Print fields in reverse order:

```
{ for (i = NF; i > 0; --i) print $i }
```

Print all lines between start/stop pairs:

```
/start/, /stop/
```

Print all lines whose first field is different from previous one:

```
$1 != prev { print; prev = $1 }
```

SEE ALSO

lex(1), sed(1)
A. V. Aho, B. W. Kernighan, P. J. Weinberger, <u>Awk - a pattern scanning and processing language</u>

NOTES

There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate "" to it.

NAME

basename - strip filename affixes

SYNTAX

basename string [suffix]

DESCRIPTION

Basename deletes any prefix ending in `/' and the *suffix*, if present in *string*, from *string*, and prints the result on the standard output. It is normally used inside substitution marks ` ` in shell procedures.

This shell procedure invoked with the argument /*usr*/*src*/*cmd*/*cat*.*c* compiles the named file and moves the output to *cat* in the current directory:

```
cc $1
mv a.out `basename $1 .c`
```

SEE ALSO

sh(1)

NAME

bc - arbitrary-precision arithmetic language

SYNTAX

bc [-c] [-l] [file ...]

DESCRIPTION

Bc is an interactive processor for a language which resembles C but provides unlimited precision arithmetic. It takes input from any files given, then reads the standard input. The -l argument stands for the name of an arbitrary precision math library. The syntax for bc programs is as follows; L means letter a-z, E means expression, S means statement.

```
Comments
     are enclosed in /* and */.

Names
     simple variables: L
     array elements: L [ E ]
     The words `ibase', `obase', and `scale'

Other operands
     arbitrarily long numbers with optional sign and
     decimal point.
     ( E )
     sqrt ( E )
     length ( E )   number of significant decimal digits
     scale ( E )    number of digits right of decimal point
     L ( E , ... , E )

Operators
     +  -  *  /  %  ^ (% is remainder; ^ is power)
     ++   --          (prefix and postfix; apply to names)
     ==  <=  >=  !=  <  >
     =  =+  =-  =*  =/  =%  =^

Statements
     E
     { S ; ... ; S }
     if ( E ) S
     while ( E ) S
     for ( E ; E ; E ) S
     null statement
     break
     quit

Function definitions
     define L ( L ,..., L ) {
          auto L, ... , L
          S; ... S
```

```
                return ( E )
        }
```

Functions in -l math library

```
        s(x) sine
        c(x) cosine
        e(x) exponential
        l(x) log
        a(x) arctangent
        j(n,x)    Bessel function
```

All function arguments are passed by value.

The value of a statement that is an expression is printed unless the main operator is an assignment. Either semicolons or newlines may separate statements. Assignment to _scale_ influences the number of digits to be retained on arithmetic operations in the manner of _dc_(1). Assignments to _ibase_ or _obase_ set the input and output number radix respectively.

The same letter may be used as an array, a function, and a simple variable simultaneously. All variables are global to the program. `Auto' variables are pushed down during function calls. When using arrays as function arguments or defining them as automatic variables empty square brackets must follow the array name.

For example

```
scale = 20
define e(x){
     auto a, b, c, i, s
     a = 1
     b = 1
     s = 1
     for(i=1; 1==1; i++){
          a = a*x
          b = b*i
          c = a/b
          if(c == 0) return(s)
          s = s+c
     }
}
```

defines a function to compute an approximate value of the exponential function and

```
     for(i=1; i<=10; i++) e(i)
```

prints approximate values of the exponential function of the first ten integers.

_Bc_ is actually a preprocessor for _dc_(1), which it invokes automatically, unless the -c (compile only) option is present. In this case the _dc_ input is sent to the standard output instead.

FILES

/usr/lib/lib.b	mathematical library
dc(1)	desk calculator proper

SEE ALSO

dc(1)
L. L. Cherry and R. Morris, _BC - An arbitrary precision desk-calculator language_

NOTES

No &&, ||, or ! operators.
For statement must have all three E's.
Quit is interpreted when read, not when executed.

NAME

cat - catenate and print

SYNTAX

cat [-u] file ...

DESCRIPTION

Cat reads each _file_ in sequence and writes it on the standard output. Thus

```
cat file
```

prints the file and

```
cat file1 file2 >file3
```

concatenates the first two files and places the result on the third.

If no _file_ is given, or if the argument `-' is encountered, _cat_ reads from the standard input. Output is buffered in 512-byte blocks unless the -u option is present.

SEE ALSO

pr(1), cp(1)

NOTES

Beware of `cat a b >a' and `cat a b >b', which destroy input files before reading them.

NAME

cb - C program beautifier

SYNTAX

cb

DESCRIPTION

_Cb_ places a copy of the C program from the standard input on the standard output with spacing and indentation that displays the structure of the program.

NOTES

Output is not always as one would desire.

NAME

cc - C compiler

SYNOPSIS

cc [option] ... file ...

DESCRIPTION

Cc is the XENIX C compiler. It accepts several types of arguments:

Arguments whose names end with `.c' are taken to be C source programs; they are compiled, and each object program is left on the file whose name is that of the source with `.o' substituted for `.c'. The `.o' file is normally deleted, however, if a single C program is compiled and loaded all at one go.

In the same way, arguments whose names end with `.s' are taken to be assembly source programs and are assembled, producing a `.o' file.

The following options are interpreted by *cc*. See *ld*(1) for load-time options.

-c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

-p Arrange for the compiler to produce code which counts the number of times each routine is called; also, if loading takes place, replace the standard startup routine by one which automatically calls *monitor*(3) at the start and arranges to write out a *mon.out* file at normal termination of execution of the object program. An execution profile can then be generated by use of *prof*(1).

-O Invoke an object-code optimizer.

-S Compile the named C programs, and leave the assembler-language output on corresponding files suffixed `.s'.

-P Run only the macro preprocessor and place the result for each `.c' file in a corresponding `.i' file. The resultant file has no `#' lines in it.

-o *output*
Name the final output file *output*. If this option is used the file `a.out' will be left undisturbed.

-D*name*=*def*

-D*name*	Define the *name* to the preprocessor, as if by \`#define'. If no definition is given, the name is defined as 1.
-U*name*	Remove any initial definition of *name*.
-I*dir*	\`#include' files whose names do not begin with \`/' are always sought first in the directory of the *file* argument, then in directories named in -I options, then in directories on a standard list.

Other arguments are taken to be either loader option arguments, or C-compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name **a.out**.

FILES

file.c	input file
file.o	object file
a.out	loaded output
/tmp/t[123]*	temproraries
/lib/cpp	preprocessor
/lib/c0	compiler for *cc* pass 1
/lib/c1	compiler for *cc* pass 2
/lib/c2	optional optimizer
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	runtime startoff with monitoring
/lib/libc.a	standard library, see *intro*(3)
/usr/include	standard directory for \`#include' files

SEE ALSO

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978
D. M. Ritchie, *C Reference Manual*
adb(1), ld(1)

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self-explanatory. Occasional messages may be produced by the assembler or the loader.

NAME

cd - change working directory

SYNTAX

cd directory

DESCRIPTION

Directory becomes the new working directory. The process must have execute (search) permission in *directory*.

Because a new process is created to execute each command, *cd* would be ineffective if it were written as a normal command. It is therefore recognized and executed by the Shell.

SEE ALSO

sh(1), pwd(1), chdir(2)

NAME

chmod - change mode

SYNTAX

chmod mode file ...

DESCRIPTION

The mode of each named file is changed according to *mode*, which may be absolute or symbolic. An absolute *mode* is an octal number constructed from the OR of the following modes:

4000	set user ID on execution
2000	set group ID on execution
1000	sticky bit, see *chmod*(2)
0400	read by owner
0200	write by owner
0100	execute (search in directory) by owner
0070	read, write, execute (search) by group
0007	read, write, execute (search) by others

A symbolic *mode* has the form:

[*who*] *op* *permission* [*op* *permission*] ...

The *who* part is a combination of the letters **u** (for user's permissions), **g** (group) and **o** (other). The letter **a** stands for **ugo**. If *who* is omitted, the default is *a* but the setting of the file creation mask (see umask(2)) is taken into account.

Op can be + to add *permission* to the file's mode, - to take away *permission* and = to assign *permission* absolutely (all other bits will be reset).

Permission is any combination of the letters **r** (read), **w** (write), **x** (execute), **s** (set owner or group id) and **t** (save text - sticky). Letters **u**, **g** or **o** indicate that *permission* is to be taken from the current mode. Omitting *permission* is only useful with = to take away all permissions.

The first example denies write permission to others, the second makes a file executable:

```
chmod o-w file
chmod +x file
```

Multiple symbolic modes separated by commas may be given. Operations are performed in the order specified. The letter **s** is only useful with **u** or **g**.

Only the owner of a file (or the super-user) may change its mode.

SEE ALSO

ls(1), chmod(2), chown (1), stat(2), umask(2)

NAME

chown, chgrp - change owner or group

SYNTAX

chown owner file ...

chgrp group file ...

DESCRIPTION

Chown changes the owner of the *files* to *owner*. The owner may be either a decimal UID or a login name found in the password file.

Chgrp changes the group-ID of the *files* to *group*. The group may be either a decimal GID or a group name found in the group-ID file.

Only the super-user can change owner or group, in order to simplify as yet unimplemented accounting procedures.

FILES

/etc/passwd
/etc/group

SEE ALSO

chown(2), passwd(5), group(5)

NAME

cmp - compare two files

SYNTAX

cmp [-l] [-s] file1 file2

DESCRIPTION

The two files are compared. (If *file1* is `-', the standard input is used.) Under default options, *cmp* makes no comment if the files are the same; if they differ, it announces the byte and line number at which the difference occurred. If one file is an initial subsequence of the other, that fact is noted.

Options:

-l Print the byte number (decimal) and the differing bytes (octal) for each difference.

-s Print nothing for differing files; return codes only.

SEE ALSO

diff(1), comm(1)

DIAGNOSTICS

Exit code 0 is returned for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

NAME

col - filter reverse line feeds

SYNTAX

col [-bfx]

DESCRIPTION

Col reads the standard input and writes the standard output. It performs the line overlays implied by reverse line feeds (ESC-7 in ASCII) and by forward and reverse half line feeds (ESC-9 and ESC-8). _Col_ is particularly useful for filtering multicolumn output made with the `.rt' command of _nroff_ and output resulting from use of the _tbl_(1) preprocessor.

Although _col_ accepts half line motions in its input, it normally does not emit them on output. Instead, text that would appear between lines is moved to the next lower full line boundary. This treatment can be suppressed by the -f (fine) option; in this case the output from _col_ may contain forward half line feeds (ESC-9), but will _still_ never contain either kind of reverse line motion.

If the -b option is given, _col_ assumes that the output device in use is not capable of backspacing. In this case, if several characters are to appear in the same place, only the last one read will be taken.

The control characters SO (ASCII code 017), and SI (016) are assumed to start and end text in an alternate character set. The character set (primary or alternate) associated with each printing character read is remembered; on output, SO and SI characters are generated where necessary to maintain the correct treatment of each character.

Col normally converts white space to tabs to shorten printing time. If the **-x** option is given, this conversion is suppressed.

All control characters are removed from the input except space, backspace, tab, return, newline, ESC (033) followed by one of 789, SI, SO, and VT (013). This last character is an alternate form of full reverse line feed, for compatibility with some other hardware conventions. All other nonprinting characters are ignored.

SEE ALSO

troff(1), tbl(1), greek(1)

NOTES

Can't back up more than 128 lines.
No more than 800 characters, including backspaces, on a line.

NAME

comm - select or reject lines common to two sorted files

SYNTAX

comm [- [123]] file1 file2

DESCRIPTION

Comm reads _file1_ and _file2_, which should be ordered in ASCII collating sequence, and produces a three column output: lines only in _file1_; lines only in _file2_; and lines in both files. The filename `-' means the standard input.

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus comm -12 prints only the lines common to the two files; comm -23 prints only lines in the first file but not in the second; comm -123 is a no-op.

SEE ALSO

cmp(1), diff(1), uniq(1)

NAME

copy - copy groups of files

SYNTAX

copy [option] ... source ... dest

DESCRIPTION

The copy command copies the contents of directories to another directory. It is possible to copy whole file systems since directories are made when needed.

If files, directories, or special files do not exist at the destination, then they are created with the same modes and flags of the source. In addition, the super-user may set the user and group ids. The owner and mode will not be changed if the destination file exists. Note that there may be more than one source directory. If so, then the effect is the same as if the copy command had been issued, each with only one source.

All of the options must be given as separate arguments and they may appear in any order even after the other arguments. The arguments are:

-a	Asks the user before attempting a copy. If the response does not begin with a 'y', then a copy will not be done. This option also sets the `-ad' flag.
-l	Uses links instead whenever they can be used. Otherwise a copy is done. Note that links are never done for special files or directories.
-n	Requires the destination file to be new. If not, then the copy command will not change the destination file. Of course the `-n' flag is meaningless for directories. For special files a `-n' flag is assumed (i.e., the destination of a special file must not exist).
-o	Only the super user may set this option. If set then every file copied will have its owner and group set to those of the source. If not set, then the owner will be that of the user who invoked the program.
-m	If set then every file copied will have its modification time and access time set to that of the source. If not set, then the modification time will be set to the time of the copy.
-r	If set, then every directory is recursively examined as it is encountered. If not set then any

directories that are found will be ignored.

-ad	Asks the user whether a `-r' flag applies when a directory is discovered. If the answer does not begin with a 'y', then the directory will be ignored.
-v	If the verbose option is set, then all kinds of messages will be printed that reveal what the program is doing.
source	This may be a file, directory or special file. It must exist. If it is not a directory, then the results of the command will be the same as for the cp command.
dest	The destination must be either a file or directory different from the source.

If the source and destination are anything but directories, then copy will act just like a cp command. If both are directories, then copy will copy each file into the destination directory according to the flags that have been set.

DIAGNOSTICS

Should be self-explanatory

NAME

cp - copy

SYNTAX

cp file1 file2

cp file ... directory

DESCRIPTION

File1 is copied onto *file2*. The mode and owner of *file2* are preserved if it already existed; the mode of the source file is used otherwise.

In the second form, one or more *files* are copied into the *directory* with their original file-names.

Cp refuses to copy a file onto itself.

SEE ALSO

cat(1), pr(1), mv(1), copy(1)

NAME

crypt - encode/decode

SYNTAX

crypt [password]

DESCRIPTION

Crypt reads from the standard input and writes on the standard output. The *password* is a key that selects a particular transformation. If no *password* is given, *crypt* demands a key from the terminal and turns off printing while the key is being typed in. *Crypt* encrypts and decrypts with the same key:

```
crypt key <clear >cypher
crypt key <cypher | pr
```

will print the clear.

Files encrypted by *crypt* are compatible with those treated by the editor *ed* in encryption mode.

The security of encrypted files depends on three factors: the fundamental method must be hard to solve; direct search of the key space must be infeasible; `sneak paths' by which keys or cleartext can become visible must be minimized.

Crypt implements a one-rotor machine designed along the lines of the German Enigma, but with a 256-element rotor. Methods of attack on such machines are known, but not widely; moreover the amount of work required is likely to be large.

The transformation of a key into the internal settings of the machine is deliberately designed to be expensive, i.e. to take a substantial fraction of a second to compute. However, if keys are restricted to (say) three lower-case letters, then encrypted files can be read by expending only a substantial fraction of five minutes of machine time.

Since the key is an argument to the *crypt* command, it is potentially visible to users executing *ps*(1) or a derivative. To minimize this possibility, *crypt* takes care to destroy any record of the key immediately upon entry. No doubt the choice of keys and key security are the most vulnerable aspect of *crypt*.

FILES

/dev/tty for typed key

SEE ALSO

ed(1), makekey(8)

NOTES

There is no warranty of merchantability nor any warranty of fitness for a particular purpose nor any other warranty, either express or implied, as to the accuracy of the enclosed materials or as to their suitability for any particular purpose. Accordingly, Bell Telephone Laboratories assumes no responsibility for their use by the recipient. Further, Bell Laboratories assumes no obligation to furnish any assistance of any kind whatsoever, or to furnish any additional information or documentation.

NAME

csh - a shell (command interpreter) with C-like syntax

SYNTAX

csh [-cefinstvVxX] [arg ...]

DESCRIPTION

Csh is a command language interpreter. It begins by executing commands from the file `.cshrc' in the home directory of the invoker. If this is a login shell then it also executes commands from the file `.login' there. In the normal case, the shell will then begin reading commands from the terminal, prompting with `% '. Processing of arguments and the use of the shell to process files containing command scripts will be described later.

The shell then repeatedly performs the following actions: a line of command input is read and broken into words. This sequence of words is placed on the command history list and then parsed. Finally each command in the current line is executed.

When a login shell terminates it executes commands from the file `.logout' in the users home directory.

Lexical structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters `&' `|' `;' `<' `>' `(' `)' form separate words. If doubled in `&&', `||', `<<' or `>>' these pairs form single words. These parser metacharacters may be made part of other words, or prevented their special meaning, by preceding them with `\'. A newline preceded by a `\' is equivalent to a blank.

In addition strings enclosed in matched pairs of quotations, `'', ``' or `"', form parts of a word; metacharacters in these strings, including blanks and tabs, do not form separate words. These quotations have semantics to be described subsequently. Within pairs of `' or `"' characters a newline preceded by a `\' gives a true newline character.

When the shell's input is not a terminal, the character `#' introduces a comment which continues to the end of the input line. It is prevented this special meaning when preceded by `\' and in quotations using ``', `'', and `"'.

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a sequence of simple commands separated by `|' characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines may be separated by `;', and are then executed sequentially. A sequence of pipelines may be executed without waiting for it to terminate by following it with an `&'. Such a sequence is automatically prevented from being terminated by a hangup signal; the _nohup_ command need not be used.

Any of the above may be placed in `(' `)' to form a simple command (which may be a component of a pipeline, etc.) It is also possible to separate pipelines with `||' or `&&' indicating, as in the C language, that the second is to be executed only if the first fails or succeeds respectively. (See _Expressions_.)

Substitutions

We now describe the various transformations the shell performs on the input in the order in which they occur.

History substitutions

History substitutions can be used to reintroduce sequences of words from previous commands, possibly performing modifications on these words. Thus history substitutions provide a generalization of a _redo_ function.

History substitutions begin with the character `!' and may begin anywhere in the input stream if a history substitution is not already in progress. This `!' may be preceded by an `\' to prevent its special meaning; a `!' is passed unchanged when it is followed by a blank, tab, newline, `=' or `('. History substitutions also occur when an input line begins with `↑'. This special abbreviation will be described later.

Any input line which contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution.

Commands input from the terminal which consist of one or more words are saved on the history list, the size of which is controlled by the _history_ variable. The previous command is always retained. Commands are numbered sequentially from 1.

For definiteness, consider the following output from the history command:

```
 9  write michael
10  ex write.c
11  cat oldwrite.c
12  diff *write.c
```

The commands are shown with their event numbers. It is not usually necessary to use event numbers, but the current event number can be made part of the prompt by placing an `!' in the prompt string.

With the current event 13 we can refer to previous events by event number `!11', relatively as in `!-2' (referring to the same event), by a prefix of a command word as in `!d' for event 12 or `!w' for event 9, or by a string contained in a word in the command as in `!?mic?' also referring to event 9. These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case `!!' refers to the previous command; thus `!!' alone is essentially a redo. The form `!#' references the current command (the one being typed in). It allows a word to be selected from further left in the line, to avoid retyping a long name, as in `!#:1'.

To select words from an event we can follow the event specification by a `:' and a designator for the desired words. The words of a input line are numbered from 0, the first (usually command) word being 0, the second word (first argument) being 1, etc. The basic word designators are:

0	first (command) word
n	n'th argument
↑	first argument, i.e. `1'
$	last argument
%	word matched by (immediately preceding) ?s? search
x-y	range of words
-y	abbreviates `0-y'
*	abbreviates `↑-$', or nothing if only 1 word in event
x*	abbreviates `x-$'
x-	like `x*' but omitting word `$'

The `:' separating the event specification from the word designator can be omitted if the argument selector begins with a `↑', `$', `*' `-' or `%'. After the optional word designator can be placed a sequence of modifiers, each preceded by a `:'. The following modifiers are defined:

h	Remove a trailing pathname component, leaving the h
r	Remove a trailing `.xxx' component, leaving the roo
s/l/r/	Substitute l for r
t	Remove all leading pathname components, leaving t
&	Repeat the previous substitution.
g	Apply the change globally, prefixing the above, e.g

p	Print the new command but do not execute it.
q	Quote the substituted words, preventing further sub
x	Like q, but break into words at blanks, tabs and ne

Unless preceded by a `g' the modification is applied only to the first modifiable word. In any case it is an error for no word to be applicable.

The left hand side of substitutions are not regular expressions in the sense of the editors, but rather strings. Any character may be used as the delimiter in place of `/'; a `\' quotes the delimiter into the _l_ and _r_ strings. The character `&' in the right hand side is replaced by the text from the left. A `\' quotes `&' also. A null _l_ uses the previous string either from a _l_ or from a contextual scan string _s_ in `!?_s_?'. The trailing delimiter in the substitution may be omitted if a newline follows immediately as may the trailing `?' in a contextual scan.

A history reference may be given without an event specification, e.g. `!$'. In this case the reference is to the previous command unless a previous history reference occurred on the same line in which case this form repeats the previous reference. Thus `!?foo?↑ !$' gives the first and last arguments from the command matching `?foo?'.

A special abbreviation of a history reference occurs when the first non-blank character of an input line is a `↑'. This is equivalent to `!:s↑' providing a convenient shorthand for substitutions on the text of the previous line. Thus `↑lb↑lib' fixes the spelling of `lib' in the previous command. Finally, a history substitution may be surrounded with `{' and `}' if necessary to insulate it from the characters which follow. Thus, after `ls -ld ~paul' we might do `!{l}a' to do `ls -ld ~paula', while `!la' would look for a command starting `la'.

Quotations with ' and "

The quotation of strings by `'' and `"' can be used to prevent all or some of the remaining substitutions. Strings enclosed in `'' are prevented any further interpretation. Strings enclosed in `"' are yet variable and command expanded as described below.

In both cases the resulting text becomes (all or part of) a single word; only in one special case (see _Command Substitition_ below) does a `"' quoted string yield parts of more than one word; `' quoted strings never do.

Alias substitution

The shell maintains a list of aliases which can be established, displayed and modified by the _alias_ and _unalias_ commands. After a command line is scanned, it is parsed into distinct commands and the first word of each command, left-to-right, is checked to see if it has an alias. If it does, then the text which is the alias for that command is reread with the history mechanism available as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

Thus if the alias for \`ls' is \`ls -l' the command \`ls /usr' would map to \`ls -l /usr', the argument list here being undisturbed. Similarly if the alias for \`lookup' was \`grep !↑ /etc/passwd' then \`lookup bill' would map to \`grep bill /etc/passwd'.

If an alias is found, the word transformation of the input text is performed and the aliasing process begins again on the reformed input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing. Other loops are detected and cause an error.

Note that the mechanism allows aliases to introduce parser metasyntax. Thus we can \`alias print 'pr \!* | lpr'' to make a command which _pr_'_s_ its arguments to the line printer.

Variable substitution

The shell maintains a set of variables, each of which has as value a list of zero or more words. Some of these variables are set by the shell or referred to by it. For instance, the _argv_ variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways.

The values of variables may be displayed and changed by using the _set_ and _unset_ commands. Of the variables referred to by the shell a number are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the _verbose_ variable is a toggle which causes command input to be echoed. The setting of this variable results from the -v command line option.

Other operations treat variables numerically. The \`@' command permits numeric calculations to be performed and the result assigned to a variable. Variable values are, however, always represented as (zero or more) strings. For the purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed keyed by \`$' characters. This expansion can be prevented by preceding the \`$' with a \`\' except within \`"'s where it always occurs, and within \`'s where it **never** occurs. Strings quoted by \`\`' are interpreted later (see <u>Command substitution</u> below) so \`$' substitution does not occur there until later, if at all. A \`$' is passed unchanged if followed by a blank, tab, or end-of-line.

Input/output redirections are recognized before variable expansion, and are variable expanded separately. Otherwise, the command name and entire argument list are expanded together. It is thus possible for the first (command) word to this point to generate more than one word, the first of which becomes the command name, and the rest of which become arguments.

Unless enclosed in \`"' or given the \`:q' modifier the results of variable substitution may eventually be command and filename substituted. Within \`"' a variable whose value consists of multiple words expands to a (portion of) a single word, with the words of the variables value separated by blanks. When the \`:q' modifier is applied to a substitution the variable will expand to multiple words with each word separated by a blank and quoted to prevent later command or filename substitution.

The following metasequences are provided for introducing variable values into the shell input. Except as noted, it is an error to reference a variable which is not set.

$name
${name}
    Are replaced by the words of the value of variable <u>name</u>, each separated by a blank. Braces insulate <u>name</u> from following characters which would otherwise be part of it. Shell variables have names consisting of up to 20 letters, digits, and underscores.

If <u>name</u> is not a shell variable, but is set in the environment, then that value is returned (but : modifiers and the other forms given below are not available in this case).

$name[selector]
${name[selector]}
    May be used to select only some of the words from the value of <u>name</u>. The selector is subjected to \`$' substitution and may consist of a single number or two numbers separated by a \`-'. The first word of a

variables value is numbered `1'. If the first number of a range is omitted it defaults to `1'. If the last member of a range is omitted it defaults to `$#name'. The selector `*' selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

$#name
${#name}
Gives the number of words in the variable. This is useful for later use in a `[selector]'.

$0
Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$number
${number}
Equivalent to `$argv[number]'.

$*
Equivalent to `$argv[*]'.

The modifiers `:h', `:t', `:r', `:q' and `:x' may be applied to the substitutions above as may `:gh', `:gt' and `:gr'. If braces `{' '}' appear in the command form then the modifiers must appear within the braces. The current implementation allows only one `:' modifier on each `$' expansion.

The following substitutions may not be modified with `:' modifiers.

$?name
${?name}
Substitutes the string `1' if name is set, `0' if it is not.

$?0
Substitutes `1' if the current input filename is know, `0' if it is not.

$$
Substitute the (decimal) process number of the (parent) shell.

Command and filename substitution

The remaining substitutions, command and filename substitution, are applied selectively to the arguments of builtin commands. This means that portions of expressions which are not evaluated are not subjected to these expansions. For commands which are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Command substitution

Command substitution is indicated by a command enclosed in \``'. The output from such a command is normally broken into separate words at blanks, tabs and newlines, with null words being discarded, this text then replacing the original string. Within \`"'s, only newlines force new words; blanks and tabs are preserved.

In any case, the single final newline does not force a new word. Note that it is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

Filename substitution

If a word contains any of the characters \`*', \`?', \`[' or \`{' or begins with the character \`~', then that word is a candidate for filename substitution, also known as \`globbing'. This word is then regarded as a pattern, and replaced with an alphabetically sorted list of file names which match the pattern. In a list of words specifying filename substitution it is an error for no pattern to match an existing file name, but it is not required for each pattern to match. Only the metacharacters \`*', \`?' and \`[' imply pattern matching, the characters \`~' and \`{' being more akin to abbreviations.

In matching filenames, the character \`.' at the beginning of a filename or immediately following a \`/', as well as the character \`/' must be matched explicitly. The character \`*' matches any string of characters, including the null string. The character \`?' matches any single character. The sequence \`[...]' matches any one of the characters enclosed. Within \`[...]', a pair of characters separated by \`-' matches any character lexically between the two.

The character \`~' at the beginning of a filename is used to refer to home directories. Standing alone, i.e. \`~' it expands to the invokers home directory as reflected in the value of the variable <u>home</u>. When followed by a name consisting of letters, digits and \`-' characters the shell searches for a user with that name and substitutes their home directory; thus \`~ken' might expand to \`/usr/ken' and \`~ken/chmach' to \`/usr/ken/chmach'. If the character \`~' is followed by a character other than a letter or \`/' or appears not at the beginning of a word, it is left undisturbed.

The metanotation `a{b,c,d}e' is a shorthand for `abe ace ade'. Left to right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct may be nested. Thus `~source/sl/{oldls,ls}.c' expands to `/usr/source/sl/oldls.c /usr/source/sl/ls.c' whether or not these files exist without any chance of error if the home directory for `source' is `/usr/source'. Similarly `../{memo,*box}' might expand to `../memo ../box ../mbox'. (Note that `memo' was not sorted with the results of matching `*box'.) As a special case `{', `}' and `{}' are passed undisturbed.

Input/output

The standard input and standard output of a command may be redirected with the following syntax:

< name
: Open file _name_ (which is first variable, command and filename expanded) as the standard input.

<< word
: Read the shell input up to a line which is identical to _word_. _Word_ is not subjected to variable, filename or command substitution, and each input line is compared to _word_ before any substitutions are done on this input line. Unless a quoting `\', `"', `'' or `\`' appears in _word_ variable and command substitution is performed on the intervening lines, allowing `\' to quote `$', `\' and `\`'. Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as standard input.

> name
>! name
>& name
>&! name
: The file _name_ is used as standard output. If the file does not exist then it is created; if the file exists, its is truncated, its previous contents being lost.

  If the variable _noclobber_ is set, then the file must not exist or be a character special file (e.g. a terminal or `/dev/null') or an error results. This helps prevent accidental destruction of files. In this case the `!' forms can be used and suppress this check.

  The forms involving `&' route the diagnostic output into the specified file as well as the standard output. Name is expanded in the same way as `<' input filenames are.

>> name
>>& name
>>! name
>>&! name
: Uses file name as standard output like `>' but places output at the end of the file. If the variable noclobber is set, then it is an error for the file not to exist unless one of the `!' forms is given. Otherwise similar to `>'.

If a command is run detached (followed by `&') then the default standard input for the command is the empty file `/dev/null'. Otherwise the command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The `<<' mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input.

Diagnostic output may be directed through a pipe with the standard output. Simply use the form `|&' rather than just `|'.

Expressions

A number of the builtin commands (to be described subsequently) take expressions, in which the operators are similar to those of C, with the same precedence. These expressions appear in the @, exit, if, and while commands. The following operators are available:

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

Here the precedence increases to the right, `==' and `!=', `<=' `>=' `<' and `>', `<<' and `>>', `+' and `-', `*' `/' and `%' being, in groups, at the same level. The `==' and `!=' operators compare their arguments as strings, all others operate on numbers. Strings which begin with `0' are considered octal numbers. Null or missing arguments are considered `0'. The result of all expressions are strings, which represent decimal numbers. It is important to note that no two components of an expression can appear in the same word; except when adjacent to components of expressions which are syntactically significant to the parser (`&' `|' `<' `>' `(' `)') they should be surrounded by spaces.

Also available in expressions as primitive operands are command executions enclosed in `{' and `}' and file enquiries of the form `-_l_ name' where _l_ is one of:

r	read access
w	write access
x	execute access
e	existence
o	ownership
z	zero size
f	plain file
d	directory

The specified name is command and filename expanded and then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible then all enquiries return false, i.e. `0'. Command executions succeed, returning true, i.e. `1', if the command exits with status 0, otherwise they fail, returning false, i.e. `0'. If more detailed status information is required then the command should be executed outside of an expression and the variable _status_ examined.

Control flow

The shell contains a number of commands which can be used to regulate the flow of control in command files (shell scripts) and (in limited but useful ways) from terminal input. These commands all operate by forcing the shell to reread or skip in its input and, due to the implementation, restrict the placement of some of the commands.

The _foreach_, _switch_, and _while_ statements, as well as the _if-then-else_ form of the _if_ statement require that the major keywords appear in a single simple command on an input line as shown below.

If the shell's input is not seekable, the shell buffers up input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward goto's will succeed on non-seekable inputs.)

Builtin commands

Builtin commands are executed within the shell. If a builtin command occurs as any component of a pipeline except the last then it is executed in a subshell.

alias
alias name
alias name wordlist

The first form prints all aliases. The second form prints the alias for name. The final form assigns the specified _wordlist_ as the alias of _name_; _wordlist_ is command and filename substituted. _Name_ is not allowed to be _alias_ or _unalias_

alloc

Shows the amount of dynamic core in use, broken down into used and free core, and address of the last location in the heap. With an argument shows each used and free block on the internal dynamic memory chain indicating its address, size, and whether it is used or free. This is a debugging command and may not work in production versions of the shell; it requires a modified version of the system memory allocator.

break

Causes execution to resume after the _end_ of the nearest enclosing _forall_ or _while_. The remaining commands on the current line are executed. Multi-level breaks are thus possible by writing them all on one line.

breaksw

Causes a break from a _switch_, resuming after the _endsw_.

case label:

A label in a _switch_ statement as discussed below.

cd
cd name
chdir
chdir name

Change the shells working directory to directory _name_. If no argument is given then change to the home directory of the user.

If _name_ is not found as a subdirectory of the current directory (and does not begin with \`/', \`./', or \`../'), then each component of the variable _cdpath_ is checked to see if it has a subdirectory _name_. Finally, if all else fails but _name_ is a shell variable whose value begins with \`/', then this is tried to see if it is a directory.

continue

Continue execution of the nearest enclosing _while_ or _foreach_. The rest of the commands on the current line are executed.

default:
: Labels the default case in a _switch_ statement. The default should come after all _case_ labels.

echo wordlist
: The specified words are written to the shells standard output. A `\c' causes the echo to complete without printing a newline, akin to the `\c' in _nroff_(1). A `\n' in wordlist causes a newline to be printed. Otherwise the words are echoed, separated by spaces.

else
end
endif
endsw
: See the description of the _foreach_, _if_, _switch_, and _while_ statements below.

exec command
: The specified command is executed in place of the current shell.

exit
exit(expr)
: The shell exits either with the value of the _status_ variable (first form) or with the value of the specified _expr_ (second form).

foreach name (wordlist)
  ...
end
: The variable _name_ is successively set to each member of _wordlist_ and the sequence of commands between this command and the matching _end_ are executed. (Both _foreach_ and _end_ must appear alone on separate lines.)

  The builtin command _continue_ may be used to continue the loop prematurely and the builtin command _break_ to terminate it prematurely. When this command is read from the terminal, the loop is read up once prompting with `?' before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal you can rub it out.

glob wordlist
: Like _echo_ but no `\' escapes are recognized and words are delimited by null characters in the output. Useful for programs which wish to use the shell to filename expand a list of words.

goto word
: The specified _word_ is filename and command expanded to

yield a string of the form `label'. The shell rewinds its input as much as possible and searches for a line of the form `label:' possibly preceded by blanks or tabs. Execution continues after the specified line.

history
Displays the history event list.

if (expr) command
If the specified expression evaluates true, then the single _command_ with arguments is executed. Variable substitution on _command_ happens early, at the same time it does for the _rest of_ the _if_ command. _Command_ must be a simple command, not a pipeline, a command list, or a parenthesized command list. Input/output redirection occurs even if _expr_ is false, when command is **not** executed (this is a bug).

if (expr) **then**
...
else if (expr2) **then**
...
else
...
endif
If the specified _expr_ is true then the commands to the first _else_ are executed; else if _expr2_ is true then the commands to the second else are executed, etc. Any number of _else-if_ pairs are possible; only one _endif_ is needed. The _else_ part is likewise optional. (The words _else_ and _endif_ must appear at the beginning of input lines; the _if_ must appear alone on its input line or after an _else_.)

login
Terminate a login shell, replacing it with an instance of **/bin/login**. This is one way to log off, included for compatibility with **/bin/sh**.

logout
Terminate a login shell. Especially useful if _ignoreeof_ is set.

nice
nice +number
nice command
nice +number command
The first form sets the _nice_ for this shell to 4. The second form sets the _nice_ to the given number. The final two forms run command at priority 4 and _number_ respectively. The super-user may specify negative niceness by using `nice -number ...'. Command is

always executed in a sub-shell, and the restrictions place on commands in simple _if_ statements apply.

nohup
nohup command

The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ignored. On the Computer Center systems at UC Berkeley, this also _submits_ the process. Unless the shell is running _detached_, _nohup_ has no effect. All processes detached with ``&'' are automatically _nohup'ed_. (Thus, _nohup_ is not really needed.)

onintr
onintr -
onintr label

Control the action of the shell on interrupts. The first form restores the default action of the shell on interrupts which is to terminate shell scripts or to return to the terminal command input level. The second form `onintr -' causes all interrupts to be ignored. The final form causes the shell to execute a `goto label' when an interrupt is received or a child process terminates because it was interrupted.

In any case, if the shell is running detached and interrupts are being ignored, all forms of _onintr_ have no meaning and interrupts continue to be _ignored_ by the shell and all invoked commands.

rehash

Causes the internal hash table of the contents of the directories in the _path_ variable to be recomputed. This is needed if new commands are added to directories in the _path_ while you are logged in. This should only be necessary if you add commands to one of your own directories, or if a systems programmer changes the contents of one of the system directories.

repeat count command

The specified _command_ which is subject to the same restrictions as the _command_ in the one line _if_ statement above, is executed _count_ times. I/O redirections occurs exactly once, even if _count_ is 0.

set
set name
set name=word
set name[index]=word
set name=(wordlist)

The first form of the command shows the value of all shell variables. Variables which have other than a single word as value print as a parenthesized word list. The second form sets name to the null string. The third form sets name to the single word. The fourth form sets the index'th component of name to word; this component must already exist. The final form sets name to the list of words in wordlist. In all cases the value is command and filename expanded.

These arguments may be repeated to set multiple values in a single set command. Note however, that variable expansion happens for all arguments before any setting occurs.

setenv name value
(Version 7 systems only.) Sets the value of environment variable name to be value, a single string. Useful environment variables are `TERM' the type of your terminal and `SHELL' the shell you are using.

shift
shift variable
The members of argv are shifted to the left, discarding argv[1]. It is an error for argv not to be set or to have less than one word as value. The second form performs the same function on the specified variable.

source name
The shell reads commands from name. Source commands may be nested; if they are nested too deeply the shell may run out of file descriptors. An error in a source at any level terminates all nested sour

during source

list.

```
switch (string)
case str1:
    ...
  breaksw
...
default:
    ...
  breaksw
endsw
```

Each case label is successively matched, against the specified string which is first command and filename expanded. The file metacharacters `*', `?' and `[...]' may be used in the case labels, which are variable expanded. If none of the labels match before a `default' label is found, then the execution begins after the default label. Each case label and the default label must appear nline.

The command _breaksw_ causes execution to continue after the _endsw_. Otherwise control may fall through case labels and default labels as in C. If no label matches and there is no default, execution continues after the _endsw_.

time
time command
With no argument, a summary of time used by this shell and its children is printed. If arguments are given the specified simple command is timed and a time summary as described under the _time_ variable is printed. If necessary, an extra shell is created to print the time statistic when the command completes.

umask
umask value
The file creation mask is displayed (first form) or set to the specified value (second form). The mask is given in octal. Common values for the mask are 002 giving all access to the group and read and execute access to others or 022 giving all access except no write access for users in the group or others.

unalias pattern
All aliases whose names match the specified pattern are discarded. Thus all aliases are removed by `unalias *'. It is not an error for nothing to be _unaliased_.

unhash
Use of the internal hash table to speed location of executed programs is disabled.

unset pattern
All variables whose names match the specified pattern are removed. Thus all variables are removed by `unset *'; this has noticeably distasteful side-effects. It is not an error for nothing to be _unset_.

wait
All child processes are waited for. It the shell is interactive, then an interrupt can disrupt the wait, at which time the shell prints names and process numbers of all children known to be outstanding.

while (expr)
...
end
While the specified expression evaluates non-zero, the commands between the _while_ and the matching end are evaluated. _Break_ and _continue_ may be used to terminate or continue the loop prematurely. (The _while_ and _end_

must appear alone on their input lines.) Prompting occurs here the first time through the loop as for the foreach statement if the input is a terminal.

@
@ name = expr
@ name[index] = expr

The first form prints the values of all the shell variables. The second form sets the specified name to the value of expr. If the expression contains \`<', \`>', \`&' or \`|' then at least this part of the expression must be placed within \`(' \`)'. The third form assigns the value of expr to the index'th argument of name. Both name and its index'th component must already exist.

The operators \`*=', \`+=', etc are available as in C. The space separating the name from the assignment operator is optional. Spaces are, however, mandatory in separating components of expr which would otherwise be single words.

Special postfix \`++' and \`--' operators increment and decrement name respectively, i.e. \`@ i++'.

Pre-defined variables

The following variables have special meaning to the shell. Of these, argv, child, home, path, prompt, shell and status are always set by the shell. Except for child and status this setting occurs only at initialization; these variables will not then be modified unless this is done explicitly by the user.

The shell copies the environment variable PATH into the variable path, and copies the value back into the environment whenever path is set. Thus is is not necessary to worry about its setting other than in the file .cshrc as inferior csh processes will import the definition of path from the environment. (It could be set once in the .login except that commands through net(1) would not see the definition.)

argv	Set to the arguments to the shell, it is from this variable that positional parameters are substituted, i.e. \`$1' is replaced by \`$argv[1]', etc.
cdpath	Gives a list of alternate directories searched to find subdirectories in chdir commands.
child	The process number printed when the last

command was forked with `&'. This variable is _unset_ when this process terminates.

echo	Set when the **-x** command line option is given. Causes each command and its arguments to be echoed just before it is executed. For non-builtin commands all expansions occur before echoing. Builtin commands are echoed before command and filename substitution, since these substitutions are then done selectively.
histchars	Can be assigned a two character string. The first character is used as a history character in place of ``!'', the second character is used in place of the ``^'' substitution mechanism. For example, ``set histchars=",;"'' will cause the history characters to be comma and semicolon.
history	Can be given a numeric value to control the size of the history list. Any command which has been referenced in this many events will not be discarded. Too large values of _history_ may run the shell out of memory. The last executed command is always saved on the history list.
home	The home directory of the invoker, initialized from the environment. The filename expansion of `~' refers to this variable.
ignoreeof	If set the shell ignores end-of-file from input devices which are terminals. This prevents shells from accidentally being killed by control-D's.
mail	The files where the shell checks for mail. This is done after each command completion which will result in a prompt, if a specified interval has elapsed. The shell says `You have new mail.' if the file exists with an access time not greater than its modify time. If the first word of the value of _mail_ is numeric it specifies a different mail checking interval, in seconds, than the default, which is 10 minutes. If multiple mail files are specified, then the shell says `New mail in _name_' when there is mail in the file _name_.

noclobber	As described in the section on *Input/output*, restrictions are placed on output redirection to insure that files are not accidentally destroyed, and that `>>' redirections refer to existing files.
noglob	If set, filename expansion is inhibited. This is most useful in shell scripts which are not dealing with filenames, or after a list of filenames has been obtained and further expansions are not desirable.
nonomatch	If set, it is not an error for a filename expansion to not match any existing files; rather the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, i.e. `echo [' still gives an error.
path	Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no *path* variable then only full path names will execute. The usual search path is `.', `/bin' and `/usr/bin', but this may vary from system to system. For the super-user the default search path is `/etc', `/bin' and `/usr/bin'. A shell which is given neither the **-c** nor the **-t** option will normally hash the contents of the directories in the *path* variable after reading *.cshrc*, and each time the *path* variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to give the *rehash* or the commands may not be found.
prompt	The string which is printed before each command is read from an interactive terminal input. If a `!' appears in the string it will be replaced by the current event number unless a preceding `\' is given. Default is `% ', or `# ' for the super-user.
shell	The file in which the shell resides. This is used in forking shells to interpret files which have execute bits set, but which are not executable by the system. (See the description of *Non-builtin Command Execution* below.) Initialized to the (system-dependent) home of the shell.

status	The status returned by the last command. If it terminated abnormally, then 0200 is added to the status. Builtin commands which fail return exit status \`1', all other builtin commands set status \`0'.
time	Controls automatic timing of commands. If set, then any command which takes more than this many cpu seconds will cause a line giving user, system, and real times and a utilization percentage which is the ratio of user plus system times to real time to be printed when it terminates.
verbose	Set by the **-v** command line option, causes the words of each command to be printed after history substitution.

Non-builtin command execution

When a command to be executed is found to not be a builtin command the shell attempts to execute the command via _exec_(2). Each word in the variable _path_ names a directory from which the shell will attempt to execute the command. If it is given neither a **-c** nor a **-t** option, the shell will hash the names in these directories into an internal table so that it will only try an _exec_ in a directory if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via _unhash_), or if the shell was given a **-c** or **-t** argument, and in any case for each directory component of _path_ which does not begin with a \`\`/'', the shell concatenates with the given command name to form a path name of a file which it then attempts to execute.

Parenthesized commands are always executed in a subshell. Thus \`(cd ; pwd) ; pwd' prints the _home_ directory; leaving you where you were (printing this _after_ the home directory), while \`cd ; pwd' leaves you in the _home_ directory. Parenthesized commands are most often used to prevent _chdir_ from affecting the current shell.

If the file has execute permissions but is not an executable binary to the system, then it is assumed to be a file containing shell commands an a new shell is spawned to read it.

If there is an _alias_ for _shell_ then the words of the alias will be prepended to the argument list to form the shell command. The first word of the _alias_ should be the full path name of the shell (e.g. \`$shell'). Note that this is a special, late occurring, case of _alias_ substitution, and only allows words to be prepended to the argument list without modification.

Argument list processing

If argument 0 to the shell is \`-' then this is a login shell. The flag arguments are interpreted as follows:

-c Commands are read from the (single) following argument which must be present. Any remaining arguments are placed in _argv_.

-e The shell exits if any invoked command terminates abnormally or yields a non-zero exit status.

-f The shell will start faster, because it will neither search for nor execute commands from the file \`.cshrc' in the invokers home directory.

-i The shell is interactive and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.

-n Commands are parsed, but not executed. This may aid in syntactic checking of shell scripts.

-s Command input is taken from the standard input.

-t A single line of input is read and executed. A \`\' may be used to escape the newline at the end of this line and continue onto another line.

-v Causes the _verbose_ variable to be set, with the effect that command input is echoed after history substitution.

-x Causes the _echo_ variable to be set, so that commands are echoed immediately before execution.

-V Causes the _verbose_ variable to be set even before \`.cshrc' is executed.

-X Is to **-x** as **-V** is to **-v**.

After processing of flag arguments if arguments remain but none of the **-c**, **-i**, **-s**, or **-t** options was given the first argument is taken as the name of a file of commands to be executed. The shell opens this file, and saves its name for possible resubstitution by \`$0'. Since many systems use either the standard version 6 or version 7 shells whose shell scripts are not compatible with this shell, the shell will execute such a `standard' shell if the first character of a script is not a `#', i.e. if the script does not start with a comment. Remaining arguments initialize the variable _argv_.

Signal handling

The shell normally ignores _quit_ signals. The _interrupt_ and _quit_ signals are ignored for an invoked command if the command is followed by `&'; otherwise the signals have the values which the shell inherited from its parent. The shells handling of interrupts can be controlled by _onintr_. Login shells catch the _terminate_ signal; otherwise this signal is passed on to children from the state in the shell's parent. In no case are interrupts allowed when a login shell is reading the file `.logout'.

AUTHOR

William Joy

FILES

~/.cshrc	Read at beginning of execution by each shell.
~/.login	Read by login shell, after `.cshrc' at login.
~/.logout	Read by login shell, at logout.
/bin/sh	Shell for scripts not starting with a `#'.
/tmp/sh*	Temporary file for `<<'.
/dev/null	Source of empty file.
/etc/passwd	Source of home directories for `~name'.

LIMITATIONS

Words can be no longer than 512 characters. The number of characters in an argument varies from system to system. Early version 6 systems typically have 512 character limits while later version 6 and version 7 systems have 5120 character limits. The number of arguments to a command which involves filename expansion is limited to 1/6'th the number of characters allowed in an argument list. Also command substitutions may substitute no more characters than are allowed in an argument list.

To detect looping, the shell restricts the number of _alias_ substititutions on a single line to 20.

SEE ALSO

access(2), exec(2), fork(2), pipe(2), signal(2), umask(2), wait(2), a.out(5), environ(5), `An introduction to the C shell'

NOTES

Control structure should be parsed rather than being recognized as built-in commands. This would allow control commands to be placed anywhere, to be combined with `|', and to be used with `&' and `;' metasyntax.

Commands within loops, prompted for by `?', are not placed in the _history_ list.

It should be possible to use the `:' modifiers on the output of command substitutions. All and more than one `:' modifier should be allowed on `$' substitutions.

Some commands should not touch _status_ or it may be so transient as to be almost useless. Oring in 0200 to _status_ on abnormal termination is a kludge.

In order to be able to recover from failing _exec_ commands on version 6 systems, the new command inherits several open files other than the normal standard input and output and diagnostic output. If the input and output are redirected and the new command does not close these files, some files may be held open unnecessarily.

There are a number of bugs associated with the importing/exporting of the PATH. For example, directories in the path using the ~ syntax are not expanded in the PATH. Unusual paths, such as (), can cause csh to core dump.

This version of _csh_ does not support or use the process control features of the 4th Berkeley Distribution. It contains a number of known bugs which have been fixed in the process control version. This version is not supported.

NAME

dc - desk calculator

SYNTAX

dc [file]

DESCRIPTION

Dc is an arbitrary precision arithmetic package. Ordinarily it operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained. The overall structure of *dc* is a stacking (reverse Polish) calculator. If an argument is given, input is taken from that file until its end, then from the standard input. The following constructions are recognized:

number
: The value of the number is pushed on the stack. A number is an unbroken string of the digits 0-9. It may be preceded by an underscore _ to input a negative number. Numbers may contain decimal points.

+ - / * % ^
: The top two values on the stack are added (+), subtracted (-), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. Any fractional part of an exponent is ignored.

s*x* The top of the stack is popped and stored into a register named *x*, where *x* may be any character. If the s is capitalized, *x* is treated as a stack and the value is pushed on it.

l*x* The value in register *x* is pushed on the stack. The register *x* is not altered. All registers start with zero value. If the l is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

d The top value on the stack is duplicated.

p The top value on the stack is printed. The top value remains unchanged. P interprets the top of the stack as an ascii string, removes it, and prints it.

f All values on the stack and in registers are printed.

q exits the program. If executing a string, the recursion level is popped by two. If q is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

x treats the top element of the stack as a character string and executes it as a string of dc commands.

X replaces the number on the top of the stack with its scale factor.

[...]
  puts the bracketed ascii string onto the top of the stack.

<_x_ >_x_ =_x_
  The top two elements of the stack are popped and compared. Register _x_ is executed if they obey the stated relation.

v replaces the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.

! interprets the rest of the line as a UNIX command.

c All values on the stack are popped.

i The top value on the stack is popped and used as the number radix for further input. I pushes the input base on the top of the stack.

o The top value on the stack is popped and used as the number radix for further output.

O pushes the output base on the top of the stack.

k the top of the stack is popped, and that value is used as a non-negative scale factor: the appropriate number of places are printed on output, and maintained during multiplication, division, and exponentiation. The interaction of scale factor, input base, and output base will be reasonable if all are changed together.

z The stack level is pushed onto the stack.

Z replaces the number on the top of the stack with its length.

? A line of input is taken from the input source (usually the terminal) and executed.

; : are used by _bc_ for array operations.

An example which prints the first ten values of n! is

```
        [lal+dsa*plal0>y]sy
        0sal
        lyx
```

SEE ALSO

bc(1), which is a preprocessor for dc providing infix notation and a C-like syntax which implements functions and reasonable control structures for programs.

DIAGNOSTICS

`x is unimplemented' where x is an octal number.
`stack empty' for not enough elements on the stack to do what was asked.
`Out of space' when the free list is exhausted (too many digits).
`Out of headers' for too many numbers being kept around.
`Out of pushdown' for too many items on the stack.
`Nesting Depth' for too many levels of nested execution.

NAME

dd - convert and copy a file

SYNTAX

dd [option=value] ...

DESCRIPTION

Dd copies the specified input file to the specified output with possible conversions. The standard input and output are used by default. The input and output block size may be specified to take advantage of raw physical I/O.

option	values
if=	input file name; standard input is default
of=	output file name; standard output is default
ibs=*n*	input block size *n* bytes (default 512)
obs=*n*	output block size (default 512)
bs=*n*	set both input and output block size, superseding *ibs* and *obs*; also, if no conversion is specified, it is particularly efficient since no copy need be done
cbs=*n*	conversion buffer size
skip=*n*	skip *n* input records before starting copy
files=*n*	copy *n* files from (tape) input
seek=*n*	seek *n* records from beginning of output file before copying
count=*n*	copy only *n* input records
conv=ascii	convert EBCDIC to ASCII
ebcdic	convert ASCII to EBCDIC
ibm	slightly different map of ASCII to EBCDIC
lcase	map alphabetics to lower case
ucase	map alphabetics to upper case
swab	swap every pair of bytes
noerror	do not stop processing on an error
sync	pad every input record to *ibs*
... , ...	several comma-separated conversions

Where sizes are specified, a number of bytes is expected. A number may end with **k, b** or **w** to specify multiplication by 1024, 512, or 2 respectively; a pair of numbers may be separated by **x** to indicate a product.

Cbs is used only if *ascii* or *ebcdic* conversion is specified. In the former case *cbs* characters are placed into the conversion buffer, converted to ASCII, and trailing blanks trimmed and new-line added before sending the line to the output. In the latter case ASCII characters are read into the conversion buffer, converted to EBCDIC, and blanks added to make up an output record of size *cbs*.

After completion, *dd* reports the number of whole and partial input and output blocks.

For example, to read an EBCDIC tape blocked ten 80-byte EBCDIC card images per record into the ASCII file *x*:

```
dd if=/dev/rmt0 of=x ibs=800 cbs=80 conv=ascii,lcase
```

Note the use of raw magtape. *Dd* is especially suited to I/O on the raw physical devices because it allows reading and writing in arbitrary record sizes.

To skip over a file before copying from magnetic tape do

```
(dd of=/dev/null; dd of=x) </dev/rmt0
```

SEE ALSO

cp(1), tr(1)

DIAGNOSTICS

f+p records in(out): numbers of full and partial records read(written)

NOTES

The ASCII/EBCDIC conversion tables are taken from the 256 character standard in the CACM Nov, 1968. The `ibm' conversion, while less blessed as a standard, corresponds better to certain IBM print train conventions. There is no universal solution.

Newlines are inserted only on conversion to ASCII; padding is done only on conversion to EBCDIC. These should be separate options.

NAME

df - disk free

SYNOPSIS

df [filesystem] ...

DESCRIPTION

Df prints out the number of free blocks available on the *filesystems*. If no file system is specified, the free space on all of the normally mounted file systems is printed.

FILES

Default file systems vary with installation.

SEE ALSO

icheck(1)

NAME

diff - differential file comparator

SYNTAX

diff [-efbh] file1 file2

DESCRIPTION

Diff tells what lines must be changed in two files to bring them into agreement. If *file1* (*file2*) is `-', the standard input is used. If *file1* (*file2*) is a directory, then a file in that directory whose file-name is the same as the file-name of *file2* (*file1*) is used. The normal output contains lines of these forms:

n1 a *n3*,*n4*
n1,*n2* d *n3*
n1,*n2* c *n3*,*n4*

These lines resemble *ed* commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging `a' for `d' and reading backward one may ascertain equally how to convert *file2* into *file1*. As in *ed*, identical pairs where *n1* = *n2* or *n3* = *n4* are abbreviated as a single number.

Following each of these lines come all the lines that are affected in the first file flagged by `<', then all the lines that are affected in the second file flagged by `>'.

The -b option causes trailing blanks (spaces and tabs) to be ignored and other strings of blanks to compare equal.

The -e option produces a script of *a*, *c* and *d* commands for the editor *ed*, which will recreate *file2* from *file1*. The -f option produces a similar script, not useful with *ed*, in the opposite order. In connection with -e, the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version *ed* scripts ($2,$3,...) made by *diff* need be on hand. A `latest version' appears on the standard output.

```
(shift; cat $*; echo '1,$p') | ed - $1
```

Except in rare circumstances, *diff* finds a smallest sufficient set of file differences.

Option -h does a fast, half-hearted job. It works only when changed stretches are short and well separated, but does work on files of unlimited length. Options -e and -f are unavailable with -h.

FILES

/tmp/d?????
/usr/lib/diffh for **-h**

SEE ALSO

cmp(1), comm(1), ed(1)

DIAGNOSTICS

Exit status is 0 for no differences, 1 for some, 2 for trouble.

NOTES

Editing scripts produced under the **-e** or **-f** option are naive about creating lines consisting of a single `.'.

NAME

diff3 - 3-way differential file comparison

SYNTAX

diff3 [-ex3] file1 file2 file3

DESCRIPTION

Diff3 compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====	all three files differ
====1	_file1_ is different
====2	_file2_ is different
====3	_file3_ is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

_f_ : _n1_ a	Text is to be appended after line number _n1_ in file _f_, where _f_ = 1, 2, or 3.
_f_ : _n1_ , _n2_ c	Text is to be changed in the range line _n1_ to line _n2_. If _n1_ = _n2_, the range may be abbreviated to _n1_.

The original contents of the range follows immediately after a **c** indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

Under the -e option, _diff3_ publishes a script for the editor _ed_ that will incorporate into _file1_ all changes between _file2_ and _file3_, _i.e._ the changes that normally would be flagged ==== and ====3. Option -x (-3) produces a script to incorporate only changes flagged ==== (====3). The following command will apply the resulting script to `file1'.

```
(cat script; echo '1,$p') | ed - file1
```

FILES

/tmp/d3?????
/usr/lib/diff3

SEE ALSO

diff(1)

NOTES

Text lines that consist of a single `.' will defeat -e.
Files longer than 64K bytes won't work.

NAME

du - summarize disk usage

SYNTAX

du [-s] [-a] [name ...]

DESCRIPTION

Du gives the number of blocks contained in all files and (recursively) directories within each specified directory or file *name*. If *name* is missing, `.' is used.

The optional argument -s causes only the grand total to be given. The optional argument -a causes an entry to be generated for each file. Absence of either causes an entry to be generated for each directory only.

A file which has two links to it is only counted once.

NOTES

Non-directories given as arguments (not under -a option) are not listed.
If there are too many distinct linked files, *du* counts the excess files multiply.

NAME

dump - incremental file system dump

SYNOPSIS

dump [key [argument ...] filesystem]

DESCRIPTION

Dump copies to magnetic tape all files changed after a certain date in the _filesystem_. The _key_ specifies the date and other options about the dump. The _key_ consists of characters from the set 0123456789fusd.

f   Place the dump on the next _argument_ file instead of the tape.

u   If the dump completes successfully, write the date of the beginning of the dump on file `/etc/ddate'. This file records a separate date for each filesystem and each dump level.

0-9 This number is the `dump level'. All files modified since the last date stored in the file `/etc/ddate' for the same filesystem at lesser levels will be dumped. If no date is determined by the level, the beginning of time is assumed; thus the option 0 causes the entire filesystem to be dumped.

s   The size of the dump tape is specified in feet. The number of feet is taken from the next _argument_. When the specified size is reached, the dump will wait for reels to be changed. The default size is 2300 feet.

d   The density of the tape, expressed in BPI, is taken from the next _argument_. This is used in calculating the amount of tape used per write. The default is 1600.

If no arguments are given, the _key_ is assumed to be 9u and the program attempts to dump the default filesystem to the default tape.

Now a short suggestion on how perform dumps. Start with a full level 0 dump

```
dump 0u
```

Next, periodic level 9 dumps should be made on an exponential progression of tapes. (Sometimes called Tower of Hanoi - 1 2 1 3 1 2 1 4 ... tape 1 used every other time, tape 2 used every fourth, tape 3 used every eighth, etc.)

```
dump 9u
```

When the level 9 incremental approaches a full tape (about 78000 blocks at 1600 BPI blocked 20), a level 1 dump should be made.

```
dump 1u
```

After this, the exponential series should progress as uninterrupted. These level 9 dumps are based on the level 1 dump which is based on the level 0 full dump. This progression of levels of dump can be carried as far as desired.

FILES

Default filesystem and tape vary with installation. For safety, however, we recommend that default disk filesystems not be used, as common operator errors can destroy that default disk.
/etc/ddate: record dump dates of filesystem/level.

SEE ALSO

restor(1), dump(5), dumpdir(1), sddate (1M)

DIAGNOSTICS

If the dump requires more than one tape, it will ask you to change tapes. Reply with a new-line when this has been done.

BUGS

Sizes are based on 1600 BPI blocked tape. The raw magtape device has to be used to approach these densities. Read errors on the filesystem are ignored. Write errors on the magtape are usually fatal.

NAME

dumpdir - print the names of files on a dump tape

SYNOPSIS

dumpdir [f filename]

DESCRIPTION

Dumpdir is used to read magtapes dumped with the _dump_ command and list the names and inode numbers of all the files and directories on the tape.

The f option causes _filename_ as the name of the tape instead of the default.

FILES

default tape unit varies with installation
rst*

SEE ALSO

dump(1), restor(1)

DIAGNOSTICS

If the dump extends over more than one tape, it may ask you to change tapes. Reply with a new-line when the next tape has been mounted.

BUGS

There is redundant information on the tape that could be used in case of tape reading problems. Unfortunately, _dumpdir_ doesn't use it.

NAME

echo - echo arguments

SYNTAX

echo [**-n**] [arg] ...

DESCRIPTION

Echo writes its arguments separated by blanks and terminated by a newline on the standard output. If the flag **-n** is used, no newline is added to the output.

Echo is useful for producing diagnostics in shell programs and for writing constant data on pipes. To send diagnostics to the standard error file, do \`echo ... 1>&2'.

NAME

ed - text editor

SYNTAX

ed [-] [-x] [name]

DESCRIPTION

Ed is the standard text editor.

If a *name* argument is given, *ed* simulates an *e* command (see below) on the named file; that is to say, the file is read into *ed's* buffer so that it can be edited. If -x is present, an *x* command is simulated first to handle an encrypted file. The optional - suppresses the printing of character counts by *e*, *r*, and *w* commands.

Ed operates on a copy of any file it is editing; changes made in the copy have no effect on the file until a *w* (write) command is given. The copy of the text being edited resides in a temporary file called the *buffer*.

Commands to *ed* have a simple and regular structure: zero or more *addresses* followed by a single character *command*, possibly *followed* by parameters to the command. These addresses specify one or more lines in the buffer. Missing addresses are supplied by default.

In general, only one command may appear on a line. Certain commands allow the addition of text to the buffer. While *ed* is accepting text, it is said to be in *input mode*. In this mode, no commands are recognized; all input is merely collected. Input mode is left by typing a period \`.' alone at the beginning of a line.

Ed supports a limited form of *regular expression* notation. A regular expression specifies a set of strings of characters. A member of this set of strings is said to be *matched* by the regular expression. In the following specification for regular expressions the word \`character' means any character but newline.

1. Any character except a special character matches itself. Special characters are the regular expression delimiter plus \\[. and sometimes ^*$.

2. A . matches any character.

3. A \\ followed by any character except a digit or () matches that character.

4. A nonempty string *s* bracketed [*s*] (or [^*s*]) matches any character in (or not in) *s*. In *s*, \\ has no special

meaning, and] may only appear as the first letter. A substring _a-b_, with _a_ and _b_ in ascending ASCII order, stands for the inclusive range of ASCII characters.

5. A regular expression of form 1-4 followed by * matches a sequence of 0 or more matches of the regular expression.

6. A regular expression, _x_, of form 1-8, bracketed \(_x_\) matches what _x_ matches.

7. A \ followed by a digit _n_ matches a copy of the string that the bracketed regular expression beginning with the _n_th \(matched.

8. A regular expression of form 1-8, _x_, followed by a regular expression of form 1-7, _y_ matches a match for _x_ followed by a match for _y_, with the _x_ match being as long as possible while still permitting a _y_ match.

9. A regular expression of form 1-8 preceded by ^ (or followed by $), is constrained to matches that begin at the left (or end at the right) end of a line.

10. A regular expression of form 1-9 picks out the longest among the leftmost matches in a line.

11. An empty regular expression stands for a copy of the last regular expression encountered.

Regular expressions are used in addresses to specify lines and in one command (see _s_ below) to specify a portion of a line which is to be replaced. If it is desired to use one of the regular expression metacharacters as an ordinary character, that character may be preceded by `\'. This also applies to the character bounding the regular expression (often `/') and to `\' itself.

To understand addressing in _ed_ it is necessary to know that at any time there is a _current line_. Generally speaking, the current line is the last line affected by a command; however, the exact effect on the current line is discussed under the description of the command. Addresses are constructed as follows.

1. The character `.' addresses the current line.

2. The character `$' addresses the last line of the buffer.

3. A decimal number _n_ addresses the _n_-th line of the buffer.

4. `'x' addresses the line marked with the name _x_, which must be a lower-case letter. Lines are marked with the _k_ command described below.

5. A regular expression enclosed in slashes `/' addresses the line found by searching forward from the current line and stopping at the first line containing a string that matches the regular expression. If necessary the search wraps around to the beginning of the buffer.

6. A regular expression enclosed in queries `?' addresses the line found by searching backward from the current line and stopping at the first line containing a string that matches the regular expression. If necessary the search wraps around to the end of the buffer.

7. An address followed by a plus sign `+' or a minus sign `-' followed by a decimal number specifies that address plus (resp. minus) the indicated number of lines. The plus sign may be omitted.

8. If an address begins with `+' or `-' the addition or subtraction is taken with respect to the current line; e.g. `-5' is understood to mean `.-5'.

9. If an address ends with `+' or `-', then 1 is added (resp. subtracted). As a consequence of this rule and rule 8, the address `-' refers to the line before the current line. Moreover, trailing `+' and `-' characters have cumulative effect, so `--' refers to the current line less 2.

10. To maintain compatibility with earlier versions of the editor, the character `^' in addresses is equivalent to `-'.

Commands may require zero, one, or two addresses. Commands which require no addresses regard the presence of an address as an error. Commands which accept one or two addresses assume default addresses when insufficient are given. If more addresses are given than such a command requires, the last one or two (depending on what is accepted) are used.

Addresses are separated from each other typically by a comma `,'. They may also be separated by a semicolon `;'. In this case the current line `.' is set to the previous address before the next address is interpreted. This feature can be used to determine the starting line for forward and backward searches (`/', `?'). The second address of any two-address sequence must correspond to a line following the line corresponding to the first address.

In the following list of ed commands, the default addresses are shown in parentheses. The parentheses are not part of the address, but are used to show that the given addresses are the default.

As mentioned, it is generally illegal for more than one command to appear on a line. However, most commands may be suffixed by \`p' or by \`l', in which case the current line is either printed or listed respectively in the way discussed below.

(.)a
<text>
.

The append command reads the given text and appends it after the addressed line. \`.' is left on the last line input, if there were any, otherwise at the addressed line. Address \`0' is legal for this command; text is placed at the beginning of the buffer.

(., .)c
<text>
.

The change command deletes the addressed lines, then accepts input text which replaces these lines. \`.' is left at the last line input; if there were none, it is left at the line preceding the deleted lines.

(., .)d
The delete command deletes the addressed lines from the buffer. The line originally after the last line deleted becomes the current line; if the lines deleted were originally at the end, the new last line becomes the current line.

e filename
The edit command causes the entire contents of the buffer to be deleted, and then the named file to be read in. \`.' is set to the last line of the buffer. The number of characters read is typed. \`filename' is remembered for possible use as a default file name in a subsequent r or w command. If \`filename' is missing, the remembered name is used.

E filename
This command is the same as e, except that no diagnostic results when no w has been given since the last buffer alteration.

f filename
The filename command prints the currently remembered file name. If \`filename' is given, the currently

remembered file name is changed to `filename'.

(1,$)g/regular expression/command list
In the global command, the first step is to mark every line which matches the given regular expression. Then for every such line, the given command list is executed with `.' initially set to that line. A single command or the first of multiple commands appears on the same line with the global command. All lines of a multi-line list except the last line must be ended with `\'. A, i, and c commands and associated input are permitted; the `.' terminating input mode may be omitted if it would be on the last line of the command list. The commands g and v are not permitted in the command list.

(.)i

<text>

.

This command inserts the given text before the addressed line. `.' is left at the last line input, or, if there were none, at the line before the addressed line. This command differs from the a command only in the placement of the text.

(., .+1)j
This command joins the addressed lines into a single line; intermediate newlines simply disappear. `.' is left at the resulting line.

(.)kx
The mark command marks the addressed line with name x, which must be a lower-case letter. The address form `'x' then addresses this line.

(., .)l
The list command prints the addressed lines in an unambiguous way: non-graphic characters are printed in two-digit octal, and long lines are folded. The l command may be placed on the same line after any non-i/o command.

(., .)ma
The move command repositions the addressed lines after the line addressed by a. The last of the moved lines becomes the current line.

(., .)p
The print command prints the addressed lines. `.' is left at the last line printed. The p command may be placed on the same line after any non-i/o command.

(., .)P
: This command is a synonym for p.

q
: The quit command causes ed to exit. No automatic write of a file is done.

Q
: This command is the same as q, except that no diagnostic results when no w has been given since the last buffer alteration.

($)r filename
: The read command reads in the given file after the addressed line. If no file name is given, the remembered file name, if any, is used (see e and f commands). The file name is remembered if there was no remembered file name already. Address `0' is legal for r and causes the file to be read at the beginning of the buffer. If the read is successful, the number of characters read is typed. `.' is left at the last line read in from the file.

(., .)s/regular expression/replacement/ or,
(., .)s/regular expression/replacement/g
: The substitute command searches each addressed line for an occurrence of the specified regular expression. On each line in which a match is found, all matched strings are replaced by the replacement specified, if the global replacement indicator `g' appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. It is an error for the substitution to fail on all addressed lines. Any character other than space or new-line may be used instead of `/' to delimit the regular expression and the replacement. `.' is left at the last line substituted.

: An ampersand `&' appearing in the replacement is replaced by the string matching the regular expression. The special meaning of `&' in this context may be suppressed by preceding it by `\'. The characters `\n' where n is a digit, are replaced by the text matched by the n-th regular subexpression enclosed between `\(' and `\)'. When nested, parenthesized subexpressions are present, n is determined by counting occurrences of `\(' starting from the left.

: Lines may be split by substituting new-line characters into them. The new-line in the replacement string must be escaped by preceding it by `\'.

(., .)ta
: This command acts just like the m command, except that

a copy of the addressed lines is placed after address _a_ (which may be 0). `.' is left on the last line of the copy.

(., .)u
The undo command restores the preceding contents of the current line, which must be the last line in which a substitution was made.

(1, $)v/regular expression/command list
This command is the same as the global command _g_ except that the command list is executed _g_ with `.' initially set to every line _except_ those matching the regular expression.

(1, $)w filename
The write command writes the addressed lines onto the given file. If the file does not exist, it is created mode 666 (readable and writable by everyone). The file name is remembered if there was no remembered file name already. If no file name is given, the remembered file name, if any, is used (see _e_ and _f_ commands). `.' is unchanged. If the command is successful, the number of characters written is printed.

(1,$)W filename
This command is the same as _w_, except that the addressed lines are appended to the file.

x A key string is demanded from the standard input. Later _r_, _e_ and _w_ commands will encrypt and decrypt the text with this key by the algorithm of _crypt_(1). An explicitly empty key turns off encryption.

($)= The line number of the addressed line is typed. `.' is unchanged by this command.

!<shell command>
The remainder of the line after the `!' is sent to _sh_(1) to be interpreted as a command. `.' is unchanged.

(.+1)<newline>
An address alone on a line causes the addressed line to be printed. A blank line alone is equivalent to `.+1p'; it is useful for stepping through text.

If an interrupt signal (ASCII DEL) is sent, _ed_ prints a `?' and returns to its command level.

Some size limitations: 512 characters per line, 256 characters per global command list, 64 characters per file name, and 128K characters in the temporary file. The limit on the number of lines depends on the amount of core: each line takes 1 word.

When reading a file, _ed_ discards ASCII NUL characters and all characters after the last newline. It refuses to read files containing non-ASCII characters.

FILES

/tmp/e*
ed.hup: work is saved here if terminal hangs up

SEE ALSO

B. W. Kernighan, _A Tutorial Introduction to the ED Text Editor_
B. W. Kernighan, _Advanced editing on UNIX_
sed(1), crypt(1)

DIAGNOSTICS

\`?name' for inaccessible file; \`?' for errors in commands; \`?TMP' for temporary file overflow.

To protect against throwing away valuable work, a _q_ or _e_ command is considered to be in error, unless a _w_ has occurred since the last buffer change. A second _q_ or _e_ will be obeyed regardless.

NOTES

The _l_ command mishandles DEL.
A ! command cannot be subject to a _g_ command.
Because 0 is an illegal address for a _w_ command, it is not possible to create an empty file with _ed_.

NAME

edit - text editor (variant of the ex editor for new or casual users)

SYNTAX

edit [-r] name ...

DESCRIPTION

Edit is a variant of the text editor _ex_ recommended for new or casual users who wish to use a command oriented editor. The following brief introduction should help you get started with _edit_. A more complete basic introduction is provided by _Edit: A tutorial_ . A _Ex/edit command summary_ (_version 2.0_) is also very useful. See _ex_(UCB) for other useful documents; in particular, if _you_ are using a CRT terminal you will want to learn about the display editor _vi_.

BRIEF INTRODUCTION

To edit the contents of an existing file you begin with the command ``edit name'' to the shell. _Edit_ makes a copy of the file which you can then edit, and tells you how many lines and characters are in the file. To create a new file, just make up a name for the file and try to run _edit_ on it; you will cause an error diagnostic, but don't worry.

Edit prompts for commands with the character `:', which you should see after starting the editor. If you are editing an existing file, then you will have some lines in _edit's_ buffer (its name for the copy of the file you are editing). Most commands to _edit_ use its ``current line'' if you don't tell them which line to use. Thus if you say **print** (which can be abbreviated **p**) and hit carriage return (as you should after all _edit_ commands) this current line will be printed. If you **delete** (**d**) the current line, _edit_ will print the new current line. When you start editing, _edit_ makes the last line of the file the current line. If you **delete** this last line, then the new last line becomes the current one. In general, after a **delete**, the next line in the file becomes the current line. (Deleting the last line is a special case.)

If you start with an empty file, or wish to add some new lines, then the **append** (**a**) command can be used. After you give this command (typing a carriage return after the word append) _edit_ will read lines from your terminal until you give a line consisting of just a ``.'', placing these lines after the current line. The last line you type then becomes the current line. The command **insert** (**i**) is like **append** but places the lines you give before, rather than after, the current line.

Edit numbers the lines in the buffer, with the first line having number 1. If you give the command ``1'' then _edit_ will type this first line. If you then give the command **delete** _edit_ will delete the first line, and line 2 will become _line_ 1, and _edit_ will print the current line (the new line 1) so you can _see_ where you are. In general, the current line will always be the last line affected by a command.

You can make a change to some text within the current line by using the **substitute (s)** command. You say ``s/_old_/_new_/'' where _old_ is replaced by the old characters you want to get rid of and _new_ is the new characters you want to replace it with.

The command **file** (f) will tell you how many lines there are in the buffer you are editing and will say ``[Modified]'' if you have changed it. After modifying a file you can put the buffer text back to replace the file by giving a **write** (w) command. You can then leave the editor by issuing a **quit** (q) command. If you run _edit_ on a file, but don't change it, it is not necessary (_but_ does no harm) to **write** the file back. If you try to **quit** from _edit_ after modifying the buffer without writing it out, _you_ will be warned that there has been ``No **write** since last change'' and _edit_ will await another command. If you wish not to **write** _the_ buffer out then you can issue another **quit** command. The buffer is then irretrievably discarded, and you return to the shell.

By using the **delete** and **append** commands, and giving line numbers to see lines in the file you can make any changes you desire. You should learn at least a few more things, however, if you are to use _edit_ more than a few times.

The **change** (c) command will change the current line to a sequence of lines you supply (as in **append** you give lines up to a line consisting of only a ``.''). You can tell **change** to change more than one line by giving the line numbers of the lines you want to change, i.e. ``3,5change''. You can print lines this way too. Thus ``1,23p'' prints the first 23 lines of the file.

The **undo** (u) command will reverse the effect of the last command you gave which changed the buffer. Thus if give a **substitute** command which doesn't do what you want, you can say **undo** and the old contents of the line will be restored. You can also **undo** an **undo** command so that you can continue to change your mind. _Edit_ will give you a warning message when commands you do affect more than one line of the buffer. If the amount of change seems unreasonable, you should consider doing an _undo_ and looking to see what happened. If you decide that the change is ok, then you can undo again to get it back. Note that commands such as write and quit cannot be undone.

To look at the next line in the buffer you can just hit carriage return. To look at a number of lines hit ^D (control key and, while it is held down D key, then let up both) rather than carriage return. This will show you a half screen of lines on a CRT or 12 lines on a hardcopy terminal. You can look at the text around where you are by giving the command ``z.''. The current line will then be the last line printed; you can get back to the line where you were before the ``z.'' command by saying ``''''. The **z** command can also be given other following characters ``z-'' prints a screen of text (or 24 lines) ending where you are; ``z+'' prints the next screenful. If you want less than a screenful of lines do, e.g., ``z.12'' to get 12 lines total. This method of giving counts works in general; thus you can delete 5 lines starting with the current line with the command ``delete 5''.

To find things in the file you can use line numbers if you happen to know them; since the line numbers change when you insert and delete lines this is somewhat unreliable. You can search backwards and forwards in the file for strings by giving commands of the form **/text/** to search forward for text or ?text? to search backward for text. If a search reaches the end of the file without finding the text it wraps, end around, and continues to search back to the line where you are. A useful feature here is a search of the form **/^text/** which searches for text at the beginning of a line. Similarly **/text$/** searches for text at the end of a line. You can leave off the trailing / or ? in these commands.

The current line has a symbolic name ``.''; this is most useful in a range of lines as in ``.,$print'' which prints the rest of the lines in the file. To get to the last line in the file you can refer to it by its symbolic name ``$''. Thus the command ``$ delete'' or ``$d'' deletes the last line in the file, no matter which line was the current line before. Arithmetic with line references is also possible. Thus the line ``$-5'' is the fifth before the last, and ``.+20'' is 20 lines after the present.

You can find out which line you are at by doing ``.=''. This is useful if you wish to move or copy a section of text within a file or between files. Find out the first and last line numbers you wish to copy or move (say 10 to 20). For a move you can then say ``10,20move "a'' which deletes these lines from the file and places them in a buffer named a. Edit has 26 such buffers named a through z. You can later get these lines back by doing ``"a move .'' to put the contents of buffer a after the current line. If you want to move or copy these lines between files you can give an **edit** (**e**) command after copying the lines, following it with the name of the other file you wish to edit, i.e. ``edit chapter2''. By changing move to copy above you can get a pattern for copying lines. If the text you wish to move or copy is all within one file then you can just say ``10,20move $'' for example. It is not necessary to use named buffers in this case (but you can if you wish).

SEE ALSO

ex (UCB), vi (UCB), `Edit: A tutorial', by Ricki Blau and James Joyce

AUTHOR

William Joy

NOTES

See ex(UCB).

NAME

ex - text editor

SYNTAX

ex [-] [-v] [-t tag] [-r] [+*lineno*] name ...

DESCRIPTION

Ex is the root of a family of editors: *edit*, *ex* and *vi*. *Ex* is a superset of *ed*, with the most notable extension being a display editing facility. Display based editing is the focus of *vi*.

If you have not used *ed*, or are a casual user, you will find that the editor *edit* is convenient for you. It avoids some of the complexities of *ex* used mostly by systems programmers and persons very familiar with *ed*.

If you have a CRT terminal, you may wish to use a display based editor; in this case see *vi*(UCB), which is a command which focuses on the display editing portion of *ex*.

DOCUMENTATION

For *edit* and *ex* see the *Ex/edit command summary - Version 2.0*. The document *Edit: A tutorial* provides a comprehensive introduction to *edit* assuming no previous knowledge of computers or the UNIX system.

The *Ex Reference Manual - Version 2.0* is a comprehensive and complete manual for the command mode features of *ex*, but you cannot learn to use the editor by reading it. For an introduction to more advanced forms of editing using the command mode of *ex* see the editing documents written by Brian Kernighan for the editor *ed*; the material in the introductory and advanced documents works also with *ex*.

An Introduction to Display Editing with Vi introduces the display editor *vi* and provides reference material on *vi*. The *Vi Quick Reference* card summarizes the commands of *vi* in a useful, functional way, and is useful with the *Introduction*.

FOR ED USERS

If you have used *ed* you will find that *ex* has a number of new features useful on CRT terminals. Intelligent terminals and high speed terminals are very pleasant to use with *vi*. Generally, the editor uses far more of the capabilities of terminals than *ed* does, and uses the terminal capability data base *termcap*(UCB) and the type of the terminal you are using from the variable TERM in the environment to determine how to drive your terminal efficiently. The editor makes use of features such as insert and delete character and line in its visual command (which can be abbreviated **vi**) and which is the central mode of editing when using *vi*(UCB).

There is also an interline editing **open** (o) command which works on all terminals.

Ex contains a number of new features for easily viewing the text of the file. The **z** command gives easy access to windows of text. Hitting ^D causes the editor to scroll a half-window of text and is more useful for quickly stepping through a file than just hitting return. Of course, the screen oriented **visual** mode gives constant access to editing context.

Ex gives you more help when you make mistakes. The **undo** (u) command allows you to reverse any single change which goes astray. *Ex* gives you a lot of feedback, normally printing changed lines, and indicates when more than a few lines are affected by a command so that it is easy to detect when a command has affected more lines than it should have.

The editor also normally prevents overwriting existing files unless you edited them so that you don't accidentally clobber with a *write* a file other than the one you are editing. If the system (or editor) crashes, or you accidentally hang up the phone, you can use the editor **recover** command to retrieve your work. This will get you back to within a few lines of where you left off.

Ex has several features for dealing with more than one file at a time. You can give it a list of files on the command line and use the **next** (n) command to deal with each in turn. The **next** command can also be given a list of file names, or a pattern as used by the shell to specify a new set of files to be dealt with. In general, filenames in the editor may be formed with full shell metasyntax. The metacharacter `%' is also available in forming filenames and is replaced by the name of the current file. For editing large groups of related files you can use *ex's* **tag** command to quickly locate functions and other important points in any of the files. This is useful when working on a large program when you want to quickly find the definition of a particular function. The command *ctags*(UCB) builds a *tags* file or a group of C programs.

For moving text between files and within a file the editor has a group of buffers, named *a* through *z*. You can place text in these named buffers and carry it over when you edit another file.

There is a command **&** in *ex* which repeats the last **substitute** command. In addition there is a confirmed substitute command. You give a range of substitutions to be done and the editor interactively asks whether each substitution is desired.

You can use the **substitute** command in _ex_ to systematically convert the case of letters between upper and lower case. It is possible to ignore case of letters in searches and substitutions. _Ex_ also allows regular expressions which match words to be constructed. This is convenient, for example, in searching for the word ``edit'' if your document also contains the word ``editor.''

_Ex_ has a set of _options_ which you can set to tailor it to your liking. One option which is very useful is the _autoindent_ option which allows the editor to automatically _supply leading_ white space to align text. You can then use the ^D key as a backtab and space and tab forward to align new code easily.

Miscellaneous new useful features include an intelligent join (j) command which supplies white space between joined lines automatically, commands < and > which shift groups of lines, and the ability to filter portions of the buffer through commands such as _sort_.

FILES

/usr/lib/ex2.0strings	error messages
/usr/lib/ex2.0recover	recover command
/usr/lib/ex2.0preserve	preserve command
/etc/termcap	describes capabilities of terminals
~/.exrc	editor startup file
/tmp/Ex_nnnnn_	editor temporary
/tmp/Rx_nnnnn_	named buffer temporary
/usr/preserve	preservation directory

SEE ALSO

awk(1), ed(1), grep(1), sed(1), edit(UCB), grep(UCB), termcap(UCB), vi(UCB)

AUTHOR

William Joy

NOTES

The _undo_ command causes all marks to be lost on lines _changed_ and then restored if the marked lines were changed.

Undo never clears the buffer modified condition.

The _z_ command prints a number of logical rather than physical lines. More than a screen full of output may result if long lines are present.

File input/output errors don't print a name if the command line `-' option is used.

There is no easy way to do a single scan ignoring case.

Because of the implementation of the arguments to _next_, only 512 bytes of argument list are allowed there.

The format of /_etc_/_termcap_ and the large number of capabilities of terminals used by the editor cause terminal type setup to be rather slow.

The editor does not warn if text is placed in named buffers and not used before exiting the editor.

Null characters are discarded in input files, and cannot appear in resultant files.

NAME

expr - evaluate arguments as an expression

SYNTAX

expr arg ...

DESCRIPTION

The arguments are taken as an expression. After evaluation, the result is written on the standard output. Each token of the expression is a separate argument.

The operators and keywords are listed below. The list is in order of increasing precedence, with equal precedence operators grouped.

expr | _expr_
: yields the first _expr_ if it is neither null nor `0', otherwise yields the second _expr_.

expr & _expr_
: yields the first _expr_ if neither _expr_ is null or `0', otherwise yields `0'.

expr _relop_ _expr_
: where _relop is one of_ < <= = != >= >, yields `1' if the indicated comparison is true, `0' if false. The comparison is numeric if both _expr_ are integers, otherwise lexicographic.

expr + _expr_
expr - _expr_
: addition or subtraction of the arguments.

expr * _expr_
expr / _expr_
expr % _expr_
: multiplication, division, or remainder of the arguments.

expr : _expr_
: The matching operator compares the string first argument with the regular expression second argument; regular expression syntax is the same as that of _ed_(1). The \(...\) pattern symbols can be used to select a portion of the first argument. Otherwise, the matching operator yields the number of characters matched (`0' on failure).

(_expr_)
: parentheses for grouping.

Examples:

To add 1 to the Shell variable _a_:

```
a=`expr $a + 1`
```

To find the filename part (least significant part) of the pathname stored in variable _a_, which may or may not contain `/':

```
expr $a : '.*/\(.*\)' '|' $a
```

Note the quoted Shell metacharacters.

SEE ALSO

ed(1), sh(1), test(1)

DIAGNOSTICS

Expr returns the following exit codes:

```
0    if the expression is neither null nor `0',
1    if the expression is null or `0',
2    for invalid expressions.
```

NAME

factor, primes - factor a number, generate large primes

SYNTAX

factor [number]

primes

DESCRIPTION

When _factor_ is invoked without an argument, it waits for a number to be typed in. If you type in a positive number less than 2^{56} (about 7.2e16) it will factor the number and print its prime factors; each one is printed the proper number of times. Then it waits for another number. It exits if it encounters a zero or any non-numeric character.

If _factor_ is invoked with an argument, it factors the number as above and then exits.

Maximum time to factor is proportional to sqrt(n) and occurs when _n_ is prime or the square of a prime. It takes 1 minute to factor a prime near 10^{14} on a PDP11.

When _primes_ is invoked, it waits for a number to be typed in. If you type in a positive number less than 2^{56} it will print all primes greater than or equal to this number.

DIAGNOSTICS

`Ouch.' for input out of range or for garbage input.

NAME

file - determine file type

SYNTAX

file filename ...

file -f fileofnames

DESCRIPTION

File performs a series of tests on each argument in an attempt to classify it. If an argument appears to be ascii, _file_ examines the first 512 bytes and tries to guess its language.

If the first argument is a -_f_ flag, _file_ will take the list of filenames from the file.

For a.out files, the relationship between flags to cc and the file classification is:

cc flag	classification
i	separate
n	pure
s	not "not stripped"
Z	23fixed

NOTES

It often makes mistakes. In particular it often suggests that command files are C programs. Also, programs that begin with comments are described as English text.

NAME

find - find files

SYNTAX

find pathname-list expression

DESCRIPTION

Find recursively descends the directory hierarchy for each pathname in the _pathname-list_ (i.e., one or more pathnames) seeking files that match a boolean _expression_ written in the primaries given below. In the descriptions, the argument _n_ is used as a decimal integer where +_n_ means more than _n_, -_n_ means less than _n_ and _n_ means exactly _n_.

-name filename
: True if the _filename_ argument matches the current file name. Normal Shell argument syntax may be used if escaped (watch out for \`[', \`?' and \`*').

-perm onum
: True if the file permission flags exactly match the octal number _onum_ (see _chmod_(1)). If _onum_ is prefixed by a minus sign, more flag bits (017777, see _stat_(2)) become significant and the flags are compared: (_flags_&_onum_)==_onum_.

-type c
: True if the type of the file is _c_, where _c_ is b, c, d or f for block special file, character special file, directory or plain file.

-links n
: True if the file has _n_ links.

-user uname
: True if the file belongs to the user _uname_ (login name or numeric user ID).

-group gname
: True if the file belongs to group _gname_ (group name or numeric group ID).

-size n
: True if the file is _n_ blocks long (512 bytes per block).

-inum n
: True if the file has inode number _n_.

-atime n
: True if the file has been accessed in _n_ days.

-mtime n
: True if the file has been modified in _n_ days.

-exec command
: True if the executed command returns a zero value as exit status. The end of the command must be

punctuated by an escaped semicolon. A command argument `{}' is replaced by the current pathname.

-ok command
: Like **-exec** except that the generated command is written on the standard output, then the standard input is read and the command executed only upon response **y**.

-print
: Always true; causes the current pathname to be printed.

-newer file
: True if the current file has been modified more recently than the argument file.

The primaries may be combined using the following operators (in order of decreasing precedence):

1) A parenthesized group of primaries and operators (parentheses are special to the Shell and must be escaped).

2) The negation of a primary (`!' is the unary not operator).

3) Concatenation of primaries (the and operation is implied by the juxtaposition of two primaries).

4) Alternation of primaries (`**-o**' is the or operator).

EXAMPLE

To remove all files named `a.out' or `*.o' that have not been accessed for a week:

```
find / \( -name a.out -o -name '*.o' \) -atime +7 -exec rm
{} \;
```

FILES

/etc/passwd
/etc/group

SEE ALSO

sh(1), test(1), filsys(5)

NOTES

The syntax is painful.

NAME

finger - user information lookup program

SYNTAX

finger [options] name ...

DESCRIPTION

By default *finger* lists the login name, full name, terminal name and write status (as a '*' before the terminal name if write permission is denied), idle time, login time, and office location and phone number (if they are known) for each current UNIX user. (Idle time is minutes if it is a single integer, hours and minutes if a ':' is present, or days and hours if a 'd' is present.)

A longer format also exists and is used by *finger* whenever a list of peoples names is given. (Account names as well as first and last names of users are accepted.) This format is multi-line, and includes all the information described above as well as the user's home directory and login shell, any plan which the person has placed in the file .*plan* in their home directory, and the project on which they are working from the file .*project* also in the home directory.

Finger options include:

-f Suppress the printing of the header line (short format).

-l Force long output format.

-p Suppress printing of the .*plan* files

-s Force short output format.

FILES

/etc/utmp	who file
/etc/passwd	for users names, offices, phones, directories and shells
/usr/adm/lastlog	last login times
~/.plan	plans
~/.project	projects

SEE ALSO

w(1), who(1)

AUTHOR

Earl T. Cohen

NOTES

Only the first line of the .*project* file is printed.

The encoding of the gecos field is UCB dependent - it knows that an office `197MC' is `197M Cory Hall', and tht `529BE' is `529B Evans Hall'.

NAME

grep, egrep, fgrep - search a file for a pattern

SYNTAX

grep [option] ... expression [file] ...

egrep [option] ... [expression] [file] ...

fgrep [option] ... [strings] [file]

DESCRIPTION

Commands of the _grep_ family search the input _files_ (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output; unless the -h flag is used, the file name is shown if there is more than one input file.

Grep patterns are limited regular expressions in the style of _ed_(1); it uses a compact nondeterministic algorithm. _Egrep_ patterns are full regular expressions; it uses a fast deterministic algorithm that sometimes needs exponential space. _Fgrep_ patterns are fixed strings; it is fast and compact.

The following options are recognized.

-v All lines but those matching are printed.

-c Only a count of matching lines is printed.

-l The names of files with matching lines are listed (once) separated by newlines.

-n Each line is preceded by its line number in the file.

-b Each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

-s No output is produced, only status.

-h Do not print filename headers with output lines.

-y Alphabetic letters in the pattern will match letters of either case in the input (_grep_ and _fgrep_ only).

-e _expression_
Same as a simple _expression_ argument, but useful when the _expression_ begins with a -.

-f _file_
The regular expression (_egrep_) or string list (_fgrep_)

is taken from the _file_.

-x (Exact) only lines matched in their entirety are printed (_fgrep_ only).

Care should be taken when using the characters $ * [^ | ? ' " ( ) and \ in the _expression_ as they are also meaningful to the Shell. It is safest to enclose the entire _expression_ argument in single quotes ' '.

Fgrep searches for lines that contain one of the (newline-separated) _strings_.

Egrep accepts extended regular expressions. In the following description `character' excludes newline:

A \ followed by a single character matches that character.

The character ^ ($) matches the beginning (end) of a line.

A . matches any character.

A single character not otherwise endowed with special meaning matches that character.

A string enclosed in brackets [] matches any single character from the string. Ranges of ASCII character codes may be abbreviated as in `a-z0-9'. A] may occur only as the first character of the string. A literal - must be placed where it can't be mistaken as a range indicator.

A regular expression followed by * (+, ?) matches a sequence of 0 or more (1 or more, 0 or 1) matches of the regular expression.

Two regular expressions concatenated match a match of the first followed by a match of the second.

Two regular expressions separated by | or newline match either a match for the first or a match for the second.

A regular expression enclosed in parentheses matches a match for the regular expression.

The order of precedence of operators at the same parenthesis level is [] then *+? then concatenation then | and newline.

SEE ALSO

ed(1), sed(1), sh(1)

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files.

NOTES

Ideally there should be only one *grep*, but we don't know a single algorithm that spans a wide enough range of space-time tradeoffs.

Lines are limited to 256 characters; longer lines are truncated.

NAME

head - give first few lines of a stream

SYNTAX

head [-count] [file ...]

DESCRIPTION

This filter gives the first _count_ lines of each of the specified files, or of the standard input. If _count_ is omitted it defaults to 10.

SEE ALSO

tail(1)

AUTHOR

Bill Joy

NAME

join - relational database operator

SYNTAX

join [options] file1 file2

DESCRIPTION

Join forms, on the standard output, a join of the two relations specified by the lines of *file1* and *file2*. If *file1* is `-', the standard input is used.

File1 and *file2* must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined, normally the first in each line.

There is one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line normally consists of the common field, then the rest of the line from *file1*, then the rest of the line from *file2*.

Fields are normally separated by blank, tab or newline. In this case, multiple separators count as one, and leading separators are discarded.

These options are recognized:

-a*n* In addition to the normal output, produce a line for each unpairable line in file *n*, where *n* is 1 or 2.

-e *s* Replace empty output fields by string *s*.

-j*n* *m*
Join on the *m*th field of file *n*. If *n* is missing, use the *m*th field in each file.

-o *list*
Each output line comprises the fields specifed in *list*, each element of which has the form *n*.*m*, where *n* is a file number and *m* is a field number.

-t*c* Use character *c* as a separator (tab character). Every appearance of *c* in a line is significant.

SEE ALSO

sort(1), comm(1), awk(1)

NOTES

With default field separation, the collating sequence is that of *sort* -*b*; with -t, the sequence is that of a plain sort.

The conventions of _join_, _sort_, _comm_, _uniq_, _look_ and _awk_(1) are wildly incongruous.

NAME

kill - terminate a process with extreme prejudice

SYNTAX

kill [-signo] processid ...

DESCRIPTION

Kill sends signal 15 (terminate) to the specified processes. If a signal number preceded by \`-' is given as first argument, that signal is sent instead of terminate (see *signal*(2)). This will kill processes that do not catch the signal; in particular \`kill -9 ...' is a sure kill.

By convention, if process number 0 is specified, all members in the process group (i.e. processes resulting from the current login) are signaled.

The killed processes must belong to the current user unless he is the super-user. To shut the system down and bring it up single user the super-user may use \`kill -1 1'; see *init*(8).

The process number of an asynchronous process started with \`&' is reported by the shell. Process numbers can also be found by using *ps*(1).

SEE ALSO

ps(1), kill(2), signal(2)

NAME

ld - loader

SYNTAX

ld [option] file ...

DESCRIPTION

_Ld_ combines several object programs into one, resolves external references, and searches libraries. In the simplest case several object _files_ are given, and _ld_ combines them, producing an object _module_ which can be _either_ executed or become the input for a further _ld_ run. (In the latter case, the -r option must be _given_ to preserve the relocation bits.) The output of _ld_ is left on **a.out**. This file is made executable only if _no_ errors occurred during the load.

The argument routines are concatenated in the order specified. The entry point of the output is the beginning of the first routine.

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. If a routine from a library references another routine in the library, and the library has not been processed by _ranlib_(1), the referenced routine must appear after the referencing routine in the library. Thus the order of programs within libraries may be important. If the first member of a library is named `__.SYMDEF', then it is understood to be a dictionary for the library such as produced by _ranlib_; the dictionary is searched iteratively to satisfy as many references as possible.

The symbols `_etext', `_edata' and `_end' (`etext', `edata' and `end' in _C_) are reserved, and if referred to, are set to the first location above the program, the first location above initialized data, and the first location above all data respectively. It is erroneous to define these symbols.

_Ld_ understands several options. Except for **-l**, they should appear before the file names.

-s `Strip' the output, that is, remove the symbol table and relocation bits to save space (but impair the usefulness of the debugger). This information can also be removed by _strip_(1).

-u Take the following argument as a symbol and enter it as undefined in the symbol table. This is useful for loading wholly from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

-l*x* This option is an abbreviation for the library name \`/lib/lib*x*.a', where *x* is a string. If that does not exist, *ld* tries \`/usr/lib/lib*x*.a'. A library is searched when its name is encountered, so the placement of a -l is significant.

-x Do not preserve local (non-.globl) symbols in the output symbol table; only enter external symbols. This option saves some space in the output file.

-X Save local symbols except for those whose names begin with \`L'. This option is used by *cc*(1) to discard internally generated labels while retaining symbols local to routines.

-r Generate relocation bits in the output file so that it can be the subject of another *ld* run. This flag also prevents final definitions from being given to common symbols, and suppresses the \`undefined symbol' diagnostics.

-d Force definition of common storage even if the -r flag is present.

-n Arrange that when the output file is executed, the text portion will be read-only and shared among all users executing the file. This involves moving the data areas up to the first possible 4K word boundary following the end of the text.

-i When the output file is executed, the program text and data areas will live in separate address spaces. The only difference between this option and -n is that here the data starts at location 0.

-o The *name* argument after -o is used as the name of the *ld* output file, instead of a.out.

-e The following argument is taken to be the name of the entry point of the loaded program; location 0 is the default.

-O This is an overlay file, only the text segment will be replaced by *exec*(2). Shared data must have the same layout as in the program overlaid.

-D The next argument is a decimal number that sets the size of the data segment.

FILES

/lib/lib*.a	libraries
/usr/lib/lib*.a	more libraries
a.out	output file

SEE ALSO

as(1), ar(1), cc(1), ranlib(1)

NOTES

NAME

lex - generator of lexical analysis programs

SYNTAX

lex [-tvfn] [file] ...

DESCRIPTION

Lex generates programs to be used in simple lexical analyis of text. The input files (standard input default) contain regular expressions to be searched for, and actions written in C to be executed when expressions are found.

A C source program, `lex.yy.c' is generated, to be compiled thus:

```
cc lex.yy.c -lln
```

This program, when run, copies unrecognized portions of the input to the output, and executes the associated C action for each regular expression that is recognized.

The following lex program converts upper case to lower, removes blanks at the end of lines, and replaces multiple blanks by single blanks.

```
%%
[A-Z] putchar(yytext[0]+'a'-'A');
[ ]+$
[ ]+  putchar(' ');
```

The options have the following meanings.

-t Place the result on the standard output instead of in file `lex.yy.c'.

-v Print a one-line summary of statistics of the generated analyzer.

-n Opposite of **-v**; **-n** is default.

-f `Faster' compilation: don't bother to pack the resulting tables; limited to small programs.

SEE ALSO

yacc(1)
M. E. Lesk and E. Schmidt, LEX - Lexical Analyzer Generator

NAME

lint - a C program verifier

SYNTAX

lint [**-abchnpuvx**] file ...

DESCRIPTION

Lint attempts to detect features of the C program *files* which are likely to be bugs, or non-portable, or wasteful. It also checks the type usage of the program more strictly than the compilers. Among the things which are currently found 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.

By default, it is assumed that all the *files* are to be loaded together; they are checked for mutual compatibility. Function definitions for certain libraries are available to *lint*; these libraries are referred to by a conventional name, such as `-lm', in the style of *ld*(1).

Any number of the options in the following list may be used. The -D, -U, and -I options of *cc*(1) are also recognized as separate arguments.

p Attempt to check portability to the *IBM* and *GCOS* dialects of C.

h Apply a number of heuristic tests to attempt to intuit bugs, improve style, and reduce waste.

b Report *break* statements that cannot be reached. (This is not the default because, unfortunately, most *lex* and many *yacc* outputs produce dozens of such comments.)

v Suppress complaints about unused arguments in functions.

x Report variables referred to by extern declarations, but never used.

a Report assignments of long values to int variables.

c Complain about casts which have questionable portability.

u Do not complain about functions and variables used and not defined, or defined and not used (this is suitable for running _lint_ on a subset of files out of a larger program).

n Do not check compatibility against the standard library.

Exit(2) and other functions which do not return are not understood; this causes various lies.

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.

/*NOSTRICT*/
shuts off strict type checking in the next expression.

/*ARGSUSED*/
turns on the -v option for the next function.

/*LINTLIBRARY*/
at the beginning of a file shuts off complaints about unused functions in this file.

FILES
/usr/lib/lint[12] programs
/usr/lib/llib-lc declarations for standard functions
/usr/lib/llib-port declarations for portable functions

SEE ALSO
cc(1)
S. C. Johnson, _Lint, a C Program Checker_

NAME

ln - make a link

SYNTAX

ln name1 [name2]

DESCRIPTION

A link is a directory entry referring to a file; the same file (together with its size, all its protection information, etc.) may have several links to it. There is no way to distinguish a link to a file from its original directory entry; any changes in the file are effective independently of the name by which the file is known.

_Ln_ creates a link to an existing file _name1_. If _name2_ is given, the link has that name; otherwise it is placed in the current directory and its name is the last component of _name1_.

It is forbidden to link to a directory or to link across file systems.

SEE ALSO

rm(1)

NAME

look - find lines in a sorted list

SYNTAX

look [-df] string [file]

DESCRIPTION

Look consults a sorted _file_ and prints all lines that begin with _string_. It uses binary search.

The options **d** and **f** affect comparisons as in _sort_(1):

d `Dictionary' order: only letters, digits, tabs and blanks participate in comparisons.

f Fold. Upper case letters compare equal to lower case.

If no _file_ is specified, /_usr_/_dict_/_words_ is assumed with collating sequence -df.

FILES

/usr/dict/words

SEE ALSO

sort(1), grep(1)

NAME
lookall - look through all text files on UNIX

SYNTAX
lookall [-C*n*]

DESCRIPTION
Lookall accepts keywords from the standard input, performs a search similar to that of *refer*(1), and writes the result on the standard output. *Lookall* consults, however, an index to all the text files on the system rather than just bibliographies. Only the first 50 words of each file (roughly) were used to make the indexes. Blank lines are taken as delimiters between queries.

The -C*n* option specifies a coordination level search: up to *n* keywords may be missing from the answers, and the answers are listed with those containing the most keywords first.

The command sequence in /*usr*/*dict*/*lookall*/*makindex* regenerates the index.

FILES
The directory /*usr*/*dict*/*lookall* contains the index files.

DIAGNOSTICS
`Warning: index precedes file ...' means that a file has been changed since the index was made and it may be retrieved (or not retrieved) erroneously.

NOTES
Coordination level searching doesn't work as described: only those acceptable items with the smallest number of missing keywords are retreived.

NAME

lorder - find ordering relation for an object library

SYNTAX

lorder file ...

DESCRIPTION

The input is one or more object or library archive (see _ar_(1)) _files_. The standard output is a list of pairs of object file names, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by _tsort_(1) to find an ordering of a library suitable for one-pass access by _ld_(1).

This brash one-liner intends to build a new library from existing `.o' files.

```
ar cr library `lorder *.o | tsort`
```

FILES

*symref, *symdef
nm(1), sed(1), sort(1), join(1)

SEE ALSO

tsort(1), ld(1), ar(1)

NOTES

The names of object files, in and out of libraries, must end with `.o'; nonsense results otherwise.

NAME

lpr, vpr - line printer spooler

SYNTAX

```
lpr [ option ] ... [ file ] ...
vpr [ -b banner ] [ file ]
```

DESCRIPTION

Lpr causes the *files* to be queued for printing on a line printer. If no files are named, the standard input is read. The following options are available:

-r Remove the file when it has been queued.

-c Copy the file to insulate against changes that may happen before printing.

-m Report by *mail*(*1*) when printing is complete.

-n Do not report by mail. This is the default option.

Vpr is the program used by *lpr* when the online printer is a Versatec machine to insert an identification *banner* before the output, strip overstrikes, and, where possible, remove blank lines to make 66-line pages fit on 64 lines. If the file */usr/adm/vpacct* is writable, *vpr* places accounting information on it.

FILES

/usr/spool/lpd/lock
/usr/spool/lpd/cf* data file
/usr/spool/lpd/df* daemon control file
/usr/spool/lpd/tf* temporary version of control file
/usr/bin/vpr for Versatec printer
/usr/adm/vpacct

SEE ALSO

lpd(8)

NAME

ls - list contents of directory

SYNTAX

ls [**-ltasdrucifg**] name ...

DESCRIPTION

For each directory argument, _ls_ lists the contents of the directory; for each file argument, _ls_ repeats its name and any other information requested. The output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments appear before directories and their contents. There are several options:

-l List in long format, giving mode, number of links, owner, size in bytes, and time of last modification for each file. (See below.) If the file is a special file the size field will instead contain the major and minor device numbers.

-t Sort by time modified (latest first) instead of by name, as is normal.

-a List all entries; usually \`.' and \`..' are suppressed.

-s Give size in blocks, including indirect blocks, for each entry.

-d If argument is a directory, list only its name, not its contents (mostly used with -l to get status on directory).

-r Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

-u Use time of last access instead of last modification for sorting (-t) or printing (-l).

-c Use time of last modification to inode (mode, etc.) instead of last modification to file for sorting (-t) or printing (-l).

-i Print i-number in first column of the report for each file listed.

-f Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off -l, -t, -s, and -r, and turns on -a; the order is the order in which entries appear in the directory.

-g   Give group ID instead of owner ID in long listing.

The mode printed under the -l option contains 11 characters which are interpreted as follows: the first character is

d	if the entry is a directory;
b	if the entry is a block-type special file;
c	if the entry is a character-type special file;
-	if the entry is a plain file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions; the next to permissions to others in the same user-group; and the last to all others. Within each set the three characters indicate permission respectively to read, to write, or to execute the file as a program. For a directory, `execute' permission is interpreted to mean permission to search the directory for a specified file. The permissions are indicated as follows:

r	if the file is readable;
w	if the file is writable;
x	if the file is executable;
-	if the indicated permission is not granted.

The group-execute permission character is given as s if the file has set-group-ID mode; likewise the user-execute permission character is given as s if the file has set-user-ID mode.

The last character of the mode (normally `x' or `-') is t if the 1000 bit of the mode is on. See chmod(1) for the meaning of this mode.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks is printed.

FILES

/etc/passwd to get user ID's for `ls -l'.
/etc/group to get group ID's for `ls -g'.

NAME

m4 - macro processor

SYNTAX

m4 [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 arguments, or if an argument is \`-', the standard input is read. The processed text is written on the standard output.

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 quotes (\`') are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. 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.
undefine	removes the definition of the macro named in its argument.
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 _unix_ is predefined on UNIX versions of _m4_.

changequote	Change quote characters to the first and second arguments. _Changequote_ without arguments restores the original values (i.e., \`').
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.
eval	evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include +, -, *, /, %, ^ (exponentiation); relationals; parentheses.
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	is identical to _include_, except that it says nothing if the file is inaccessible.
syscmd	executes the UNIX command given in the first argument. No value is returned.
maketemp	fills in a string of XXXXX in its argument with the current process id.
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.

SEE ALSO

B. W. Kernighan and D. M. Ritchie, _The M4 Macro Processor_

NAME

mail - send or receive mail among users

SYNTAX

mail person ...
mail [**-r**] [**-q**] [**-p**] [**-f** file]

DESCRIPTION

Mail with no argument prints a user's mail, message-by-message, in last-in, first-out order; the optional argument -r causes first-in, first-out order. If the -p flag is given, the mail is printed with no questions asked; otherwise, for each message, *mail* reads a line from the standard input to direct disposition of the message.

newline
Go on to next message.

d Delete message and go on to the next.

p Print message again.

- Go back to previous message.

s [*file*] ...
Save the message in the named *files* (`mbox' default).

w [*file*] ...
Save the message, without a header, in the named *files* (`mbox' default).

m [*person*] ...
Mail the message to the named *persons* (yourself is default).

EOT (control-D)
Put unexamined mail back in the mailbox and stop.

q Same as EOT.

x Exit, without changing the mailbox file.

!command
Escape to the Shell to do command.

? Print a command summary.

An interrupt stops the printing of the current letter. The optional argument -q causes *mail* to exit after interrupts without changing the mailbox.

When _persons_ are named, _mail_ takes the standard input up to an end-of-file (or a line with just `.') and adds it to each _person's_ `mail' file. The message is preceded by the sender's name and a postmark. Lines that look like postmarks are prepended with `>'. A _person_ is usually a user name recognized by _login_(1). To denote a recipient on a remote system, prefix _person_ by the system name and exclamation mark (see _uucp_(1)).

The -f option causes the named file, e.g. `mbox', to be printed as if it were the mail file.

Each user owns his own mailbox, which is by default generally readable but not writable. The command does not delete an empty mailbox nor change its mode, so a user may make it unreadable if desired.

When a user logs in he is informed of the presence of mail.

FILES

/usr/spool/mail/*	mailboxes
/etc/passwd	to identify sender and locate persons
mbox	saved mail
/tmp/ma*	temp file
dead.letter	unmailable text
uux(1)	

SEE ALSO

xsend(1), write(1), uucp(1)

NOTES

There is a locking mechanism intended to prevent two senders from accessing the same mailbox, but it is not perfect and races are possible.

NAME

make - maintain program groups

SYNTAX

make [**-f** makefile] [option] ... file ...

DESCRIPTION

Make executes commands in _makefile_ to update one or more target _names_. _Name_ is typically a program. If no **-f** option is present, `makefile' and `Makefile' are tried in order. If _makefile_ is `-', the standard input is taken. More than one **-f** option may appear

Make updates a target if it depends on prerequisite files that have been modified since the target was last modified, or if the target does not exist.

Makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated list of targets, then a colon, then a list of prerequisite files. Text following a semicolon, and all following lines that begin with a tab, are shell commands to be executed to update the target.

Sharp 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 `.c' files and a common file `incl'.

```
pgm: a.o b.o
     cc a.o b.o -lm -o pgm
a.o: incl a.c
     cc -c a.c
b.o: incl b.c
     cc -c b.c
```

Makefile entries of the form

```
stringl = string2
```

are macro definitions. Subsequent appearances of $(_stringl_) are replaced by _string2_. If _stringl_ is a single character, the parentheses are optional.

Make infers prerequisites for files for which _makefile_ gives no construction commands. For example, a `.c' file may be inferred as prerequisite for a `.o' file and be compiled to produce the `.o' file. Thus the preceding example can be done more briefly:

```
pgm: a.o b.o
     cc a.o b.o -lm -o pgm
a.o b.o: incl
```

Prerequisites are inferred according to selected suffixes listed as the \`prerequisites' for the special name \`.SUFFIXES'; multiple lists accumulate; an empty list clears what came before. Order is significant; the first possible name for which both a file and a rule as described in the next paragraph exist is inferred. The default list is

```
.SUFFIXES: .out .o .c .e .r .f .y .l .s
```

The rule to create a file with suffix _s2_ that depends on a similarly named file with suffix _s1_ is specified as an entry for the \`target' _s1s2_. In such an entry, the special macro $* stands for the target name with suffix deleted, $@ for the full target name, $< for the complete list of prerequisites, and $? for the list of prerequisites that are out of date. For example, a rule for making optimized \`.o' files from \`.c' files is

```
.c.o: ; cc -c -O -o $@ $*.c
```

Certain macros are used by the default inference rules to communicate optional arguments to any resulting compilations. In particular, \`CFLAGS' is used for _cc_ and _f77_(1) options, \`LFLAGS' and \`YFLAGS' for _lex_ and _yacc_(1) options.

Command lines are executed one at a time, each by its own shell. A line is printed when it is executed unless the special target \`.SILENT' is in _makefile_, or the first character of the command is \`@'.

Commands returning nonzero status (see _intro_(1)) cause _make_ to terminate unless the special target \`.IGNORE' is in _makefile_ or the command begins with <tab><hyphen>.

Interrupt and quit cause the target to be deleted unless the target depends on the special name \`.PRECIOUS'.

Other options:

-i Equivalent to the special entry \`.IGNORE:'.

-k When a command returns nonzero status, abandon work on the current entry, but continue on branches that do not depend on the current entry.

-n Trace and print, but do not execute the commands needed to update the targets.

-t Touch, i.e. update the modified date of targets, without executing any commands.

-r Equivalent to an initial special entry `.SUFFIXES:' with no list.

-s Equivalent to the special entry `.SILENT:'.

FILES

makefile, Makefile

SEE ALSO

sh(1), touch(1)
S. I. Feldman *Make - A Program for Maintaining Computer Programs*

NOTES

Some commands return nonzero status inappropriately. Use -i to overcome the difficulty.
Commands that are directly executed by the shell, notably *cd*(1), are ineffectual across newlines in *make*.

NAME

man - print sections of this manual

SYNTAX

man [option ...] [chapter] title ...

DESCRIPTION

Man locates and prints the section of this manual named *title* in the specified *chapter*. (In this context, the word `page' is often used as a synonym for `section'.) The *title* is entered in lower case. The *chapter* number does not need a letter suffix. If no *chapter* is specified, the whole manual is searched for *title* and all occurrences of it are printed.

Options and their meanings are:

-t Phototypeset the section using *troff*(1).

-n Print the section on the standard output using *nroff*(1).

-k Display the output on a Tektronix 4014 terminal using *troff*(1) and *tc*(1).

-e Appended or prefixed to any of the above causes the manual section to be preprocessed by *neqn* or *eqn*(1); -e alone means -te.

-w Print the path names of the manual sections, but do not print the sections themselves.

(default)
Copy an already formatted manual section to the terminal, or, if none is available, act as -n. It may be necessary to use a filter to adapt the output to the particular terminal's characteristics.

Further *options*, e.g. to specify the kind of terminal you have, are passed on to *troff*(1) or *nroff*. *Options* and *chapter* may be changed before each *title*.

For example:

```
man man
```

would reproduce this section, as well as any other sections named *man* that may exist in other chapters of the manual, e.g. *man*(7).

FILES

/usr/man/man?/*

/usr/man/cat?/*

SEE ALSO

nroff(1), eqn(1), tc(1), man(7)

NOTES

The manual is supposed to be reproducible either on a phototypesetter or on a terminal. However, on a terminal some information is necessarily lost.

NAME

mesg - permit or deny messages

SYNTAX

mesg [n] [y]

DESCRIPTION

Mesg with argument **n** forbids messages via *write*(1) by revoking non-user write permission on the user's terminal. *Mesg* with argument **y** reinstates permission. All by itself, *mesg* reports the current state without changing it.

FILES

/dev/tty*
/dev

SEE ALSO

write(1)

DIAGNOSTICS

Exit status is 0 if messages are receivable, 1 if not, 2 on error.

NAME
mkconf - generate configuration tables

SYNOPSIS
/sys/conf/mkconf

DESCRIPTION
Mkconf examines a machine configuration table on its standard input. Its output is three files; *l.s*, *c.c* and *mch0.s*. *L.s* is an assembler program that represents the interrupt vectors located in low memory addresses and the device register addresses. *C.c* contains initialized block and character device switch tables, a switch table for line protocols and declarations of various configuration dependent and parameterized variables. *Mch0.s* contains conditional assembly switches which define the tape controller to be used for system crash dumps.

Input to *mkconf* is a sequence of lines. The following describe devices on the machine:

lp	(LP11)
rf	(RS11)
tc	(TU56)
rk	(RK03/RK05)
tm	(TU10/TE10)
rp	(RP03)
hp	(RP04/5/6/RM02/3)
ht	(TU16/TE16)
ts	(TS11)
rx	(RX01/2)
hk	(RK06/7)
rl	(RL01/2)
dc*	(DC11)
kl*	(KL11/DL11-ABC)
dl*	(DL11-E)
dn*	(DN11)
dh*	(DH11)
dhdm*	(DM11-BB)
du*	(DU11)
dz*	(DZ11)

The devices marked with * may be preceded by a number telling how many are to be included. The console typewriter is automatically included; don't count it as part of the KL or DL specification. Count DN's in units of 4 (1 system unit).

The following lines are also accepted.

root *dev minor*
The specified block device (e.g. hp) is used for the root. *minor* is a decimal number giving the minor

NAME

mkdir - make a directory

SYNTAX

mkdir dirname ...

DESCRIPTION

Mkdir creates specified directories in mode 777. Standard entries, `.', for the directory itself, and `..' for its parent, are made automatically.

Mkdir requires write permission in the parent directory.

SEE ALSO

rm(1)

DIAGNOSTICS

Mkdir returns exit code 0 if all directories were successfully made. Otherwise it prints a diagnostic and returns nonzero.

NAME

mkfs - construct a file system

SYNOPSIS

/etc/mkfs special proto [m n]

DESCRIPTION

Mkfs constructs a file system by writing on the special file *special* according to the directions found in the prototype file *proto*. The prototype file contains tokens separated by spaces or new lines. The first token is the name of a file to be copied onto block zero as the bootstrap program, see *bproc*(8). The second token is a number specifying the size of the created file system. Typically it will be the number of blocks on the device, perhaps diminished by space for swapping. The next token is the number of i-nodes in the i-list. The next set of tokens comprise the specification for the root file. File specifications consist of tokens giving the mode, the user-id, the group id, and the initial contents of the file. The syntax of the contents field depends on the mode.

The mode token for a file is a 6 character string. The first character specifies the type of the file. (The characters **-bcd** specify regular, block special, character special and directory files respectively.) The second character of the type is either **u** or **-** to specify set-user-id mode or not. The third is **g** or **-** for the set-group-id mode. The rest of the mode is a three digit octal number giving the owner, group, and other read, write, execute permissions, see *chmod*(1).

Two decimal number tokens come after the mode; they specify the user and group ID's of the owner of the file.

If the file is a regular file, the next token is a pathname whence the contents and size are copied.

If the file is a block or character special file, two decimal number tokens follow which give the major and minor device numbers.

If the file is a directory, *mkfs* makes the entries **.** and **..** and then reads a list of names and (recursively) file specifications for the entries in the directory. The scan is terminated with the token **$**.

If the prototype file cannot be opened and its name consists of a string of digits, *mkfs* builds a file system with a single empty directory on *it*. The size of the file system is the value of *proto* interpreted as a decimal number. The number of i-nodes is calculated as a function of the filsystem size. The boot program is left uninitialized.

A sample prototype specification follows:

```
/usr/mdec/uboot
4872 55
d--777 3 1
usr  d--777 3 1
     sh   ---755 3 1 /bin/sh
     ken  d--755 6 1
          $
     b0   b--644 3 1 0 0
     c0   c--644 3 1 0 0
     $
$
```

SEE ALSO

filsys(5), dir(5), bproc(8)

BUGS

There should be some way to specify links.

NAME

mkstr - create an error message file by massaging 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 of the specified *files*, placing a massaged version of the input file in a file whose name consists of the specified *prefix* and the original name. A typical usage of *mkstr* would be

```
mkstr pistrings xx *.c
```

This command would cause all the error messages from the C source files in the current directory to be placed in the file *pistrings* and processed copies of the source for these files to be placed in files whose names are prefixed with *xx*.

To process the error messages in the source to the message file *mkstr* keys on the string `error("' in the input stream. Each time it occurs, the C string starting at the `"' is placed in the message file followed by a null character and a new-line character; the null character terminates the message so it can be easily used when retrieved, the new-line 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, i.e.:

```
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) {
oops:
               perror(efilname);
               exit(1);
          }
```

```
                }
                if (lseek(efil, (long) a1, 0) || read(efil, buf, 256) <=
                     goto oops;
                printf(buf, a2, a3, a4);
           }
```

The optional - causes the error messages to be placed at the end of the specified message file for recompiling part of a large <u>mkstr</u>ed program.

SEE ALSO

lseek(2), xstr(UCB)

AUTHORS

Bill Joy and Charles Haley

NOTES

All the arguments except the name of the file to be processed are unnecessary.

NAME

mount, umount - mount and dismount file system

SYNOPSIS

/etc/mount [special name [-r]]

/etc/umount special

DESCRIPTION

Mount announces to the system that a removable file system is present on the device *special*. The file *name* must exist already; it must be a directory (unless the root of the mounted file system is not a directory). It becomes the name of the newly mounted root. The optional last argument indicates that the file system is to be mounted read-only.

Umount announces to the system that the removable file system previously mounted on device *special* is to be removed. First, any pending I/O for the file system is completed, and the file system is flagged clean. *Mount* will refuse to mount a file system which is not flagged clean; this can happen if a system crash prevented the use of *umount* or *haltsys*(8). In such a case, use *fsck*(1M) to clean the file system, then try *mount* again.

These commands maintain a table of mounted devices. If invoked without an argument, *mount* prints the table.

Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

FILES

/etc/mtab: mount table

SEE ALSO

mount(2), mtab(5)

DIAGNOSTICS

Exit code 0 is returned for a successful mount, 1 for a failure, 2 for attempting to mount an unclean structure.

BUGS

Mounting file systems full of garbage will crash the system. Mounting a root directory on a non-directory makes some apparently good pathnames invalid.

NAME

mv - move or rename files and directories

SYNTAX

```
mv file1 file2

mv file ... directory
```

DESCRIPTION

Mv moves (changes the name of) *file1* to *file2*.

If *file2* already exists, it is removed before *file1* is moved. If *file2* has a mode which forbids writing, *mv* prints the mode (see *chmod*(2)) and reads the standard input to obtain a line; if the line begins with y, the move takes place; if not, *mv* exits.

In the second form, one or more *files* are moved to the *directory* with their original file-names.

Mv refuses to move a file onto itself.

SEE ALSO

cp(1), chmod(2), copy(1)

NOTES

If *file1* and *file2* lie on different file systems, *mv* must copy the file and delete the original. In this case the owner name becomes that of the copying process and any linking relationship with other files is lost.

Mv should take -f flag, like *rm*, to suppress the question if the target exists and is not writable.

NAME

newgrp - log in to a new group

SYNTAX

newgrp group

DESCRIPTION

Newgrp changes the group identification of its caller, analogously to *login*(1). The same person remains logged in, and the current directory is unchanged, but calculations of access permissions to files are performed with respect to the new group ID.

A password is demanded if the group has a password and the user himself does not.

When most users log in, they are members of the group named `other.' *Newgrp* is known to the shell, which executes it directly without a fork.

FILES

/etc/group, /etc/passwd

SEE ALSO

login(1), group(5)

NAME

nice, nohup - run a command at low priority

SYNTAX

nice [-*number*] command [arguments]

nohup command [arguments]

DESCRIPTION

Nice executes *command* with low scheduling priority. If the *number* argument is present, the priority is incremented (higher numbers mean lower priorities) by that amount up to a limit of 20. The default *number* is 10.

The super-user may run commands with priority higher than normal by using a negative priority, e.g. `--10'.

Nohup executes *command* immune to hangup and terminate signals from the controlling terminal. The priority is incremented by 5. *Nohup* should be invoked from the shell with `&' in order to prevent it from responding to interrupts by or stealing the input from the next person who logs in on the same terminal.

FILES

nohup.out standard output and standard error file under *nohup*

SEE ALSO

nice(2)

DIAGNOSTICS

Nice returns the exit status of the subject command.

nm -n /xenix.vt > foo

rtopen
rtstrategy

NAME

nm - print name list

SYNTAX

nm [**-gnoprucx**] [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 (blanks if undefined) and one of the letters **U** (undefined), **A** (absolute), **T** (text segment symbol), **D** (data segment symbol), **B** (bss segment symbol), or **C** (common symbol). If the symbol is local (non-external) the type letter is in lower case. The output is sorted alphabetically.

Options are:

-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.

-p Don't sort; print in symbol-table order.

-r Sort in reverse order.

-u Print only undefined symbols.

-c Print only C program symbols (symbols which begin with `_') as they appeared in the C program.

-x Symbol values are printed in hexadecimal rather than octal.

FILES

a.out Default input file.

SEE ALSO

ar(1), ar(5), a.out(5)

cp

NAME

nroff, troff - text formatting and typesetting

SYNTAX

nroff [option] ... [file] ...

troff [option] ... [file] ...

DESCRIPTION

Troff formats text in the named *files* for printing on a Graphic Systems C/A/T phototypesetter; *nroff* for typewriter-like devices. Their capabilities are described in the *Nroff/Troff user's manual*.

If no *file* argument is present, the standard input is read. An argument consisting of a single minus (-) is taken to be a file name corresponding to the standard input. The options, which may appear in any order so long as they appear before the files, are:

-o*list*	Print only pages whose page numbers appear in the comma-separated *list* of numbers and ranges. A range *N*-*M* means pages *N* through *M*; an initial -*N* means from the beginning to page *N*; and a final *N*- means from *N* to the end.
-n*N*	Number first generated page *N*.
-s*N*	Stop every *N* pages. *Nroff* will halt prior to every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a newline. *Troff* will stop the phototypesetter every *N* pages, produce a trailer to allow changing cassettes, and resume when the typesetter's start button is pressed.
-m*name*	Prepend the macro file **/usr/lib/tmac/tmac.***name* to the input *files*.
-r*aN*	Set register *a* (one-character) to *N*.
-i	Read standard input after the input files are exhausted.
-q	Invoke the simultaneous input-output mode of the **rd** request.

Nroff only

-T*name*	Prepare output for specified terminal. Known *names* are **37** for the (default) Teletype Corporation Model 37 terminal, **tn300** for the GE TermiNet 300 (or any terminal without half-line capability), **300S** for the

DASI-300S, 300 for the DASI-300, and 450 for the DASI-450 (Diablo Hyterm).

-e Produce equally-spaced words in adjusted lines, using full terminal resolution.

-h Use output tabs during horizontal spacing to speed output and reduce output character count. Tab settings are assumed to be every 8 nominal character widths.

Troff only

-t Direct output to the standard output instead of the phototypesetter.

-f Refrain from feeding out paper and stopping phototypesetter at the end of the run.

-w Wait until phototypesetter is available, if currently busy.

-b Report whether the phototypesetter is busy or available. No text processing is done.

-a Send a printable ASCII approximation of the results to the standard output.

-pN Print all characters in point size _N_ while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.

-g Prepare output for a GCOS phototypesetter and direct it to the standard output (see _gcat_(1)).

If the file _/usr/adm/tracct_ is writable, _troff_ keeps phototypesetter accounting records there. The integrity of that file may be secured by making _troff_ a `set user-id' program.

FILES

/usr/lib/suftab	suffix hyphenation tables
/tmp/ta*	temporary file
/usr/lib/tmac/tmac.*	standard macro files
/usr/lib/term/*	terminal driving tables for _nroff_
/usr/lib/font/*	font width tables for _troff_
/dev/cat	phototypesetter
/usr/adm/tracct	accounting statistics for /dev/cat

SEE ALSO

J. F. Ossanna, _Nroff/Troff user's manual_
B. W. Kernighan, _A TROFF Tutorial_
eqn(1), tbl(1)

col(1), tk(1) (*nroff* only)
tc(1), gcat(1) (*troff* only)

NAME

od - octal dump

SYNTAX

od [-bcdox] [file] [[+]offset[.][b]]

DESCRIPTION

_Od_ dumps _file_ in one or more formats as selected by the first argument. If the first argument is missing, -o is default. The meanings of the format argument characters are:

b Interpret bytes in octal.

c Interpret bytes in ASCII. Certain non-graphic characters appear as C escapes: null=\0, backspace=\b, formfeed=\f, newline=\n, return=\r, tab=\t; others appear as 3-digit octal numbers.

d Interpret words in decimal.

o Interpret words in octal.

x Interpret words in hex.

The _file_ argument specifies which file is to be dumped. If no file argument is specified, the standard input is used.

The offset argument specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If `.' is appended, the offset is interpreted in decimal. If `b' is appended, the offset is interpreted in blocks of 512 bytes. If the file argument is omitted, the offset argument must be preceded `+'.

Dumping continues until end-of-file.

SEE ALSO

adb(1)

NAME

pr - print file

SYNTAX

pr [option] ... [file] ...

DESCRIPTION

_Pr_ produces a printed listing of one or more _files_. The output is separated into pages headed by a date, the name of the file or a specified header, and the page number. If there are no file arguments, _pr_ prints its standard input.

Options apply to all following files but may be reset between files:

-_n_	Produce _n_-column output.
+_n_	Begin printing with page _n_.
-h	Take the next argument as a page header.
-w_n_	For purposes of multi-column output, take the width of the page to be _n_ characters instead of the default 72.
-l_n_	Take the length of the page to be _n_ lines instead of the default 66.
-t	Do not print the 5-line header or the 5-line trailer normally supplied for each page.
-s_c_	Separate columns by the single character _c_ instead of by the appropriate amount of white space. A missing _c_ is taken to be a tab.
-m	Print all _files_ simultaneously, each in one column,
-b	A form feed character is used to separate pages, normally pages are separated by a number of newline characters.

Inter-terminal messages via _write_(1) are forbidden during a _pr_.

FILES

/dev/tty* to suspend messages.

SEE ALSO

cat(1)

DIAGNOSTICS

There are no diagnostics when _pr_ is printing on a terminal.

NAME

prep - prepare text for statistical processing

SYNTAX

prep [-dio] file ...

DESCRIPTION

Prep reads each *file* in sequence and writes it on the standard output, one `word' to a line. A word is a string of alphabetic characters and imbedded apostrophes, delimited by space or punctuation. Hyphented words are broken apart; hyphens at the end of lines are removed and the hyphenated parts are joined. Strings of digits are discarded.

The following option letters may appear in any order:

-d Print the word number (in the input stream) with each word.

-i Take the next *file* as an `ignore' file. These words will not appear in the output. (They will be counted, for purposes of the -d count.)

-o Take the next *file* as an `only' file. Only these words will appear in the output. (All other words will also be counted for the -d count.)

-p Include punctuation marks (single nonalphanumeric characters) as separate output lines. The punctuation marks are not counted for the -d count.

Ignore and only files contain words, one per line.

SEE ALSO

deroff(1)

NAME

print - pr to the line printer

SYNTAX

print file ...

DESCRIPTION

Print *pr*'*s* a copy of each named file on the line printer. *It is a* one line shell script:

```
pr $* | lpr
```

SEE ALSO

lpr(UCB), pr(1)

NOTES

NAME

printenv - print out the environment

SYNTAX

printenv [name]

DESCRIPTION

Printenv prints out the values of the variables in the environment. If a *name* is specified, only its value is printed.

If a *name* is specified and it is not defined in the environment, *printenv* returns exit status 1, else it returns status 0.

SEE ALSO

sh(1), environ(5), csh(UCB)

NOTES

NAME

prof - display profile data

SYNTAX

prof [-v] [-a] [-l] [-<u>low</u> [-<u>high</u>]] [file]

DESCRIPTION

<u>Prof</u> interprets the file <u>mon.out</u> produced by the <u>monitor</u> subroutine. Under default modes, the symbol table in the named object file (<u>a.out</u> default) is read and correlated with the <u>mon.out</u> 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.

If the -v option is used, all printing is suppressed and a graphic version of the profile is produced on the standard output for display by the <u>plot</u>(1) filters. The numbers <u>low</u> and <u>high</u>, by default 0 and 100, cause a selected percentage of the profile to be plotted with accordingly higher resolution.

In order for the number of calls to a routine to be tallied, the -p option of <u>cc</u> must have been given when the file containing the routine was compiled. This option also arranges for the <u>mon.out</u> file to be produced automatically.

FILES

mon.out	for profile
a.out	for namelist

SEE ALSO

monitor(3), profil(2), cc(1), plot(1)

NOTES

Beware of quantization errors.

If you use an explicit call to <u>monitor</u>(3) you will need to make sure the buffer size is equal to or smaller than the program size.

NAME

ps - process status

SYNTAX

ps [**aklx**] [namelist]

DESCRIPTION

Ps prints certain indicia about active processes. The a option asks for information about all processes with terminals (ordinarily only one's own processes are displayed); x asks even about processes with no terminal; l asks for a long listing. The short listing contains the process ID, tty letter, the cumulative execution time of the process and an approximation to the command line.

The long listing is columnar and contains

F	Flags associated with the process. 01: in core; 02: system process; 04: locked in core (e.g. for physical I/O); 10: being swapped; 20: being traced by another process.
S	The state of the process. 0: nonexistent; S: sleeping; W: waiting; R: running; I: intermediate; Z: terminated; T: stopped.
UID	The user ID of the process owner.
PID	The process ID of the process; as in certain cults it is possible to kill a process if you know its true name.
PPID	The process ID of the parent process.
CPU	Processor utilization for scheduling.
PRI	The priority of the process; high numbers mean low priority.
NICE	Used in priority computation.
ADDR	The core address of the process if resident, otherwise the disk address.
SZ	The size in blocks of the core image of the process.
WCHAN	The event for which the process is waiting or sleeping; if blank, the process is running.
TTY	The controlling tty for the process.

TIME The cumulative execution time for the process.

The command and its arguments.

A process that has exited and has a parent, but has not yet been waited for by the parent is marked <defunct>. *Ps* makes an educated guess as to the file name and arguments given when the process was created by examining core memory or the swap area. The method is inherently somewhat unreliable and in any event a process is entitled to destroy this information, so the names cannot be counted on too much.

If the **k** option is specified, the file /*usr*/*sys*/*core* is used in place of /*dev*/*mem*. This is used for postmortem system debugging. If a second argument is given, it is taken to be the file containing the system's namelist.

FILES

/xenix	system namelist
/dev/mem	core memory
/usr/sys/core	alternate core file
/dev	searched to find swap device and tty names

SEE ALSO

kill(1)

NOTES

Things can change while *ps* is running; the picture it gives is only a close approximation to reality.
Some data printed for defunct processes is irrelevant

NAME

ptx - permuted index

SYNTAX

ptx [option] ... [input [output]]

DESCRIPTION

Ptx generates a permuted index to file *input* on file *output* (standard input and output default). It has three phases: the first does the permutation, generating one line for each keyword in an input line. The keyword is rotated to the front. The permuted file is then sorted. Finally, the sorted lines are rotated so the keyword comes at the middle of the page. *Ptx* produces output in the form:

```
.xx "tail" "before keyword" "keyword and after" "head"
```

where .xx may be an *nroff* or *troff*(1) macro for user-defined formatting. The *before keyword* and *keyword and after* fields incorporate as much of the line as will fit around the keyword when it is printed at the middle of the page. *Tail* and *head*, at least one of which is an empty string "", are wrapped-around pieces small enough to fit in the unused space at the opposite end of the line. When original text must be discarded, `/' marks the spot.

The following options can be applied:

-f Fold upper and lower case letters for sorting.

-t Prepare the output for the phototypesetter; the default line length is 100 characters.

-w *n* Use the next argument, *n*, as the width of the output line. The default line length is 72 characters.

-g *n* Use the next argument, *n*, as the number of characters to allow for each gap among the four parts of the line as finally printed. The default gap is 3 characters.

-o only
Use as keywords only the words given in the *only* file.

-i ignore
Do not use as keywords any words given in the *ignore* file. If the -*i* and -*o* options are missing, use /usr/lib/eign as the *ignore* file.

-b break
Use the characters in the *break* file to separate words. In any case, tab, newline, and space characters are always used as break characters.

-r Take any leading nonblank characters of each input line to be a reference identifier (as to a page or chapter) separate from the text of the line. Attach that identifier as a 5th field on each output line.

The index for this manual was generated using *ptx*.

FILES

/bin/sort
/usr/lib/eign

NOTES

Line length counts do not account for overstriking or proportional spacing.

NAME
pwd - working directory name

SYNTAX
pwd

DESCRIPTION
Pwd prints the pathname of the working (current) directory.

SEE ALSO
cd(1)

NAME

quot - summarize file system ownership

SYNOPSIS

quot [option] ... [filesystem]

DESCRIPTION

Quot prints the number of blocks in the named *filesystem* currently owned by each user. If no *filesystem* is named, a default name is assumed. The following options are available:

-n Cause the pipeline **ncheck filesystem | sort +0n | quot -n filesystem** to produce a list of all files and their owners.

-c Print three columns giving file size in blocks, number of files of that size, and cumulative total of blocks in that size or smaller file.

-f Print count of number of files as well as space owned by each user.

FILES

Default file system varies with system.
/etc/passwd to get user names

SEE ALSO

ls(1), du(1)

BUGS

Holes in files are counted as if they actually occupied space.

NAME

ranlib - convert 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*(1) to reconstruct the archive, so that sufficient temporary file space must be available in the file system containing the current directory.

SEE ALSO

ld(1), ar(1), copy(1), settime(1)

NOTES

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

restor - incremental file system restore

SYNOPSIS

restor key [argument ...]

DESCRIPTION

Restor is used to read magtapes dumped with the *dump* command. The *key* specifies what is to be done. *Key* is one of the characters **rRxt** optionally combined with **f.**

f Use the first *argument* as the name of the tape instead of the default.

r or R
The tape is read and loaded into the file system specified in *argument*. This should not be done lightly (see below). If the key is R *restor* asks which tape of a multi volume set to start on. This allows restor to be interrupted and then restarted (an *icheck* -*s* must be done before restart).

x Each file on the tape named by an *argument* is extracted. The file name has all `mount' prefixes removed; for example, /usr/bin/lpr is named /bin/lpr on the tape. The file extracted is placed in a file with a numeric name supplied by *restor* (actually the inode number). In order to keep the amount of tape read to a minimum, the following procedure is recommended:

Mount volume 1 of the set of dump tapes.

Type the *restor* command.

Restor will announce whether or not it found the files, give the number it will name the file, and rewind the tape.

It then asks you to `mount the desired tape volume'. Type the number of the volume you choose. On a multi volume dump the recommended procedure is to mount the last through the first volume in that order. *Restor* checks to see if any of the files requested are on the mounted tape (or a later tape, thus the reverse order) and doesn't read through the tape if no files are. If you are working with a single volume dump or the number of files being restored is large, respond to the query with `1' and *restor* will read the tapes in sequential order.

If you have a hierarchy to restore you can use dumpdir(1) to produce the list of names and a shell script to move the resulting files to their homes.

t Print the date the tape was written and the date the filesystem was dumped from.

The **r** option should only be used to restore a complete dump tape onto a clear file system or to restore an incremental dump tape onto this. Thus

```
/etc/mkfs /dev/rp0 40600
restor r /dev/rp0
```

is a typical sequence to restore a complete dump. Another *restor* can be done to get an incremental dump in on top of this.

A *dump* followed by a *mkfs* and a *restor* is used to change the size of a file system.

FILES

default tape unit varies with installation
rst*

SEE ALSO

dump(1), mkfs(1), dumpdir(1)

DIAGNOSTICS

There are various diagnostics involved with reading the tape and writing the disk. There are also diagnostics if the i-list or the free list of the file system is not large enough to hold the dump.

If the dump extends over more than one tape, it may ask you to change tapes. Reply with a new-line when the next tape has been mounted.

BUGS

There is redundant information on the tape that could be used in case of tape reading problems. Unfortunately, *restor* doesn't use it.

NAME

rev - reverse lines of a file

SYNTAX

```
rev [ file ] ...
```

DESCRIPTION

Rev copies the named files to the standard output, reversing the order of characters in every line. If no file is specified, the standard input is copied.

NAME

rm, rmdir - remove (unlink) files

SYNTAX

rm [**-fri**] file ...

rmdir dir ...

DESCRIPTION

Rm removes the entries for one or more files from a directory. If an entry was the last link to the file, the file is destroyed. Removal of a file requires write permission in its directory, but neither read nor write permission on the file itself.

If a file has no write permission and the standard input is a terminal, its permissions are printed and a line is read from the standard input. If that line begins with \`y' the file is deleted, otherwise the file remains. No questions are asked when the **-f** (force) option is given.

If a designated file is a directory, an error comment is printed unless the optional argument **-r** has been used. In that case, *rm* recursively deletes the entire contents of the specified directory, and the directory itself.

If the **-i** (interactive) option is in effect, *rm* asks whether to delete each file, and, under **-r**, whether to examine each directory.

Rmdir removes entries for the named directories, which must be empty.

SEE ALSO

unlink(2)

DIAGNOSTICS

Generally self-explanatory. It is forbidden to remove the file \`..' merely to avoid the antisocial consequences of inadvertently doing something like \`rm -r .*'.

NAME

sa, accton - system accounting

SYNOPSIS

sa [-abcijlnrstuv] [file]

/etc/accton [file]

DESCRIPTION

With an argument naming an existing _file_, _accton_ causes system accounting information for every process executed to be placed at the end of the file. If no argument is given, accounting is turned off.

_Sa_ reports on, cleans up, and generally maintains accounting files.

_Sa_ is able to condense the information in /_usr_/_adm_/_acct_ into a summary file /_usr_/_adm_/_savacct_ which contains a count of the number of times each command was called and the time resources consumed. This condensation is desirable because on a large system _acct_ can grow by 100 blocks per day. The summary file is read before the accounting file, so the reports include all available information.

If a file name is given as the last argument, that file will be treated as the accounting file; /_usr_/_adm_/_acct_ is the default. There are zillions of options:

a	Place all command names containing unprintable characters and those used only once under the name \`***other.'
b	Sort output by sum of user and system time divided by number of calls. Default sort is by sum of user and system times.
c	Besides total user, system, and real time for each command print percentage of total time over all commands.
i	Ignore the summary files /_usr_/_adm_/_savacct_ and /_usr_/_adm_/_usracct_; do not include their contents in this report.
j	Instead of total minutes time for each category, give seconds per call.
l	Separate system and user time; normally they are combined.
m	Print number of processes and number of CPU minutes for each user.

n	Sort by number of calls.
r	Reverse order of sort.
s	Merge accounting file into summary file /*usr*/*adm*/*savacct* when done.
t	For each command report ratio of real time to the sum of user and system times.
u	Superseding all other flags, print for each command in the accounting file the user ID and command name.
v	If the next character is a digit *n*, then type the name of each command used *n* times or fewer. Await a reply from the typewriter; if it begins with `y', add the command to the category `**junk**.' This is used to strip out garbage.
(default)	A table of 4 columns is printed: the number of calls, the total real time, the total combined system and user time, and the name of the command. The first line in the table contains the sum of each column.

FILES

/usr/adm/acct	raw accounting
/usr/adm/savacct	summary
/usr/adm/usracct	per-user summary

SEE ALSO

ac(1), acct(2)

NAME

sddate - print and set dump dates

SYNOPSIS

sddate [name lev date]

DESCRIPTION

If no argument is given, the contents of the dump date file `/etc/ddate' are printed. The dump date file is maintained by dump(1M) and contains the date of the most recent dump for each dump level for each filesystem.

If arguments are given, an entry is replaced or made in `/etc/ddate'. *name* is the last component of the device pathname. *lev* is the dump level number (from 0 to 9), and *date* is a time in the form taken by *date*(1).

Some sites may wish to backup filesystems by coping them verbatim to dismountable packs. *Sddate* could be used to make a `level 0' entry in `/etc/ddate', which would then allow incremental mag tape dumps.

For example:

```
sddate rrp3 5 10081520
```

makes an `/etc/ddate' entry showing a level 5 dump of `/dev/rrp3' on October 8, at 3:20 PM.

FILES

/etc/ddate

SEE ALSO

dump(1M), date(1)

DIAGNOSTICS

`bad conversion' if the date set is syntactically incorrect.

NAME

sed - stream editor

SYNTAX

```
sed [ -n ] [ -e script ] [ -f sfile ] [ file ] ...
```

DESCRIPTION

Sed copies the named _files_ (standard input default) to the standard output, edited according to a script of commands. The -f option causes the script to be taken from file _sfile_; these options accumulate. If there is just one -e option and no -f's, the flag -e may be omitted. The -n option suppresses the default output.

A script consists of editing commands, one per line, of the following form:

```
[address [, address] ] function [arguments]
```

In normal operation _sed_ cyclically copies a line of input into a _pattern space_ (unless there is something left after a `D' command), applies in sequence all commands whose _addresses_ select that pattern space, and at the end of the script copies the pattern space to the standard output (except under -n) and deletes the pattern space.

An _address_ is either a decimal number that counts input lines cumulatively across files, a `$' that addresses the last line of input, or a context address, `/regular expression/', in the style of _ed_(1) modified thus:

> The escape sequence `\n' matches a newline embedded in the pattern space.

A command line with no addresses selects every pattern space.

A command line with one address selects each pattern space that matches the address.

A command line with two addresses selects the inclusive range from the first pattern space that matches the first address through the next pattern space that matches the second. (If the second address is a number less than or equal to the line number first selected, only one line is selected.) Thereafter the process is repeated, looking again for the first address.

Editing commands can be applied only to non-selected pattern spaces by use of the negation function `!' (below).

In the following list of functions the maximum number of permissible addresses for each function is indicated in parentheses.

An argument denoted _text_ consists of one or more lines, all but the last of which end with `\' to hide the newline. Backslashes in text are treated like backslashes in the replacement string of an `s' command, and may be used to protect initial blanks and tabs against the stripping that is done on every script line.

An argument denoted _rfile_ or _wfile_ must terminate the command line and must be preceded by exactly one blank. Each _wfile_ is created before processing begins. There can be at most 10 distinct _wfile_ arguments.

(1)a\
text
    Append. Place _text_ on the output before reading the next input line.

(2)b _label_
    Branch to the `:' command bearing the _label_. If _label_ is empty, branch to the end of the script.

(2)c\
text
    Change. Delete the pattern space. With 0 or 1 address or at the end of a 2-address range, place _text_ on the output. Start the next cycle.

(2)d Delete the pattern space. Start the next cycle.

(2)D Delete the initial segment of the pattern space through the first newline. Start the next cycle.

(2)g Replace the contents of the pattern space by the contents of the hold space.

(2)G Append the contents of the hold space to the pattern space.

(2)h Replace the contents of the hold space by the contents of the pattern space.

(2)H Append the contents of the pattern space to the hold space.

(1)i\
text Insert. Place _text_ on the standard output.

(2)l List the pattern space on the standard output in an unambiguous form. Non-printing characters are spelled in two digit ascii, and long lines are folded.

(2)n Copy the pattern space to the standard output. Replace the pattern space with the next line of input.

(2)N Append the next line of input to the pattern space with an embedded newline. (The current line number changes.)

(2)p Print. Copy the pattern space to the standard output.

(2)P Copy the initial segment of the pattern space through the first newline to the standard output.

(1)q Quit. Branch to the end of the script. Do not start a new cycle.

(2)r _rfile_
Read the contents of _rfile_. Place them on the output before reading the next input line.

(2)s/_regular expression_/_replacement_/_flags_
Substitute the _replacement_ string for instances of the _regular expression_ in the pattern space. Any character may be used instead of \`/'. For a fuller description see _ed_(1). _Flags_ is zero or more of

- g Global. Substitute for all nonoverlapping instances of the _regular expression_ rather than just the first one.
- p Print the pattern space if a replacement was made.
- w _wfile_
  Write. Append the pattern space to _wfile_ if a replacement was made.

(2)t _label_
Test. Branch to the \`:' command bearing the _label_ if any substitutions have been made since the most recent reading of an input line or execution of a \`t'. If _label_ is empty, branch to the end of the script.

(2)w _wfile_
Write. Append the pattern space to _wfile_.

(2)x Exchange the contents of the pattern and hold spaces.

(2)y/_string1_/_string2_/
Transform. Replace all occurrences of characters in _string1_ with the corresponding character in _string2_.

The lengths of _string1_ and _string2_ must be equal.

(2)! _function_
Don't. Apply the _function_ (or group, if _function_ is `{') only to lines _not_ selected by the address(es).

(0): _label_
This command does nothing; it bears a _label_ for `b' and `t' commands to branch to.

(1)= Place the current line number on the standard output as a line.

(2){ Execute the following commands through a matching `}' only when the pattern space is selected.

(0) An empty command is ignored.

SEE ALSO
ed(1), grep(1), awk(1)

NAME

settime - change the access and modification dates of files

SYNTAX

settime [yymmddhhmm [.ss]] [-f fname] name ...

DESCRIPTION

Set the access and modification dates for one or more files. The dates are set to the specified date, or to the access and modification dates of the file specified via **-f**. Exactly one of these methods must be used to specify the new date(s). _yy_ is the last two digits of the year; the first _mm_ is the month number; _dd_ is the day number in the month; _hh_ is the hour number (24 hour system); the second _mm_ is the minute number; ._ss_ is optional and is the seconds. For example:

```
settime 10080045 ralph pete
```

sets the access and modification dates of files _ralph_ and _pete_ to Oct 8, 12:45 AM. The year, month and day may be omitted, the current values being the current date.

```
settime -f ralph john
```

sets the access and modification dates of the file _john_ to those of the file _ralph_.

NAME

sh, for, case, if, while, :, ., break, continue, cd, eval, exec, exit, export, login, newgrp, read, readonly, set, shift, times, trap, umask, wait - command language

SYNTAX

sh [-ceiknrstuvx] [arg] ...

DESCRIPTION

_Sh_ is a command programming language that executes commands read from a terminal or a file. See **invocation** for the meaning of arguments to the shell.

Commands.
A _simple-command_ is a sequence of non blank _words_ separated by blanks (a blank is a **tab** or a **space**). The first word specifies the name of the command to be executed. Except as specified below the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 (see _exec_(2)). The _value_ of a simple-command is its exit status if it terminates normally or 200+_status_ if it terminates abnormally (see _signal_(2) for a list of status values).

A _pipeline_ is a sequence of one or more _commands_ separated by **|**. The standard output of each command but the last is connected by a _pipe_(2) to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate.

A _list_ is a sequence of one or more _pipelines_ separated by **;**, **&**, **&&** or **||** and optionally terminated by **;** or **&**. **;** and **&** have equal precedence which is lower than that of **&&** and **||**, **&&** and **||** also have equal precedence. A semicolon causes sequential execution; an ampersand causes the preceding _pipeline_ to be executed without waiting for it to finish. The symbol **&&** (**||**) causes the _list_ following to be executed only if the preceding _pipeline_ returns a zero (non zero) value. Newlines may appear in a _list_, instead of semicolons, to delimit commands.

A _command_ is either a simple-command or one of the following. The value returned by a command is that of the last simple-command executed in the command.

for _name_ [in _word_ ...] do _list_ done
Each time a **for** command is executed _name_ is set to the next word in the **for** word list If **in** _word_ ... is omitted then **in "$@"** is assumed. Execution ends when there are no more words in the list.

case _word_ in [_pattern_ [| _pattern_] ...) _list_ ;;] ... **esac**

A **case** command executes the list associated with the first pattern that matches word. The form of the patterns is the same as that used for file name generation.

if list **then** list [**elif** list **then** list] ... [**else** list] **fi**
The list following **if** is executed and if it returns zero the list following **then** is executed. Otherwise, the list following **elif** is executed and if its value is zero the list following **then** is executed. Failing that the **else** list is executed.

while list [**do** list] **done**
A **while** command repeatedly executes the **while** list and if its value is zero executes the **do** list; otherwise the loop terminates. The value returned by a **while** command is that of the last executed command in the **do** list. **until** may be used in place of **while** to negate the loop termination test.

(list)
Execute list in a subshell.

{ list }
list is simply executed.

The following words are only recognized as the first word of a command and when not quoted.

if then else elif fi case in esac for while until do done { }

Command substitution.
The standard output from a command enclosed in a pair of grave accents (\`\`) may be used as part or all of a word; trailing newlines are removed.

Parameter substitution.
The character $ is used to introduce substitutable parameters. Positional parameters may be assigned values by **set**. Variables may be set by writing

name=value [name=value] ...

${parameter}
A parameter is a sequence of letters, digits or underscores (a name), a digit, or any of the characters *** @ # ? - $!**. The value, if any, of the parameter is substituted. The braces are required only when parameter is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If parameter is a digit then it is a positional parameter. If

parameter is * or @ then all the positional parameters, starting with **$1**, are substituted separated by spaces. **$0** is set from argument zero when the shell is invoked.

${_parameter_-_word_}
: If _parameter_ is set then substitute its value; otherwise substitute _word_.

${_parameter_=_word_}
: If _parameter_ is not set then set it to _word_; the value of the parameter is then substituted. Positional parameters may not be assigned to in this way.

${_parameter_?_word_}
: If _parameter_ is set then substitute its value; otherwise, print _word_ and exit from the shell. If _word_ is omitted then a standard message is printed.

${_parameter_+_word_}
: If _parameter_ is set then substitute _word_; otherwise substitute nothing.

In the above _word_ is not evaluated unless it is to be used as the substituted string. (So that, for example, echo ${d-`pwd`} will only execute _pwd_ if _d_ is unset.)

The following _parameters_ are automatically set by the shell.

#	The number of positional parameters in decimal.
-	Options supplied to the shell on invocation or by set.
?	The value returned by the last executed command in decimal.
$	The process number of this shell.
!	The process number of the last background command invoked.

The following _parameters_ are used but not set by the shell.

HOME	The default argument (home directory) for the **cd** command.
PATH	The search path for commands (see **execution**).
MAIL	If this variable is set to the name of a mail file then the shell informs the user of the arrival of mail in the specified file.
PS1	Primary prompt string, by default `$ '.
PS2	Secondary prompt string, by default `> '.
IFS	Internal field separators, normally **space**, **tab**, and **newline**.

Blank interpretation.
After parameter and command substitution, any results of

substitution are scanned for internal field separator characters (those found in **$IFS**) and split into distinct arguments where such characters are found. Explicit null arguments ("" or '') are retained. Implicit null arguments (those resulting from _parameters_ that have no values) are removed.

File name generation.
Following substitution, each command word is scanned for the characters *****, **?** and **[**. If one of these characters appears then the word is regarded as a pattern. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern then the word is left unchanged. The character **.** at the start of a file name or immediately following a /, and the character /, must be matched explicitly.

***** Matches any string, including the null string.

? Matches any single character.

[...]
Matches any one of the characters enclosed. A pair of characters separated by **-** matches any character lexically between the pair.

Quoting.
The following characters have a special meaning to the shell and cause termination of a word unless quoted.

; & ( ) | < > newline space tab

A character may be _quoted_ by preceding it with a \\. **\\new-line** is ignored. All characters enclosed between a pair of quote marks (''), except a single quote, are quoted. Inside double quotes ("") parameter and command substitution occurs and \ quotes the characters **\\ ` "** and **$**.

"$*" is equivalent to "$1 $2 ..." whereas
"$@" is equivalent to "$1" "$2" ... .

Prompting.
When used interactively, the shell prompts with the value of PS1 before reading a command. If at any time a newline is typed and further input is needed to complete a command then the secondary prompt (**$PS2**) is issued.

Input output.
Before a command is executed its input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a simple-command or may precede or follow a _command_ and are not passed on to the invoked command. _Substitution_ occurs before _word_ or _digit_ is used.

<*word*
: Use file *word* as standard input (file descriptor 0).

>*word*
: Use file *word* as standard output (file descriptor 1). If the *file* does not exist then it is created; otherwise it is truncated to zero length.

>>*word*
: Use file *word* as standard output. If the file exists then *output is* appended (by seeking to the end); otherwise the file is created.

<<*word*
: The shell input is read up to a line the same as *word*, or end of file. The resulting document becomes the standard input. If any character of *word* is quoted then no interpretation is placed upon the characters of the document; otherwise, parameter and command substitution occurs, **\newline** is ignored, and \ is used to quote the characters \ **$** ` and the first character of *word*.

<&*digit*
: The standard input is duplicated from file descriptor *digit*; see *dup*(2). Similarly for the standard output using >.

<&- The standard input is closed. Similarly for the standard output using >.

If one of the above is preceded by a digit then the file descriptor created is that specified by the digit (instead of the default 0 or 1). For example,

```
... 2>&1
```

creates file descriptor 2 to be a duplicate of file descriptor 1.

If a command is followed by & then the default standard input for the command is the empty file (/dev/null). Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input output specifications.

Environment.
The environment is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list; see *exec*(2) and *environ*(5). The shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a *parameter* for each name

found, giving it the corresponding value. Executed commands inherit the same environment. If the user modifies the values of these parameters or creates new ones, none of these affects the environment unless the **export** command is used to bind the shell's parameter to the environment. The environment seen by any executed command is thus composed of any unmodified name-value pairs originally inherited by the shell, plus any modifications or additions, all of which must be noted in **export** commands.

The environment for any simple-command may be augmented by prefixing it with one or more assignments to parameters. Thus these two lines are equivalent

```
TERM=450 cmd args
(export TERM; TERM=450; cmd args)
```

If the **-k** flag is set, all keyword arguments are placed in the environment, even if the occur after the command name. The following prints `a=b c' and `c':

```
echo a=b c
set -k
echo a=b c
```

Signals.
The INTERRUPT and QUIT signals for an invoked command are ignored if the command is followed by **&**; otherwise signals have the values inherited by the shell from its parent. (But see also **trap.**)

Execution.
Each time a command is executed the above substitutions are carried out. Except for the `special commands' listed below a new process is created and an attempt is made to execute the command via an exec(2).

The shell parameter **$PATH** defines the search path for the directory containing the command. Each alternative directory name is separated by a colon (**:**). The default path is **:/bin:/usr/bin.** If the command name contains a / then the search path is not used. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an a.out file, it is assumed to be a file containing shell commands. A subshell (i.e., a separate process) is spawned to read it. A parenthesized command is also executed in a subshell.

Special commands.
The following commands are executed in the shell process and except where specified no input output redirection is permitted for such commands.

: No effect; the command does nothing.

. _file_
Read and execute commands from _file_ and return. The search path **$PATH** is used to find the directory containing _file_.

break [_n_]
Exit from the enclosing **for** or **while** loop, if any. If _n_ is specified then break _n_ levels.

continue [_n_]
Resume the next iteration of the enclosing **for** or **while** loop. If _n_ is specified then resume at the _n_-th enclosing loop.

cd [_arg_]
Change the current directory to _arg_. The shell parameter $HOME is the default _arg_.

eval [_arg_ ...]
The arguments are read as input to the shell and the resulting command(s) executed.

exec [_arg_ ...]
The command specified by the arguments is executed in place of this shell without creating a new process. Input output arguments may appear and if no other arguments are given cause the shell input output to be modified.

exit [_n_]
Causes a non interactive shell to exit with the exit status specified by _n_. If _n_ is omitted then the exit status is that of the last command executed. (An end of file will also exit from the shell.)

export [_name_ ...]
The given names are marked for automatic export to the _environment_ of subsequently-executed commands. If no arguments are given then a list of exportable names is printed.

login [_arg_ ...]
Equivalent to `exec login arg ...'.

newgrp [_arg_ ...]
Equivalent to `exec newgrp arg ...'.

read _name_ ...
One line is read from the standard input; successive words of the input are assigned to the variables _name_ in order, with leftover words to the last variable. The return code is 0 unless the end-of-file is encountered.

readonly [_name_ ...]
The given names are marked readonly and the values of the these names may not be changed by subsequent assignment. If no arguments are given then a list of all readonly names is printed.

set [**-eknptuvx** [_arg_ ...]]
-e If non interactive then exit immediately if a command fails.

- **-k** All keyword arguments are placed in the environment for a command, not just those that precede the command name.
- **-n** Read commands but do not execute them.
- **-t** Exit after reading and executing one command.
- **-u** Treat unset variables as an error when substituting.
- **-v** Print shell input lines as they are read.
- **-x** Print commands and their arguments as they are executed.
- **-** Turn off the **-x** and **-v** options.

These flags can also be used upon invocation of the shell. The current set of flags may be found in **$-**.

Remaining arguments are positional parameters and are assigned, in order, to **$1**, **$2**, etc. If no arguments are given then the values of all names are printed.

shift
The positional parameters from **$2...** are renamed **$1...**

times
Print the accumulated user and system times for processes run from the shell.

trap [_arg_] [_n_] ...
Arg is a command to be read and executed when the shell receives signal(s) _n_. (Note that _arg_ is scanned once when the trap is set and once when the trap is taken.) Trap commands are executed in order of signal number. If _arg_ is absent then all trap(s) _n_ are reset to their original values. If _arg_ is the null string then this signal is ignored by the shell and by invoked commands. If _n_ is 0 then the command _arg_ is executed on exit from the shell, otherwise upon receipt of signal _n_ as numbered in _signal_(2). _Trap_ with no arguments prints a list of commands associated with each signal number.

umask [_nnn_]
The user file creation mask is set to the octal value _nnn_ (see _umask_(2)). If _nnn_ is omitted, the current value of the mask is printed.

wait [_n_]
Wait for the specified process and report its termination status. If _n_ is not given then all currently active child processes are waited for. The return code from this command is that of the process waited for.

Invocation.
If the first character of argument zero is **-**, commands are read from **$HOME/.profile**, if such a file exists. Commands

are then read as described below. The following flags are interpreted by the shell when it is invoked.

-c *string*	If the **-c** flag is present then commands are read from *string*.
-s	If the **-s** flag is present or if no arguments remain then commands are read from the standard input. Shell output is written to file descriptor 2.
-i	If the **-i** flag is present or if the shell input and output are attached to a terminal (as told by *gtty*) then this shell is *interactive*. In this case the terminate signal SIGTERM (see *signal*(2)) is ignored (so that `kill 0' does not kill an interactive shell) and the interrupt signal SIGINT is caught and ignored (so that **wait** is interruptable). In all cases SIGQUIT is ignored by the shell.

The remaining flags and arguments are described under the **set** command.

FILES

$HOME/.profile
/tmp/sh*
/dev/null

SEE ALSO

test(1), exec(2),

DIAGNOSTICS

Errors detected by the shell, such as syntax errors cause the shell to return a non zero exit status. If the shell is being used non interactively then execution of the shell file is abandoned. Otherwise, the shell returns the exit status of the last command executed (see also **exit**).

NOTES

If << is used to provide standard input to an asynchronous process invoked by &, the shell gets mixed up about naming the input document. A garbage file /tmp/sh* is created, and the shell complains about not being able to find the file by another name.

NAME

size - size of an object file

SYNTAX

size [object ...]

DESCRIPTION

Size prints the (decimal) number of bytes required by the text, data, and bss portions, and their sum in octal and decimal, of each object-file argument. If no file is specified, **a.out** is used.

SEE ALSO

a.out(5)

NAME

sleep - suspend execution for an interval

SYNTAX

sleep time

DESCRIPTION

Sleep suspends execution for *time* seconds. It is used to execute a command after a certain amount of time as in:

```
(sleep 105; command)&
```

or to execute a command every so often, as in:

```
while true
do
     command
     sleep 37
done
```

SEE ALSO

alarm(2), sleep(3)

NOTES

Time must be less than 65536 seconds.

NAME

split - split a file into pieces

SYNTAX

split [-*n*] [file [name]]

DESCRIPTION

Split reads *file* and writes it in *n*-line pieces (default 1000), as many as necessary, onto a set of output files. The name of the first output file is *name* with **aa** appended, and so on lexicographically. If no output name is given, x is default.

If no input file is given, or if - is given in its stead, then the standard input file is used.

WARNING

1000 lines is usually less than 19 pages.
Lpr does not guarantee that it prints the files in the order given.

SEE ALSO

lpr (1), wc (1)

NAME

strings - find the printable strings in a object, or other binary, file

SYNTAX

strings [-] [-o] [-*number*] file ...

DESCRIPTION

Strings looks for ascii strings in a binary file. A string is any sequence of 4 or more printing characters ending with a newline or a null. 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 offset in the file (in octal). 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

od(1)

AUTHOR

Bill Joy

NOTES

The algorithm for identifying strings is extremely primitive

NAME

sort - sort or merge files

SYNTAX

sort [-mubdfinrtx] [+pos1 [-pos2]] ... [-o name] [-T directory] [name] ...

DESCRIPTION

Sort sorts lines of all the named files together and writes the result on the standard output. The name `-' means the standard input. If no input files are named, the standard input is sorted.

The default sort key is an entire line. Default ordering is lexicographic by bytes in machine collating sequence. The ordering is affected globally by the following options, one or more of which may appear.

b Ignore leading blanks (spaces and tabs) in field comparisons.

d `Dictionary' order: only letters, digits and blanks are significant in comparisons.

f Fold upper case letters onto lower case.

i Ignore characters outside the ASCII range 040-0176 in nonnumeric comparisons.

n An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value. Option **n** implies option **b**.

r Reverse the sense of comparisons.

t*x* `Tab character' separating fields is *x*.

The notation +*pos1* -*pos2* restricts a sort key to a field beginning at *pos1* and ending just before *pos2*. *Pos1* and *pos2* each have the form *m*.*n*, optionally followed by one or more of the flags bdfinr, where *m* tells a number of fields to skip from the beginning of the line and *n* tells a number of characters to skip further. If any flags are present they override all the global ordering options for this key. If the **b** option is in effect *n* is counted from the first nonblank in the field; **b** is attached independently to *pos2*. A missing .*n* means .0; a missing -*pos2* means the end of the line. Under the **-t***x* option, fields are strings separated by *x*; otherwise fields are nonempty nonblank strings separated by blanks.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Lines that otherwise compare equal are ordered with all bytes significant.

These option arguments are also understood:

c	Check that the input file is sorted according to the ordering rules; give no output unless the file is out of sort.
m	Merge only, the input files are already sorted.
o	The next argument is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs.
T	The next argument is the name of a directory in which temporary files should be made.
u	Suppress all but one in each set of equal lines. Ignored bytes and bytes outside keys do not participate in this comparison.

Examples. Print in alphabetical order all the unique spellings in a list of words. Capitalized words differ from uncapitalized.

```
sort -u +0f +0 list
```

Print the password file (passwd(5)) sorted by user id number (the 3rd colon-separated field).

```
sort -t: +2n /etc/passwd
```

Print the first instance of each month in an already sorted file of (month day) entries. The options **-um** with just one input file make the choice of a unique representative from a set of equal lines predictable.

```
sort -um +0 -1 dates
```

FILES

/usr/tmp/stm*, /tmp/*: first and second tries for temporary files

SEE ALSO

uniq(1), comm(1), rev(1), join(1)

DIAGNOSTICS

Comments and exits with nonzero status for various trouble conditions and for disorder discovered under option **-c**.

NOTES

Very long lines are silently truncated.

NAME

Sp - convert long and narrow standard input to a wider format standard output.

SYNTAX

sp [width]

DESCRIPTION

sp Prints input in 8 character-wide columns onto the standard output. The optional argument, if numeric, specifies the width of the output.

Sp does not shorten long lines, it merely concatenates and spaces short ones.

EXAMPLE

```
ls | sp
```

SEE ALSO

prep (1), col (1)

NAME

spell, spellin, spellout - find spelling errors

SYNTAX

```
spell [ option ] ...  [ file ] ...

/usr/src/cmd/spell/spellin [ list ]

/usr/src/cmd/spell/spellout [ -d ] list
```

DESCRIPTION

Spell collects words from the named documents, and looks them up in a spelling list. Words that neither occur among nor are derivable (by applying certain inflections, prefixes or suffixes) from words in the spelling list are printed on the standard output. If no files are named, words are collected from the standard input.

Spell ignores most _troff_, _tbl_ and _eqn_(1) constructions.

Under the -v option, all words not literally in the spelling list are printed, and plausible derivations from spelling list words are indicated.

Under the -b option, British spelling is checked. Besides preferring _centre_, _colour_, _speciality_, _travelled_, etc., this option insists upon _-ise_ in words like _standardise_, Fowler and the OED to the contrary notwithstanding.

Under the -x option, every plausible stem is printed with `=' for each word.

The spelling list is based on many sources, and while more haphazard than an ordinary dictionary, is also more effective in respect to proper names and popular technical words. Coverage of the specialized vocabularies of biology, medicine and chemistry is light.

Pertinent auxiliary files may be specified by name arguments, indicated below with their default settings. Copies of all output are accumulated in the history file. The stop list filters out misspellings (e.g. thier=thy-y+ier) that would otherwise pass.

Two routines help maintain the hash lists used by _spell_. Both expect a list of words, one per line, from the standard input. _Spellin_ adds the words on the standard input to the preexisting _list_ and places a new list on the standard output. If no _list_ is specified, the new list is created from scratch. _Spellout_ looks up each word in the standard input and prints on the standard output those that are missing from (or present on, with option -d) the hash list.

FILES

D=/usr/dict/hlist[ab]: hashed spelling lists, American & British
S=/usr/dict/hstop: hashed stop list
H=/usr/dict/spellhist: history file
/usr/lib/spell
deroff(1), sort(1), tee(1), sed(1)

NOTES

The spelling list's coverage is uneven; new installations will probably wish to monitor the output for several months to gather local additions.
British spelling was done by an American.

NAME

strip - remove 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 loader. Strip works directly upon the named file(s); nothing is written to the standard output. This is useful to save space after a program has been debugged.

The effect of *strip* is the same as use of the -s option of *ld*.

FILES

/tmp/stm? temporary file

SEE ALSO

ld(1)

NAME

stty - set terminal options

SYNTAX

stty [option ...]

DESCRIPTION

Stty sets certain I/O options on the current output terminal. With no argument, it reports the current settings of the options. The option strings are selected from the following set:

even	allow even parity
-even	disallow even parity
odd	allow odd parity
-odd	disallow odd parity
raw	raw mode input (no erase, kill, interrupt, quit, EOT; parity bit passed back)
-raw	negate raw mode
cooked	same as `-raw'
cbreak	make each character available to _read_(2) as received; no erase and kill
-cbreak	make characters available to _read_ only when newline is received
-nl	allow carriage return for new-line, and output CR-LF for carriage return or new-line
nl	accept only new-line to end lines
echo	echo back every character typed
-echo	do not echo characters
lcase	map upper case to lower case
-lcase	do not map case
-tabs	replace tabs by spaces when printing
tabs	preserve tabs
ek	reset erase and kill characters back to normal # and @
erase _c_	set erase character to _c_. _C_ can be of the form `^_X_' which is interpreted as a `control X'.
kill _c_	set kill character to _c_. `^_X_' works here also.
cr0 cr1 cr2 cr3	select style of delay for carriage return (see _ioctl_(2))
nl0 nl1 nl2 nl3	select style of delay for linefeed
tab0 tab1 tab2 tab3	select style of delay for tab
ff0 ff1	select style of delay for form feed
bs0 bs1	select style of delay for backspace
tty33	set all modes suitable for the Teletype Corporation Model 33 terminal.
tty37	set all modes suitable for the Teletype Corporation Model 37 terminal.
vt05	set all modes suitable for Digital Equipment Corp.

NAME

su - substitute user id temporarily

SYNTAX

su [userid]

DESCRIPTION

Su demands the password of the specified *userid*, and if it is given, changes to that *userid* and invokes the Shell *sh*(1) without changing the current directory or the user environment (see *environ*(5)). The new user ID stays in force until the Shell exits.

If no *userid* is specified, \`root' is assumed. To remind the super-user of his responsibilities, the Shell substitutes \`#' for its usual prompt.

SEE ALSO

sh(1)

NAME

print - pr to the line printer

SYNTAX

print file ...

DESCRIPTION

Print pr's a copy of each named file on the line printer. It is a one line shell script:

```
pr $* | lpr
```

SEE ALSO

lpr(UCB), pr(1)

NOTES

	VT05 terminal
tn300	set all modes suitable for a General Electric TermiNet 300
ti700	set all modes suitable for Texas Instruments 700 series terminal
tek	set all modes suitable for Tektronix 4014 terminal
hup	hang up dataphone on last close.
-hup	do not hang up dataphone on last close.
0	hang up phone line immediately

50 75 110 134 150 200 300 600 1200 1800 2400 4800 9600 exta extb
Set terminal baud rate to the number given, if possible. (These are the speeds supported by the DH-11 interface).

SEE ALSO

ioctl(2), tabs(1)

NAME

sum - sum and count blocks in a file

SYNTAX

sum file

DESCRIPTION

Sum calculates and prints a 16-bit checksum for the named file, and also prints the number of blocks in the file. It is typically used to look for bad spots, or to validate a file communicated over some transmission line.

SEE ALSO

wc(1)

DIAGNOSTICS

`Read error' is indistinuishable from end of file on most devices; check the block count.

NAME

tabs - set terminal tabs

SYNTAX

tabs [**-n**] [terminal]

DESCRIPTION

Tabs sets the tabs on a variety of terminals. Various of the terminal names given in *term*(7) are recognized; the default is, however, suitable for most 300 baud terminals. If the -n flag is present then the left margin is not indented as is normal.

SEE ALSO

stty(1), term(7)

NAME

tail - deliver the last part of a file

SYNTAX

tail ±number[lbc] [file]

DESCRIPTION

Tail copies the named file to the standard output beginning at a designated place. If no file is named, the standard input is used.

Copying begins at distance +*number* from the beginning, or -*number* from the end of the input. *Number* is counted in units of lines, blocks or characters, according to the appended option **l**, **b** or **c**. When no units are specified, counting is by lines.

SEE ALSO

dd(1)

NOTES

Tails relative to the end of the file are treasured up in a buffer, and thus are limited in length. Various kinds of anomalous behavior may happen with character special files.

NAME

tar - tape archiver

SYNTAX

tar [key] [name ...]

DESCRIPTION

Tar saves and restores files on magtape. Its actions are controlled by the _key_ argument. The _key_ is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file or directory names specifying which files are to be dumped or restored. In all cases, appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

The function portion of the key is specified by one of the following letters:

r The named files are written on the end of the tape. The **c** function implies this.

x The named files are extracted from the tape. If the named file matches a directory whose contents had been written onto the tape, this directory is (recursively) extracted. The owner, modification time, and mode are restored (if possible). If no file argument is given, the entire content of the tape is extracted. Note that if multiple entries specifying the same file are on the tape, the last one overwrites all earlier.

t The names of the specified files are listed each time they occur on the tape. If no file argument is given, all of the names on the tape are listed.

u The named files are added to the tape if either they are not already there or have been modified since last put on the tape.

c Create a new tape; writing begins on the beginning of the tape instead of after the last file. This command implies **r**.

The following characters may be used in addition to the letter which selects the function desired.

0,...,7 This modifier selects the drive on which the tape is mounted. The default is 1.

v Normally _tar_ does its work silently. The v (verbose) option causes it to type the name of each file it treats preceded by the function letter. With the **t** function, **v** gives more information about the tape entries than just the name.

w causes _tar_ to print the action to be taken followed by file name, then wait for user confirmation. If a word beginning with `y' is given, the action is performed. Any other input means don't do it.

f causes _tar_ to use the next argument as the name of the archive instead of /dev/mt?. If the name of the file is `-', tar writes to standard output or reads from standard input, whichever is appropriate. Thus, _tar_ can be used as the head or tail of a filter chain _Tar_ can also be used to move hierarchies with the command

```
  cd fromdir; tar cf - . | (cd todir; tar xf -)
```

b causes _tar_ to use the next argument as the blocking factor for tape records. The default is 1, the maximum is 20. This option should only be used with raw magnetic tape archives (See **f** above). The block size is determined automatically when reading tapes (key letters `x' and `t').

l tells _tar_ to complain if it cannot resolve all of the links to the files dumped. If this is not specified, no error messages are printed.

m tells _tar_ to not restore the modification times. The mod time will be the time of extraction.

s causes _tar_ to use the next argument as the size of a tape volume. The minimum value allowed is 500. This option is useful when the archive is not intended for a magnetic tape device, but for some fixed size device, such as floppy disk (See **f** above).

FILES

/dev/mt?
/tmp/tar*

DIAGNOSTICS

Complaints about bad key characters and tape read/write errors.
Complaints if enough memory is not available to hold the link tables.

EXAMPLES

To backup a disk directory tree to tape using raw I/O and a blocking factor of 20:

```
tar cfb /dev/rmtl 20 directory_name
```

To restore the above files from tape to disk:

```
tar xf /dev/rmtl directory_name
```

SEE ALSO

tp(1), dump(1), restor(1), copy(1), dd(1)

NOTES

There is no way to ask for the *n*-th occurrence of a file.
Tape errors are handled ungracefully.
The **u** option can be slow.
The **b** option should not be used with archives that are going to be updated. The current magtape driver cannot backspace raw magtape. If the archive is on a disk file the **b** option should not be used at all, as updating an archive stored in this manner can destroy it.
The current limit on file name length is 100 characters.

NAME

tbl - format tables for nroff or troff

SYNTAX

tbl [files] ...

DESCRIPTION

Tbl is a preprocessor for formatting tables for _nroff_ or _troff_(1). The input files are copied to the standard output, except for lines between .TS and .TE command lines, which are assumed to describe tables and reformatted. Details are given in the reference manual.

As an example, letting \t represent a tab (which should be typed as a genuine tab) the input

```
.TS
c s s
c c s
c c c
l n n.
Household Population
Town\tHouseholds
\tNumber\tSize
Bedminster\t789\t3.26
Bernards Twp.\t3087\t3.74
Bernardsville\t2018\t3.30
Bound Brook\t3425\t3.04
Branchburg\t1644\t3.49
Bridgewater\t7897\t3.81
Far Hills\t240\t3.19
.TE
```

yields

```
    Household Population
    Town         Households
                Number   Size
Bedminster        789    3.26
Bernards Twp.    3087    3.74
Bernardsville    2018    3.30
Bound Brook      3425    3.04
Branchburg       1644    3.49
Bridgewater      7897    3.81
Far Hills         240    3.19
```

If no arguments are given, _tbl_ reads the standard input, so it may be used as a filter. When it is used with _eqn_ or _neqn_ the _tbl_ command should be first, to minimize the volume of data passed through pipes.

SEE ALSO

troff(1), eqn(1)
M. E. Lesk, _TBL_.

NAME

tc - photypesetter simulator

SYNTAX

tc [-t] [-sN] [-pL] [file]

DESCRIPTION

_Tc_ interprets its input (standard input default) as device codes for a Graphic Systems phototypesetter (cat). The standard output of _tc_ is intended for a Tektronix 4015 (a 4014 teminal with ASCII and APL character sets). The sixteen typesetter sizes are mapped into the 4014's four sizes; the entire TROFF character set is drawn using the 4014's character generator, using overstruck combinations where necessary. Typical usage:

```
troff -t file | tc
```

At the end of each page _tc_ waits for a newline (empty line) from the keyboard before continuing on to the next page. In this wait state, the command **e** will suppress the screen erase before the next page; **sN** will cause the next N pages to be skipped; and **!line** will send line to the shell.

The command line options are:

-t Don't wait between pages; for directing output into a file.

-sN Skip the first N pages.

-pL Set page length to L. L may include the scale factors **p** (points), **i** (inches), **c** (centimeters), and **P** (picas); default is picas.

'-_l_ _w_'
Multiply the default aspect ratio, 1.5, of a displayed page by _l_/_w_.

SEE ALSO

troff(1), plot(1)

NOTES

Font distinctions are lost.
The aspect ratio option is unbelievable.

NAME
tee - pipe fitting

SYNTAX
tee [-i] [-a] [file] ...

DESCRIPTION
Tee transcribes the standard input to the standard output and makes copies in the *files*. Option **-i** ignores interrupts; option -a causes the output to be appended to the *files* rather than overwriting them.

NAME

test - condition command

SYNTAX

test expr

DESCRIPTION

test evaluates the expression _expr_, and if its value is true then returns zero exit status; otherwise, a non zero exit status is returned. _test_ returns a non zero exit if there are no arguments.

The following primitives are used to construct _expr_.

-r file	true if the file exists and is readable.
-w file	true if the file exists and is writable.
-f file	true if the file exists and is not a directory.
-d file	true if the file exists and is a directory.
-s file	true if the file exists and has a size greater than zero.
-t [fildes]	true if the open file whose file descriptor number is _fildes_ (1 by default) is associated with a terminal device.
-z s1	true if the length of string _s1_ is zero.
-n s1	true if the length of the string _s1_ is nonzero.
s1 = s2	true if the strings _s1_ and _s2_ are equal.
s1 != s2	true if the strings _s1_ and _s2_ are not equal.
s1	true if _s1_ is not the null string.
n1 -eq n2	true if the integers _n1_ and _n2_ are algebraically equal. Any of the comparisons **-ne**, **-gt**, **-ge**, **-lt**, or **-le** may be used in place of **-eq**.

These primaries may be combined with the following operators:

!	unary negation operator
-a	binary _and_ operator

-o binary _or_ operator

(expr)
parentheses for grouping.

-a has higher precedence than -o. Notice that all the operators and flags are separate arguments to _test_. Notice also that parentheses are meaningful to the Shell and must be escaped.

SEE ALSO

sh(1), find(1)

NAME

time - time 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 execution time can depend on what kind of memory the program happens to land in; the user time in MOS is often half what it is in core.

The times are printed on the diagnostic output stream.

NOTES

Elapsed time is accurate to the second, while the CPU times are measured to the 60th second. Thus the sum of the CPU times can be up to a second larger than the elapsed time.

NAME

tk - paginator for the Tektronix 4014

SYNTAX

tk [-t] [-_N_] [-p_L_] [file]

DESCRIPTION

The output of _tk_ is intended for a Tektronix 4014 terminal. _Tk_ arranges for 66 lines to fit on the screen, divides the screen into _N_ columns, and contributes an eight space page offset in the (default) single-column case. Tabs, spaces, and backspaces are collected and plotted when necessary. Teletype Model 37 half- and reverse-line sequences are interpreted and plotted. At the end of each page _tk_ waits for a newline (empty line) from the keyboard before continuing on to the next page. In this wait state, the command !_command_ will send the _command_ to the shell.

The command line options are:

-t Don't wait between pages; for directing output into a file.

-_N_ Divide the screen into _N_ columns and wait after the last column.

-p_L_ Set page length to _L_ lines.

SEE ALSO

pr(1)

NAME
touch - update date last modified of a file

SYNTAX
touch [-c] file ...

DESCRIPTION
Touch attempts to set the modified date of each *file*. This is done by reading a character from the file and writing it back.

If a *file* does not exist, an attempt will be made to create it unless the **-c** option is specified.

NAME

tp - manipulate tape archive

SYNTAX

tp [key] [name ...]

DESCRIPTION

Tp saves and restores files on DECtape or magtape. Its actions are controlled by the *key* argument. The key is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file or directory names specifying which files are to be dumped, restored, or listed. In all cases, appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

The function portion of the key is specified by one of the following letters:

r The named files are written on the tape. If files with the same names already exist, they are replaced. `Same' is determined by string comparison, so `./abc' can never be the same as `/usr/dmr/abc' even if `/usr/dmr' is the current directory. If no file argument is given, `.' is the default.

u updates the tape. u is like r, but a file is replaced only if its modification date is later than the date stored on the tape; that is to say, if it has changed since it was dumped. u is the default command if none is given.

d deletes the named files from the tape. At least one name argument must be given. This function is not permitted on magtapes.

x extracts the named files from the tape to the file system. The owner and mode are restored. If no file argument is given, the entire contents of the tape are extracted.

t lists the names of the specified files. If no file argument is given, the entire contents of the tape is listed.

The following characters may be used in addition to the letter which selects the function desired.

m Specifies magtape as opposed to DECtape.

0,...,7 This modifier selects the drive on which the tape is mounted. For DECtape, **x** is default; for magtape `0' is the default.

v	Normally _tp_ does its work silently. The **v** (verbose) option causes it to type the name of each file it treats preceded by the function letter. With the **t** function, **v** gives more information about the tape entries than just the name.
c	means a fresh dump is being created; the tape directory is cleared before beginning. Usable only with **r** and **u**. This option is assumed with magtape since it is impossible to selectively overwrite magtape.
i	Errors reading and writing the tape are noted, but no action is taken. Normally, errors cause a return to the command level.
f	Use the first named file, rather than a tape, as the archive. This option is known to work only with **x**.
w	causes _tp_ to pause before treating each file, type the indicative letter and the file name (as with v) and await the user's response. Response **y** means `yes', so the file is treated. Null response means `no', and the file does not take part in whatever is being done. Response **x** means `exit'; the _tp_ command terminates immediately. In the **x** function, files previously asked about have been extracted already. With **r**, **u**, and **d** no change has been made to the tape.

FILES

/dev/tap?
/dev/mt?

SEE ALSO

ar(1), tar(1)

DIAGNOSTICS

Several; the non-obvious one is `Phase error', which means the file changed after it was selected for dumping but before it was dumped.

NOTES

A single file with several links to it is treated like several files.

Binary-coded control information makes magnetic tapes written by _tp_ difficult to carry to other machines; _tar_(1) avoids the problem.

NAME

tr - translate characters

SYNTAX

tr [-cds] [string1 [string2]]

DESCRIPTION

_Tr_ copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in _string1_ are mapped into the corresponding characters of _string2_. When _string2_ is short it is padded to the length of _string1_ by duplicating its last character. Any combination of the options **-cds** may be used: **-c** complements the set of characters in _string1_ with respect to the universe of characters whose ASCII codes are 01 through 0377 octal; **-d** deletes all input characters in _string1_; **-s** squeezes all strings of repeated output characters that are in _string2_ to single characters.

In either string the notation _a-b_ means a range of characters from _a_ to _b_ in increasing ASCII order. The character `\' followed by 1, 2 or 3 octal digits stands for the character whose ASCII code is given by those digits. A `\' followed by any other character stands for that character.

The following example creates a list of all the words in `file1' one per line in `file2', where a word is taken to be a maximal string of alphabetics. The second string is quoted to protect `\' from the Shell. 012 is the ASCII code for newline.

```
     tr -cs A-Za-z '\012' <file1 >file2
```

SEE ALSO

ed(1), ascii(7)

NOTES

Won't handle ASCII NUL in _string1_ or _string2_; always deletes NUL from input.

NAME

true, false - provide truth values

SYNTAX

true

false

DESCRIPTION

True does nothing, successfully. *False* does nothing, unsuccessfully. They are typically used in input to *sh*(1) such as:

```
while true
do
     command
done
```

SEE ALSO

sh(1)

DIAGNOSTICS

True has exit status zero, *false* nonzero.

NAME

tset - set terminal modes

SYNTAX

tset [**-**] [**-hrsuIQS**] [**-e**[c]] [**-E**[c]] [**-k**[c]]
[**-m** [ident][test baudrate]:type] [type]

DESCRIPTION

Tset causes terminal dependent processing such as setting erase and kill characters, setting or resetting delays, and the like. It is driven by the /etc/ttytype and /etc/termcap files.

The type of terminal is specified by the type argument. The type may be any type given in /etc/termcap. If type is not specified, the terminal type is read from /etc/htmp (the home directory and terminal type database), or the environment TERM, unless the **-h** flag is set or any **-m** argument was given. In this case the type is read from /etc/ttytype (the port name to terminal type database). The port name is determined by a ttyname(3) call on the diagnostic output. If the port is not found in /etc/ttytype the terminal type is set to unknown.

Ports for which the terminal type is indeterminate are identified in /etc/ttytype as dialup, plugboard, etc. The user can specify how these identifiers should map to an actual terminal type. The mapping flag, **-m**, is followed by the appropriate identifier (a 4 character or longer substring is adequate), an optional test for baud rate, and the terminal type to be used if the mapping conditions are satisfied. If more than one mapping is specified, the first correct mapping prevails. A missing identifier matches all identifiers. Baud rates are specified as with stty(1), and are compared with the speed of the diagnostic output. The test may be any combination of: **>**, **=**, **<**, **@**, and **!**. (Note: **@** is a synonym for = and ! inverts the sense of the test. Remember to escape characters meaningful to the shell.)

If the type as determined above begins with a question mark, the user is asked if s/he really wants that type. A null response means to use that type; otherwise, another type can be entered which will be used instead. (The question mark must be escaped to prevent filename expansion by the shell.)

On terminals that can backspace but not overstrike (such as a CRT), and when the erase character is the default erase character (`#' on standard systems), the erase character is changed to a Control-H (backspace). The -e flag sets the erase character to be the named character c on all terminals, so to override this option one can say -e#. The default for c is the backspace character on the terminal, usually Control-H. The **-E** flag is identical to **-e** except that it only operates on terminals that can backspace; it might be used if you had the misfortune to be stuck with an ASR33. The **-k** option works similarly, with c defaulting to Control-X. No kill processing is done if **-k** is not specified. In all of these flags, ``^X'' where X is any character is equivalent to control-X.

On version 6 systems, the terminal type specified in *htmp* is updated unless **-u** is specified.

The **-** option prints the terminal type on the standard output; this can be used to get the terminal type by saying:

```
     set termtype = `tset -`
```

If no other options are given, *tset* operates in ``fast mode'' and *only* outputs the terminal type, bypassing all other processing. The **-s** option outputs ``setenv'' commands (if your default shell is csh) or ``export'' and assignment commands (if your default shell is the Bourne shell); the **-S** option only outputs the strings to be placed in the environment variables. The **-s** option can be used as:

```
     `tset -s ...`
```

Actually, this is not possible because of a problem in the shell. Instead, if you are using the Bourne shell, use:

```
     tset -s ... > /tmp/tset$$
     /tmp/tset$$
     rm /tmp/tset$$
```

If you are using csh, use:

```
     set noglob
     set term=(`tset -S ....`)
     setenv TERM $term[1]
     setenv TERMCAP "$term[2]"
     unset term
     unset noglob
```

The **-r** option prints the terminal type on the diagnostic output. The **-Q** option supresses printing the ``Erase set to'' and ``Kill set to'' messages. The **-I** option supresses outputing the terminal initialization strings.

Tset is most useful when included in the **.login** (for *csh*(1)) or **.profile** (for *sh*(1)) file executed automatically at login, with **-m** mapping used to specify the terminal type you most frequently dial in on.

EXAMPLES

```
     tset gt42
     tset -mdialup\>300:adm3a -mdialup:dw2 -Qr -e#
     tset -m dial:ti733 -m plug:\?hp2621 -m unknown:\? -e -k^U
```

FILES

/etc/htmp Terminal type database (version 6 only)

/etc/ttytype	Port name to terminal type map database
/etc/termcap	Terminal capability database

SEE ALSO

setenv(1), ttytype(5), termcap(5), stty(1)

AUTHOR

Eric P. Allman

NOTES

For compatibility with earlier versions of tset, the following flags are accepted and mapped internally as shown:

```
-d type   ->   -m dialup:type
-p type   ->   -m plugboard:type
-a type   ->   -m arpanet:type
```

These flags will disappear eventually.

NAME

tsort - topological sort

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(1)

DIAGNOSTICS

Odd data: there is an odd number of fields in the input file.

NOTES

Uses a quadratic algorithm; not worth fixing for the typical use of ordering a library archive file.

NAME
tty - get terminal name

SYNTAX
tty

DESCRIPTION
Tty prints the pathname of the user's terminal.

DIAGNOSTICS
`not a tty' if the standard input file is not a terminal.

NAME

uniq - report repeated lines in a file

SYNTAX

uniq [-udc [+n] [-n]] [input [output]]

DESCRIPTION

Uniq reads the input file comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder is written on the output file. Note that repeated lines must be adjacent in order to be found; see _sort_(1). If the -u flag is used, just the lines that are not repeated in the original file are output. The -d option specifies that one copy of just the repeated lines is to be written. The normal mode output is the union of the -u and -d mode outputs.

The -c option supersedes -u and -d and generates an output report in default style but with each line preceded by a count of the number of times it occurred.

The _n_ arguments specify skipping an initial portion of each line in the comparison:

-_n_ The first _n_ fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.

+_n_ The first _n_ characters are ignored. Fields are skipped before characters.

SEE ALSO

sort(1), comm(1)

NAME

units - conversion program

SYNTAX

units

DESCRIPTION

Units converts quantities expressed in various standard scales to their equivalents in other scales. It works interactively in this fashion:

```
You have: inch
You want: cm
      * 2.54000e+00
      / 3.93701e-01
```

A quantity is specified as a multiplicative combination of units optionally preceded by a numeric multiplier. Powers are indicated by suffixed positive integers, division by the usual sign:

```
You have: 15 pounds force/in2
You want: atm
      * 1.02069e+00
      / 9.79730e-01
```

Units only does multiplicative scale changes. Thus it can convert Kelvin to Rankine, but not Centigrade to Fahrenheit. Most familiar units, abbreviations, and metric prefixes are recognized, together with a generous leavening of exotica and a few constants of nature including:

pi	ratio of circumference to diameter
c	speed of light
e	charge on an electron
g	acceleration of gravity
force	same as g
mole	Avogadro's number
water	pressure head per unit height of water
au	astronomical unit

\`Pound' is a unit of mass. Compound names are run together, e.g. \`lightyear'. British units that differ from their US counterparts are prefixed thus: \`brgallon'. Currency is denoted \`belgiumfranc', \`britainpound', ...

For a complete list of units, \`cat /usr/lib/units'.

FILES

/usr/lib/units

NOTES

Don't base your financial plans on the currency conversions.

NAME

uucp, uulog - unix to unix copy

SYNOPSIS

uucp [option] ... source-file ... destination-file

uulog [option] ...

DESCRIPTION

Uucp copies files named by the source-file arguments to the destination-file argument. A file name may be a path name on your machine, or may have the form

system-name!pathname

where \`system-name' is taken from a list of system names which *uucp* knows about. Shell metacharacters ?*[] appearing in the pathname part will be expanded on the appropriate system.

Pathnames may be one of

(1) a full pathname;

(2) a pathname preceded by ~*user*; where *user* is a userid on the specified system and is replaced by that user's login directory;

(3) anything else is prefixed by the current directory.

If the result is an erroneous pathname for the remote system the copy will fail. If the destination-file is a directory, the last part of the source-file name is used.

Uucp preserves execute permissions across the transmission and gives 0666 read and write permissions (see *chmod*(2)).

The following options are interpreted by *uucp*.

-d Make all necessary directories for the file copy.

-c Use the source file when copying out rather than copying the file to the spool directory.

-m Send mail to the requester when the copy is complete.

Uulog maintains a summary log of *uucp* and *uux*(1) transactions in the file \`/usr/spool/uucp/LOGFILE' by gathering information from partial log files named \`/usr/spool/uucp/LOG.*.?'. It removes the partial log files.

The options cause _uulog_ to print logging information:

-s_sys_
    Print information about work involving system _sys_.

-u_user_
    Print information about work done for the specified _user_.

FILES

/usr/spool/uucp - spool directory
/usr/lib/uucp/* - other data and program files

SEE ALSO

uux(1), mail(1)
D. A. Nowitz, _Uucp Implementation Description_

WARNING

The domain of remotely accessible files can (and for obvious security reasons, usually should) be severely restricted. You will very likely not be able to fetch files by pathname; ask a responsible person on the remote system to send them to you. For the same reasons you will probably not be able to send files to arbitrary pathnames.

BUGS

All files received by _uucp_ will be owned by uucp.
The -m option will only work sending files or receiving a single file. (Receiving multiple files specified by special shell characters ?*[] will not activate the -m option.)

NAME

uux - unix to unix command execution

SYNOPSIS

uux [-] command-string

DESCRIPTION

Uux will gather 0 or more files from various systems, execute a command on a specified system and send standard output to a file on a specified system.

The command-string is made up of one or more arguments that look like a shell command line, except that the command and file names may be prefixed by system-name!. A null system-name is interpreted as the local system.

File names may be one of

(1) a full pathname;

(2) a pathname preceded by ~*xxx*; where *xxx* is a userid on the specified system and is replaced by that user's login directory;

(3) anything else is prefixed by the current directory.

The `-' option will cause the standard input to the *uux* command to be the standard input to the command-string.

For example, the command

```
uux "!diff usg!/usr/dan/fl pwba!/a4/dan/fl > !fi.diff"
```

will get the fl files from the usg and pwba machines, execute a *diff* command and put the results in fl.diff in the local directory.

Any special shell characters such as <>;| should be quoted either by quoting the entire command-string, or quoting the special characters as individual arguments.

FILES

/usr/uucp/spool - spool directory
/usr/uucp/* - other data and programs

SEE ALSO

uucp(1)
D. A. Nowitz, *Uucp implementation description*

WARNING

An installation may, and for security reasons generally will, limit the list of commands executable on behalf of an incoming request from _uux_. Typically, a restricted site will permit little other than the receipt of mail via _uux_.

BUGS

Only the first command of a shell pipeline may have a system-name!. All other commands are executed on the system of the first command.
The use of the shell metacharacter * will probably not do what you want it to do.
The shell tokens << and >> are not implemented.
There is no notification of denial of execution on the remote machine.

NAME

vi - screen oriented (visual) display editor based on ex

SYNTAX

vi [-t tag] [-r] [+_lineno_] name ...

DESCRIPTION

_Vi_ (visual) is a display oriented text editor based on _ex_(UCB). _Ex_ and _vi_ run the same code; it is possible to get to the command mode of _ex_ from within _vi_ and vice-versa.

The _Vi Quick Reference_ card and the _Introduction to Display Editing with Vi_ provide full details on using _vi_.

FILES

See _ex_(UCB).

SEE ALSO

ex (UCB), vi (UCB), ``Vi Quick Reference'' card, ``An Introduction to Display Editing with Vi''.

NOTES

Scans with / and ? begin on the next line, skipping the remainder of the current line.

Software tabs using ^T work only immediately after the _autoindent_.

Left and right shifts on intelligent terminals don't make use of insert and delete character operations in the terminal.

The _wrapmargin_ option can be fooled since it looks at output columns when blanks are typed. If a long word passes through the margin and onto the next line without a break, then the line won't be broken.

Insert/delete within a line can be slow if tabs are present on intelligent terminals, since the terminals need help in doing this correctly.

Occasionally inverse video scrolls up into the file from a diagnostic on the last line.

Saving text on deletes in the named buffers is somewhat inefficient.

The _source_ command does not work when executed as **:source**; there is no way to use the **:append**, **:change**, and **:insert** commands, since it is not possible to give more than one line of input to a **:** escape. To use these on a **:global** you must **Q** to _ex_ command mode, execute them, and then reenter the screen editor with _vi_ or _open_.

NAME

wait - await completion of process

SYNTAX

wait

DESCRIPTION

Wait until all processes started with & have completed, and report on abnormal terminations.

Because the *wait*(2) system call must be executed in the parent process, the Shell itself executes *wait*, without creating a new process.

SEE ALSO

sh(1)

NOTES

Not all the processes of a 3- or more-stage pipeline are children of the Shell, and thus can't be waited for.

NAME

wall - write to all users

SYNOPSIS

/etc/wall

DESCRIPTION

Wall reads its standard input until an end-of-file. It then sends this message, preceded by `Broadcast Message ...', to all logged in users.

The sender should be super-user to override any protections the users may have invoked.

FILES

/dev/tty?
/etc/utmp

SEE ALSO

mesg(1), write(1)

DIAGNOSTICS

`Cannot send to ...' when the open on a user's tty file fails.

NAME

wc - word count

SYNTAX

wc [**-lwc**] [name ...]

DESCRIPTION

Wc counts lines, words and characters in the named files, or in the standard input if no name appears. A word is a maximal string of characters delimited by spaces, tabs or newlines.

If the optional argument is present, just the specified counts (lines, words or characters) are selected by the letters **l**, **w**, or **c**.

NAME

who - who is on the system

SYNTAX

who [who-file] [am I]

DESCRIPTION

Who, without an argument, lists the login name, terminal name, and login time for each current UNIX user.

Without an argument, _who_ examines the /etc/utmp file to obtain its information. If a file is given, that file is examined. Typically the given file will be /usr/adm/wtmp, which contains a record of all the logins since it was created. Then _who_ lists logins, logouts, and crashes since the creation of the wtmp file. Each login is listed with user name, terminal name (with `/dev/' suppressed), and date and time. When an argument is given, logouts produce a similar line without a user name. Reboots produce a line with `x' in the place of the device name, and a fossil time indicative of when the system went down.

With two arguments, as in `who am I' (and also `who are you'), _who_ tells who you are logged in as.

FILES

/etc/utmp

SEE ALSO

getuid(2), utmp(5)

NAME

write - write to another user

SYNTAX

write user [ttyname]

DESCRIPTION

Write copies lines from your terminal to that of another user. When first called, it sends the message

```
Message from yourname yourttyname...
```

The recipient of the message should write back at this point. Communication continues until an end of file is read from the terminal or an interrupt is sent. At that point *write* writes `EOT' on the other terminal and exits.

If you want to write to a user who is logged in more than once, the *ttyname* argument may be used to indicate the appropriate terminal name.

Permission to write may be denied or granted by use of the *mesg* command. At the outset writing is allowed. Certain commands, in particular *nroff* and *pr*(1) disallow messages in order to prevent messy output.

If the character `!' is found at the beginning of a line, *write* calls the shell to execute the rest of the line as a command.

The following protocol is suggested for using *write*: when you first write to another user, wait for him to write back before starting to send. Each party should end each message with a distinctive signal-(o) for `over' is conventional- that the other may reply. (oo) for `over and out' is suggested when conversation is about to be terminated.

FILES

/etc/utmp	to find user
/bin/sh	to execute `!'

SEE ALSO

mesg(1), who(1), mail(1)

NAME

xsend, xget, enroll - secret mail

SYNTAX

xsend person
xget
enroll

DESCRIPTION

These commands implement a secure communication channel; it is like mail(1), but no one can read the messages except the intended recipient. The method embodies a public-key cryptosystem using knapsacks.

To receive messages, use enroll; it asks you for a password that you must subsequently quote in order to receive secret mail.

To receive secret mail, use xget. It asks for your password, then gives you the messages.

To send secret mail, use xsend in the same manner as the ordinary mail command. (However, it will accept only one target). A message announcing the receipt of secret mail is also sent by ordinary mail.

FILES

/usr/spool/secretmail/*.key: keys
/usr/spool/secretmail/*.[0-9]: messages

SEE ALSO

mail (1)

NOTES

It should be integrated with ordinary mail. The announcement of secret mail makes traffic analysis possible.

NAME

xstr - extract strings from C programs to implement shared strings

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 approporiate 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
```

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(UCB)

AUTHOR

Bill Joy

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 - yet another 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*(1) 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
/lib/liby.a	library with default \`main' and \`yyerror'

SEE ALSO

lex(1)
LR Parsing by A. V. Aho and S. C. Johnson, Computing Surveys, June, 1974.
YACC - Yet Another Compiler Compiler by S. C. Johnson.

DIAGNOSTICS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard output; a more detailed report is found in the *y.output* file. Similarly, if some rules are not reachable from the start symbol, this is also reported.

NOTES

Because file names are fixed, at most one *yacc* process can be active in a given directory at a time.

NAME

intro, errno - introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

Section 2 of this manual lists all the entries into the system. Most of these calls have an error return. An error condition is indicated by an otherwise impossible returned value. Almost always this is -1; the individual sections specify the details. An error number is also made available in the external variable *errno*. *Errno* is set on erroneous calls; its value is undefined on successful calls.

There is a table of messages associated with each error, and a routine for printing the message; See *perror*(3). The possible error numbers are not recited with each writeup in section 2, since many errors are possible for most of the calls. Here is a list of the error numbers, their names as defined in <errno.h>, and the messages available using *perror*.

0 Error 0
Unused.

1 EPERM Not owner
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 ENOENT No such file or directory
This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

3 ESRCH No such process
The process whose number was given to *signal* and *ptrace* does not exist, or is already dead.

4 EINTR Interrupted system call
An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

5 EIO I/O error
Some physical I/O error occurred during a *read* or *write*. This error may in some cases occur on a call

following the one to which it actually applies.

6 ENXIO No such device or address
I/O on a special file refers to a subdevice that does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not dialled in or no disk pack is loaded on a drive.

7 E2BIG Arg list too long
An argument list longer than 5120 bytes is presented to _exec_.

8 ENOEXEC Exec format error
A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number, see _a.out_(5).

9 EBADF Bad file number
Either a file descriptor refers to no open file, or a read (resp. write) request is made to a file that is open only for writing (resp. reading).

10 ECHILD No children
Wait and the process has no living or unwaited-for children.

11 EAGAIN No more processes
In a _fork_, the system's process table is full or the user is not allowed to create any more processes.

12 ENOMEM Not enough core
During an _exec_ or _break_, a program asks for more core than the system is able to supply. This is not a temporary condition; the maximum core size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers.

13 EACCES Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

14 EFAULT Bad address
The system encountered a hardware fault in attempting to access the arguments of a system call.

15 ENOTBLK Block device required
A plain file was mentioned where a block device was required, e.g. in _mount_.

16 EBUSY Mount device busy
An attempt to mount a device that was already mounted

or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment).

17 EEXIST File exists
An existing file was mentioned in an inappropriate context, e.g. *link*.

18 EXDEV Cross-device link
A link to a file on another device was attempted.

19 ENODEV No such device
An attempt was made to apply an inappropriate system call to a device; e.g. read a write-only device.

20 ENOTDIR Not a directory
A non-directory was specified where a directory is required, for example in a path name or as an argument to *chdir*.

21 EISDIR Is a directory
An attempt to write on a directory.

22 EINVAL Invalid argument
Some invalid argument: dismounting a non-mounted device, mentioning an unknown signal in *signal*, reading or writing a file for which *seek* has generated a negative pointer. Also set by math functions, see *intro*(3).

23 ENFILE File table overflow
The system's table of open files is full, and temporarily no more *opens* can be accepted.

24 EMFILE Too many open files
Customary configuration limit is 20 per process.

25 ENOTTY Not a typewriter
The file mentioned in *stty* or *gtty* is not a terminal or one of the other devices to which these calls apply.

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program that is currently open for writing (or reading!). Also an attempt to open for writing a pure-procedure program that is being executed.

27 EFBIG File too large
The size of a file exceeded the maximum (about 1.0E9 bytes).

28 ENOSPC No space left on device
During a *write* to an ordinary file, there is no free

space left on the device.

29 ESPIPE Illegal seek
An _lseek_ was issued to a pipe. This error should also be issued for other non-seekable devices.

30 EROFS Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK Too many links
An attempt to make more than 32767 links to a file.

32 EPIPE Broken pipe
A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

33 EDOM Math argument
The argument of a function in the math package (3M) is out of the domain of the function.

34 ERANGE Result too large
The value of a function in the math package (3M) is unrepresentable within machine precision.

35 EUCLEAN File structure not clean
An attempt was made to _mount_(2) a file system whose superblock is not flagged ``clean''.

SEE ALSO
intro(3)

ASSEMBLER
as /usr/include/sys.s file ...

The PDP11 assembly language interface is given for each system call. The assembler symbols are defined in `/usr/include/sys.s'.

Return values appear in registers r0 and r1; it is unwise to count on these registers being preserved when no value is expected. An erroneous call is always indicated by turning on the c-bit of the condition codes. The error number is returned in r0. The presence of an error is most easily tested by the instructions _bes_ and _bec_ (`branch on error set (or clear)'). These are synonyms for the _bcs_ and _bcc_ instructions.

NAME

access - determine accessibility of file

SYNOPSIS

```
access(name, mode)
char *name;
```

DESCRIPTION

Access checks the given file *name* for accessibility according to *mode*, which is 4 (read), 2 (write) or 1 (execute) or a combination thereof. Specifying mode 0 tests whether the directories leading to the file can be searched and the file exists.

An appropriate error indication is returned if *name* cannot be found or if any of the desired access modes would not be granted. On disallowed accesses -1 is returned and the error code is in *errno*. 0 is returned from successful tests.

The user and group IDs with respect to which permission is checked are the real UID and GID of the process, so this call is useful to set-UID programs.

Notice that it is only access bits that are checked. A directory may be announced as writable by *access*, but an attempt to open it for writing will fail (although files may be created there); a file may look executable, but *exec* will fail unless it is in proper format.

SEE ALSO

stat(2)

ASSEMBLER

```
(access = 33.)
sys access; name; mode
```

NAME

acct - turn accounting on or off

SYNOPSIS

```
acct(file)
char *file;
```

DESCRIPTION

The system is prepared to write a record in an accounting _file_ for each process as it terminates. This call, with a null-terminated string naming an existing file as argument, turns on accounting; records for each terminating process are appended to _file_. An argument of 0 causes accounting to be turned off.

The accounting file format is given in _acct_(5).

SEE ALSO

acct(5), sa(1)

DIAGNOSTICS

On error -1 is returned. The file must exist and the call may be exercised only by the super-user. It is erroneous to try to turn on accounting when it is already on.

BUGS

No accounting is produced for programs running when a crash occurs. In particular nonterminating programs are never accounted for.

ASSEMBLER

```
(acct = 51.)
sys acct; file
```

NAME

alarm - schedule signal after specified time

SYNOPSIS

```
alarm(seconds)
unsigned seconds;
```

DESCRIPTION

Alarm causes signal SIGALRM, see *signal*(2), to be sent to the invoking process in a number of seconds given by the argument. Unless caught or ignored, the signal terminates the process.

Alarm requests are not stacked; successive calls reset the alarm clock. If the argument is 0, any alarm request is cancelled. Because the clock has a 1-second resolution, the signal may occur up to one second early; because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 65535 seconds.

The return value is the amount of time previously remaining in the alarm clock.

SEE ALSO

pause(2), signal(2), sleep(3)

ASSEMBLER

```
(alarm = 27.)
(seconds in r0)
sys alarm
(previous amount in r0)
```

NAME

brk, sbrk, break - change core allocation

SYNOPSIS

```
char *brk(addr)

char *sbrk(incr)
```

DESCRIPTION

Brk sets the system's idea of the lowest location not used by the program (called the break) to _addr_ (rounded up to the next multiple of 64 bytes on the PDP11, 256 bytes on the Interdata 8/32, 512 bytes on the VAX-11/780). Locations not less than _addr_ and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

In the alternate function _sbrk_, _incr_ more bytes are added to the program's data space and a pointer to the start of the new area is returned.

When a program begins execution via _exec_ the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use _break_.

SEE ALSO

exec(2), malloc(3), end(3)

DIAGNOSTICS

Zero is returned if the break could be set; -1 if the program requests more memory than the system limit or if too many segmentation registers would be required to implement the break.

BUGS

Setting the break in the range 0177701 to 0177777 (on the PDP11) is the same as setting it to zero.

ASSEMBLER

```
(break = 17.)
sys break; addr
```

Break performs the function of _brk_. The name of the routine differs from that in C for historical reasons.

NAME

chdir, chroot - change default directory

SYNOPSIS

```
chdir(dirname)
char *dirname;

chroot(dirname)
char *dirname;
```

DESCRIPTION

Dirname is the address of the pathname of a directory, terminated by a null byte. *Chdir* causes this directory to become the current working directory, the starting point for path names not beginning with `/'.

Chroot sets the root directory, the starting point for path names beginning with `/'. The call is restricted to the super-user.

SEE ALSO

cd(1)

DIAGNOSTICS

Zero is returned if the directory is changed; -1 is returned if the given name is not that of a directory or is not searchable.

ASSEMBLER

```
(chdir = 12.)
sys chdir; dirname

(chroot = 61.)
sys chroot; dirname
```

NAME

chmod - change mode of file

SYNOPSIS

```
chmod(name, mode)
char *name;
```

DESCRIPTION

The file whose name is given as the null-terminated string pointed to by name has its mode changed to mode. Modes are constructed by ORing together some combination of the following:

04000	set user ID on execution
02000	set group ID on execution
01000	save text image after execution
00400	read by owner
00200	write by owner
00100	execute (search on directory) by owner
00070	read, write, execute (search) by group
00007	read, write, execute (search) by others

If an executable file is set up for sharing (-n or -i option of ld(1)) then mode 1000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Thus when the next user of the file executes it, the text need not be read from the file system but can simply be swapped in, saving time. Ability to set this bit is restricted to the super-user since swap space is consumed by the images; it is only worth while for heavily used commands.

Only the owner of a file (or the super-user) may change the mode. Only the super-user can set the 1000 mode.

SEE ALSO

chmod(1)

DIAGNOSTIC

Zero is returned if the mode is changed; -1 is returned if name cannot be found or if current user is neither the owner of the file nor the super-user.

ASSEMBLER

```
(chmod = 15.)
sys chmod; name; mode
```

NAME

chown - change owner and group of a file

SYNOPSIS

```
chown(name, owner, group)
char *name;
```

DESCRIPTION

The file whose name is given by the null-terminated string pointed to by _name_ has its _owner_ and _group_ changed as specified. Only the super-user may execute this call, because if users were able to give files away, they could defeat the (nonexistent) file-space accounting procedures.

SEE ALSO

chown(1), passwd(5)

DIAGNOSTICS

Zero is returned if the owner is changed; -1 is returned on illegal owner changes.

ASSEMBLER

```
(chown = 16.)
sys chown; name; owner; group
```

NAME

close - close a file

SYNOPSIS

close(fildes)

DESCRIPTION

Given a file descriptor such as returned from an _open_, _creat_, _dup_ or _pipe_(2) call, _close_ closes the associated file. A close of all files is automatic on _exit_, but since there is a limit on the number of open files per process, _close_ is necessary for programs which deal with many files.

Files are closed upon termination of a process, and certain file descriptors may be closed by _exec_(2) (see _ioctl_(2)).

SEE ALSO

creat(2), open(2), pipe(2), exec(2), ioctl(2)

DIAGNOSTICS

Zero is returned if a file is closed; -1 is returned for an unknown file descriptor.

ASSEMBLER

```
(close = 6.)
(file descriptor in r0)
sys close
```

NAME

creat - create a new file

SYNOPSIS

```
creat(name, mode)
char *name;
```

DESCRIPTION

Creat creates a new file or prepares to rewrite an existing file called *name*, given as the address of a null-terminated string. If the file did not exist, it is given mode *mode*, as modified by the process's mode mask (see *umask*(2)). Also see *chmod*(2) for the construction of the *mode* argument.

If the file did exist, its mode and owner remain unchanged but it is truncated to 0 length.

The file is also opened for writing, and its file descriptor is returned.

The *mode* given is arbitrary; it need not allow writing. This feature is used by programs which deal with temporary files of fixed names. The creation is done with a mode that forbids writing. Then if a second instance of the program attempts a *creat*, an error is returned and the program knows that the name is unusable for the moment.

SEE ALSO

write(2), close(2), chmod(2), umask (2)

DIAGNOSTICS

The value -1 is returned if: a needed directory is not searchable; the file does not exist and the directory in which it is to be created is not writable; the file does exist and is unwritable; the file is a directory; there are already too many files open.

ASSEMBLER

```
(creat = 8.)
sys creat; name; mode
(file descriptor in r0)
```

the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

When a C program is executed, it is called as follows:

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

where _argc_ is the argument count and _argv_ is an array of character pointers to the arguments themselves. As indicated, _argc_ is conventionally at least one and the first member of the array points to a string containing the name of the file.

Argv is directly usable in another _execv_ because _argv_[_argc_] is 0.

Envp is a pointer to an array of strings that constitute the _environment_ of the process. Each string consists of a name, an ``='', and a null-terminated value. The array of pointers is terminated by a null pointer. The shell _sh_(1) passes an environment entry for each global shell variable defined when the program is called. See _environ_(5) for some conventionally used names. The C run-time start-off routine places a copy of _envp_ in the global cell _environ_, which is used by _execv_ and _execl_ to pass the environment to any subprograms executed by the current program. The _exec_ routines use lower-level routines as follows to pass an environment explicitly:

```
execle(file, arg0, arg1, . . . , argn, 0, environ);
execve(file, argv, environ);
```

Execlp and _execvp_ are called with the same arguments as _execl_ and _execv_, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

FILES

/bin/sh shell, invoked if command file found by _execlp_ or _execvp_

SEE ALSO

fork(2), environ(5)

DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not start with a valid magic number (see _a.out_(5)), if maximum memory is exceeded, or if the arguments require too much space, a return constitutes the diagnostic; the return value is -1. Even for the super-user, at least one of the

NAME

execl, execv, execle, execve, execlp, execvp, exec, exece, environ - execute a file

SYNOPSIS

```
execl(name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ..., *argn;

execv(name, argv)
char *name, *argv[ ];

execle(name, arg0, arg1, ..., argn, 0, envp)
char *name, *arg0, *arg1, ..., *argn, *envp[ ];

execve(name, argv, envp);
char *name, *argv[ ], *envp[ ];

extern char **environ;
```

DESCRIPTION

Exec in all its forms overlays the calling process with the named file, then transfers to the entry point of the core image of the file. There can be no return from a successful exec; the calling core image is lost.

Files remain open across _exec_ unless explicit arrangement has been made; see _ioctl_(2). Ignored signals remain ignored across these calls, but signals that are caught (see _signal_(2)) are reset to their default values.

Each user has a _real_ user ID and group ID and an _effective_ user ID and group ID. The real ID identifies the person using the system; the effective ID determines his access privileges. _Exec_ changes the effective user and group ID to the owner of the executed file if the file has the `set-user-ID' or `set-group-ID' modes. The real user ID is not affected.

The _name_ argument is a pointer to the name of the file to be executed. The pointers _arg_[_0_], _arg_[_1_] ... address null-terminated strings. Conventionally _arg_[_0_] is the name of the file.

From C, two interfaces are available. _Execl_ is useful when a known file with known arguments is being called; the arguments to _execl_ are the character strings constituting the file and the arguments; the first argument is conventionally the same as the file name (or its last component). A 0 argument must end the argument list.

The _execv_ version is useful when the number of arguments is unknown in advance; the arguments to _execv_ are the name of

NAME

dup, dup2 - duplicate an open file descriptor

SYNOPSIS

dup(fildes)
int fildes;

dup2(fildes, fildes2)
int fildes, fildes2;

DESCRIPTION

Given a file descriptor returned from an _open_, _pipe_, or _creat_ call, _dup_ allocates another file descriptor synonymous with the original. The new file descriptor is returned.

In the second form of the call, _fildes_ is a file descriptor referring to an open file, and _fildes2_ is a non-negative integer less than the maximum value allowed for file descriptors (approximately 19). _Dup2_ causes _fildes2_ to refer to the same file as _fildes_. If _fildes2_ already referred to an open file, it is closed first.

SEE ALSO

creat(2), open(2), close(2), pipe(2)

DIAGNOSTICS

The value -1 is returned if: the given file descriptor is invalid; there are already too many open files.

ASSEMBLER

```
(dup = 41.)
(file descriptor in r0)
(new file descriptor in r1)
sys dup
(file descriptor in r0)
```

The _dup2_ entry is implemented by adding 0100 to _fildes_.

execute-permission bits must be set for a file to be executed.

BUGS

If _execvp_ is called to execute a file that turns out to be a shell command file, and if it is impossible to execute the shell, the values of _argv_[_0_] and _argv_[-_1_] will be modified before return.

ASSEMBLER

(exec = 11.)
sys exec; name; argv

(exece = 59.)
sys exece; name; argv; envp

Plain _exec_ is obsoleted by _exece_, but remains for historical reasons.

When the called file starts execution on the PDP11, the stack pointer points to a word containing the number of arguments. Just above this number is a list of pointers to the argument strings, followed by a null pointer, followed by the pointers to the environment strings and then another null pointer. The strings themselves follow; a 0 word is left at the very top of memory.

```
  sp->    nargs
     arg0
     ...
     argn
     0
     env0
     ...
     envm
     0

 arg0:    <arg0\0>
     ...
 env0:    <env0\0>
     0
```

On the Interdata 8/32, the stack begins at a conventional place (currently 0xD0000) and grows upwards. After _exec_, the layout of data on the stack is as follows.

```
     int  0
 arg0:    byte ...
     ...
argp0:    int  arg0
     ...
     int  0
```

```
envp0:    int  env0
    ...
    int  0
 %2->     space    40
    int  nargs
    int  argp0
    int  envp0
 %3->
```

This arrangement happens to conform well to C calling conventions.

NAME
exit - terminate process

SYNOPSIS
```
exit(status)
int status;

_exit(status)
int status;
```

DESCRIPTION
Exit is the normal means of terminating a process. *Exit* closes all the process's files and notifies the parent process if it is executing a *wait*. The low-order 8 bits of *status* are available to the parent process.

This call can never return.

The C function *exit* may cause cleanup actions before the final `sys exit'. The function *_exit* circumvents all cleanup.

SEE ALSO
wait(2)

ASSEMBLER
```
(exit = 1.)
(status in r0)
sys exit
```

NAME

fork - spawn new process

SYNOPSIS

fork( )

DESCRIPTION

Fork is the only way new processes are created. The new process's core image is a copy of that of the caller of _fork_. The only distinction is the fact that the value returned in the old (parent) process contains the process ID of the new (child) process, while the value returned in the child is 0. Process ID's range from 1 to 30,000. This process ID is used by _wait_(2).

Files open before the fork are shared, and have a common read-write pointer. In particular, this is the way that standard input and output files are passed and also how pipes are set up.

SEE ALSO

wait(2), exec(2)

DIAGNOSTICS

Returns -1 and fails to create a process if: there is inadequate swap space, the user is not super-user and has too many processes, or the system's process table is full. Only the super-user can take the last process-table slot.

ASSEMBLER

```
(fork = 2.)
sys fork
(new process return)
(old process return, new process ID in r0)
```

The return locations in the old and new process differ by one word. The C-bit is set in the old process if a new process could not be created.

NAME

getpid - get process identification

SYNOPSIS

getpid( )

DESCRIPTION

Getpid returns the process ID of the current process. Most often it is used to generate uniquely-named temporary files.

SEE ALSO

mktemp(3)

ASSEMBLER

```
(getpid = 20.)
sys getpid
(pid in r0)
```

NAME

getuid, getgid, geteuid, getegid - get user and group identity

SYNOPSIS

getuid( )

geteuid( )

getgid( )

getegid( )

DESCRIPTION

Getuid returns the real user ID of the current process, _geteuid_ the effective user ID. The real user ID identifies the person who is logged in, in contradistinction to the effective user ID, which determines his access permission at the moment. It is thus useful to programs which operate using the `set user ID' mode, to find out who invoked them.

Getgid returns the real group ID, _getegid_ the effective group ID.

SEE ALSO

setuid(2)

ASSEMBLER

(getuid = 24.)
sys getuid
(real user ID in r0, effective user ID in r1)

(getgid = 47.)
sys getgid
(real group ID in r0, effective group ID in r1)

NAME

indir - indirect system call

ASSEMBLER

```
(indir = 0.)
sys indir; call
```

The system call at the location _call_ is executed. Execution resumes after the _indir_ call.

The main purpose of _indir_ is to allow a program to store arguments in system calls and execute them out of line in the data segment. This preserves the purity of the text segment.

If _indir_ is executed indirectly, it is a no-op. If the instruction at the indirect location is not a system call, _indir_ returns error code EINVAL; see _intro_(2).

NAME

ioctl, stty, gtty - control device

SYNOPSIS

```
#include <sgtty.h>

ioctl(fildes, request, argp)
struct sgttyb *argp;

stty(fildes, argp)
struct sgttyb *argp;

gtty(fildes, argp)
struct sgttyb *argp;
```

DESCRIPTION

Ioctl performs a variety of functions on character special files (devices). The writeups of various devices in section 4 discuss how *ioctl* applies to them.

For certain status setting and status inquiries about terminal devices, the functions *stty* and *gtty* are equivalent to

```
     ioctl(fildes, TIOCSETP, argp)
     ioctl(fildes, TIOCGETP, argp)
```

respectively; see *tty*(4).

The following two calls, however, apply to any open file:

```
     ioctl(fildes, FIOCLEX, NULL);
     ioctl(fildes, FIONCLEX, NULL);
```

The first causes the file to be closed automatically during a successful *exec* operation; the second reverses the effect of the first.

SEE ALSO

stty(1), tty(4), exec(2)

DIAGNOSTICS

Zero is returned if the call was successful; -1 if the file descriptor does not refer to the kind of file for which it was intended.

BUGS

Strictly speaking, since *ioctl* may be extended in different ways to devices with different properties, *argp* should have an open-ended declaration like

```
     union { struct sgttyb ...; ...  } *argp;
```

The important thing is that the size is fixed by struct sgttyb'.

ASSEMBLER

```
(ioctl = 54.)
sys ioctl; fildes; request; argp

(stty = 31.)
(file descriptor in r0)
stty; argp

(gtty = 32.)
(file descriptor in r0)
sys gtty; argp
```

NAME

kill - send signal to a process

SYNOPSIS

```
kill(pid, sig);
```

DESCRIPTION

Kill sends the signal _sig_ to the process specified by the process number in r0. See _signal_(2) for a list of signals.

The sending and receiving processes must have the same effective user ID, otherwise this call is restricted to the super-user.

If the process number is 0, the signal is sent to all other processes in the sender's process group; see _tty_(4).

If the process number is -1, and the user is the super-user, the signal is broadcast universally except to processes 0 and 1, the scheduler and initialization processes, see _init_(8).

Processes may send signals to themselves.

SEE ALSO

signal(2), kill(1)

DIAGNOSTICS

Zero is returned if the process is killed; -1 is returned if the process does not have the same effective user ID and the user is not super-user, or if the process does not exist.

ASSEMBLER

```
(kill = 37.)
(process number in r0)
sys kill; sig
```

NAME

link - link to a file

SYNOPSIS

```
link(name1, name2)
char *name1, *name2;
```

DESCRIPTION

A link to _name1_ is created; the link has the name _name2_. Either name may be an arbitrary path name.

SEE ALSO

ln(1), unlink(2)

DIAGNOSTICS

Zero is returned when a link is made; -1 is returned when _name1_ cannot be found; when _name2_ already exists; when the directory of _name2_ cannot be written; when an attempt is made to link to a directory by a user other than the super-user; when an attempt is made to link to a file on another file system; when a file has too many links.

ASSEMBLER

```
(link = 9.)
sys link; name1; name2
```

NAME

lock - lock a process in primary memory

SYNOPSIS

lock(flag)

DESCRIPTION

If the *flag* argument is non-zero, the process executing this call will not be swapped except if it is required to grow. If the argument is zero, the process is un*lock*ed. This call may only be executed by the super-user.

BUGS

*Lock*ed processes interfere with the compaction of primary memory and can cause deadlock. This system call is not considered a permanent part of the system.

ASSEMBLER

```
(lock = 53.)
sys lock; flag
```

NAME

lseek, tell - move read/write pointer

SYNOPSIS

```
long lseek(fildes, offset, whence)
long offset;

long tell(fildes)
```

DESCRIPTION

The file descriptor refers to a file open for reading or writing. The read (resp. write) pointer for the file is set as follows:

If *whence* is 0, the pointer is set to *offset* bytes.

If *whence* is 1, the pointer is set to its current location plus *offset*.

If *whence* is 2, the pointer is set to the size of the file plus *offset*.

The returned value is the resulting pointer location.

The obsolete function *tell*(*fildes*) is identical to *lseek*(*fildes*, *0L*, *1*).

Seeking far beyond the end of a file, then writing, creates a gap or `hole', which occupies no physical space and reads as zeros.

SEE ALSO

open(2), creat(2), fseek(3)

DIAGNOSTICS

-1 is returned for an undefined file descriptor, seek on a pipe, or seek to a position before the beginning of file.

BUGS

Lseek is a no-op on character special files.

ASSEMBLER

```
(lseek = 19.)
(file descriptor in r0)
sys lseek; offset1; offset2; whence
```

Offset1 and *offset2* are the high and low words of *offset*; r0 and r1 contain the pointer upon return.

NAME

mknod - make a directory or a special file

SYNOPSIS

```
mknod(name, mode, addr)
char *name;
```

DESCRIPTION

Mknod creates a new file whose name is the null-terminated string pointed to by *name*. The mode of the new file (including directory *and* special file bits) is initialized from *mode*. (The protection part of the mode is modified by the process's mode mask; see *umask*(2)). The first block pointer of the i-node is *initialized* from *addr*. For ordinary files and directories *addr* is normally *zero*. In the case of a special file, *addr* specifies which special file.

Mknod may be invoked only by the super-user.

SEE ALSO

mkdir(1), mknod(1), filsys(5)

DIAGNOSTICS

Zero is returned if the file has been made; -1 if the file already exists or if the user is not the super-user.

ASSEMBLER

```
(mknod = 14.)
sys mknod; name; mode; addr
```

NAME

mount, umount - mount or remove file system

SYNOPSIS

```
mount(special, name, rwflag)
char *special, *name;

umount(special)
char *special;
```

DESCRIPTION

Mount announces to the system that a removable file system has been mounted on the block-structured special file *special*; from now on, references to file *name* will refer to the root file on the newly mounted file system. *Special* and *name* are pointers to null-terminated strings containing the appropriate path names.

Name must exist already. *Name* must be a directory (unless the root of the mounted file system is not a directory). Its old contents are inaccessible while the file system is mounted.

The *rwflag* argument determines whether the file system can be written on; if it is 0 writing is allowed, if non-zero no writing is done. Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

Umount announces to the system that the *special* file is no longer to contain a removable file system. The associated file reverts to its ordinary interpretation. Any pending I/O for the file system is completed, and the file system is marked clean.

SEE ALSO

mount(1), intro(2), mknod(1M), fsck(1M)

DIAGNOSTICS

Mount returns 0 if the action occurred; -1 if *special* is inaccessible or not an appropriate file; if *name* does not exist; if *special* is already mounted; if *name* is in use; or if there are already too many file systems mounted. If the file system was unclean, `errno' == EUCLEAN. Use *fsck*(1m) to clean the file system.

Umount returns 0 if the action occurred; -1 if the special file is inaccessible or does not have a mounted file system, or if there are active files in the mounted file system.

ASSEMBLER

```
     (mount = 21.)
     sys mount; special; name; rwflag

     (umount = 22.)
     sys umount; special
```

NAME

mpx - create and manipulate multiplexed files

SYNOPSIS

```
mpx(name, access) char *name;

join(fd, xd)

chan(xd)

extract(i, xd)

attach(i, xd)

detach(i, xd)

connect(fd, cd, end)

npgrp(i, xd, pgrp)

ckill(i, xd, signal)

#include <sys/mx.h>
mpxcall(cmd, vec)
int *vec;
```

DESCRIPTION

mpxcall(cmd, vec) is the system call shared by the library routines described below. *Cmd* selects a command using values defined in <*sys*/*mx*.*h*>. *Vec* is the address of a structure containing the arguments for the command.

mpx(name, access)

Mpx creates and opens the file *name* with access permission *access* (see *creat*(2)) and returns a file descriptor available for reading and writing. A -1 is returned if the file cannot be created, if *name* already exists, or if the file table or other operating system data structures are full. The file descriptor is required for use with other routines.

If *name* designates a null string, a file descriptor is returned as described but no entry is created in the file system.

Once created an mpx file may be opened (see *open*(2)) by any process. This provides a form of interprocess communication whereby a process B can `call' process A by opening an mpx file created by A. To B, the file is ordinary with one exception: the *connect* primitive could be applied to it. Otherwise the functions described below are used only in process A and descendants that inherit the open mpx file.

When a process opens an mpx file, the owner of the file receives a control message when the file is next read. The method for `answering' this kind of call involves using attach and detach as described in more detail below.

Once B has opened A's mpx file it is said to have a channel to A. A channel is a pair of data streams: in this case, one from B to A and the other from A to B. Several processes may open the same mpx file yielding multiple channels within the one mpx file. By accessing the appropriate channel, A can communicate with B and any others. When A reads (see read(2)) from the mpx file data written to A by the other processes appears in A's buffer using a record format described in mpxio(5). When A writes (see write(2)) on its mpx file the data must be formatted in a similar way.

The following commands are used to manipulate mpx files and channels.

> join- adds a new channel on an mpx file to an open file F. I/O on the new channel is I/O on F.
> chan- creates a new channel.
> extract- file descriptor maintenance.
> connect- similar to join except that the open file F is connected to an existing channel.
> attach and detach- used with call protocol.
> npgrp- manipulates process group numbers so that a channel can act as a control terminal (see tty(4)).
> ckill- send signal (see signal(2)) to process group through channel.

A maximum of 15 channels may be connected to an mpx file. They are numbered 0 through 14. Join may be used to make one mpx file appear as a channel on another mpx file. A hierarchy or tree of mpx files may be set up in this way. In this case one of the mpx files must be the root of a tree where the other mpx files are interior nodes. The maximum depth of such a tree is 4.

An index is a 16-bit value that denotes a location in an mpx tree other than the root: the path through mpx `nodes' from the root to the location is expressed as a sequence of 4-bit nibbles. The branch taken at the root is represented by the low-order 4-bits of an index. Each succeeding branch is specified by the next higher-order nibble. If the length of a path to be expressed is less than 4, then the illegal channel number, 15, must be used to terminate the sequence. This is not strictly necessary for the simple case of a tree consisting of only a root node: its channels can be expressed by the numbers 0 through 14. An index i and file descriptor xd for the root of an mpx tree are required as arguments to most of the commands described below. Indices also serve as channel identifiers in the record formats given in _mpxio_(5). Since -1 is not a valid index, it can be returned as a error indication by subroutines that normally return indices.

The operating system informs the process managing an mpx file of changes in the status of channels attached to the file by generating messages that are read along with data from the channels. The form and content of these messages is described in _mpxio_(5).

join(fd, xd) establishes a connection (channel) between an mpx file and another object. _Fd_ is an open file descriptor for a character device or an _mpx_ file and _xd_ is the file descriptor of an mpx file. _Join_ returns the index for the new channel if the operation succeeds and -1 if it does not.

Following join, _fd_ may still be used in any system call that would have been meaningful before the join operation. Thus a process can read and write directly to _fd_ as well as access it via _xd_. If the number of channels required for a tree of mpx files exceeds the number of open files permitted a process by the operating system, some of the file descriptors can be released using the standard _close_(2) call. Following a close on an active file descriptor for a channel or internal mpx node, that object may still be accessed through the root of the tree.

chan(xd) allocates a channel and connects one end of it to the mpx file represented by file descriptor _xd_. _Chan_ returns the index of the new channel or a -1 indicating failure. The _extract_ primitive can be used to get a non-multiplexed file descriptor for the free end of a channel created by _chan_.

Both _chan_ and _join_ operate on the mpx file specified by _xd_. File descriptors for interior nodes of an mpx tree must be preserved or reconstructed with _extract_ for use with _join_ or _chan_. For the remaining commands described here, _xd_ denotes the file descriptor for the root of an mpx tree.

Extract(i, xd) returns a file descriptor for the object with index _i_ on the mpx tree with root file descriptor _xd_. A -1 is returned by extract if a file descriptor is not available or if the arguments do not refer to an existing channel and mpx file.

attach(i, xd)
detach(i, xd). If a process A has created an mpx file represented by file descriptor _xd_, then a process B can open (see _open_(2)) the mpx file. The purpose is to establish a channel between A and B through the mpx file. _Attach_ and

Detach are used by A to respond to such opens.

An open request by B fails immediately if a new channel cannot be allocated on the mpx file, if the mpx file does not exist, or if it does exist but there is no process (A) with a multiplexed file descriptor for the mpx file (i.e. *xd* as returned by *mpx*(2)). Otherwise a channel with index number *i* is allocated. The next time A reads on file descriptor *xd*, the WATCH control message (see *mpxio*(5)) will be delivered on channel *i*. A responds to this message with *attach* or *detach*. The former causes the open to complete and return a file descriptor to B. The latter deallocates channel *i* and causes the open to fail.

One mpx file may be placed in `listener' mode. This is done by writing *ioctl*(*xd*, *MXLSTN*, *0*) where *xd* is an mpx file descriptor and MXLSTN is defined in /*usr*/*include*/*sgtty*.*h*. The semantics of listener mode are that all file names discovered by *open*(2) to have the syntax *system*!*pathname* (see *uucp*(1)) are treated as opens on the mpx file. The operating system sends the listener process an OPEN message (see *mpxio*(5)) which includes the file name being opened. *Attach* and *detach* then apply as described above.

Detach has two other uses: it closes and releases the resources of any active channel it is applied to, and should be used to respond to a CLOSE message (see *mpxio*(5)) on a channel so the channel may be reused.

connect(fd, cd, end). *Fd* is a character file descriptor and *cd* is a file descriptor for a channel, such as might be obtained via *extract*(*chan*(*xd*), *xd*) or by *open*(2) followed by *attach*. *Connect* splices the two streams together. If *end* is negative, only the output of *fd* is spliced to the input of *cd*. If *end* is positive, the output of *cd* is spliced to the input of *fd*. If *end* is zero, then both splices are made.

npgrp(i, xd, pgrp). If *xd* is negative *npgrp* applies to the process executing it, otherwise *i* and *xd* are interpreted as a channel index and mpx file descriptor and *npgrp* is applied to the process on the non-multiplexed end of the channel. If *pgrp* is zero, the process group number of the indicated process is set to the process number of that process, otherwise the value of *pgrp* is used as the process group number.

Npgrp normally returns the new process group number. If *i* and *xd* specify a nonexistant channel, *npgrp* returns -1.

ckill(i, xd, signal) sends the specified signal (see *signal*(2)) through the channel specified by *i* and *xd*. If the channel is connected to anything other than a process, *ckill* is a null operation. If there is a process at the other end of the channel, the process group will be interrupted (see *signal*(2), *kill*(2)). *Ckill* normally returns *signal*. If *ch* and *xd* specify a nonexistent channel, *ckill* returns -1.

FILES

/usr/include/sys/mx.h
/usr/include/sgtty.h

SEE ALSO

mpxio(5)

BUGS

Mpx files are an experimental part of the operating system more subject to change and prone to bugs than other parts. Maintenance programs, e.g. *icheck*(1), diagnose mpx files as an illegal mode. Channels may only be connected to objects in the operating system that are accessible through the line discipline mechanism. Higher performace line disciplines are needed. The maximum tree depth restriction is not really checked. A non-destructive *disconnect* primitive (inverse of *connect*) is not provided. A non-blocking flow control strategy based on messages defined in *mpxio*(5) should not be attempted by novices; the enabling *ioctl* command should be protected. The *join* operation could be subsumed by *connect*. A mechanism is needed for moving a channel from one location in an mpx tree to another.

NAME

mpxcall - multiplexor and channel interface

SYNOPSIS

mpxcall(arg1, arg2, arg3, cmd)

DESCRIPTION

Mpxcall supplies a primitive interface to the kernel used by the routines listed below. Each routine that uses *mpxcall* passes an integer *cmd* as the fourth argument. These are defined in /*usr*/*include*/*mx*.*h*. *Mpxcall* always returns an integer which is to be interpreted in accordance with the definition of *cmd*.

SEE ALSO

group(2), join(2), extract(2), connect(2), chan(2), attach(2), detach(2)

DIAGNOSTICS

The value -1 is returned on error.

NAME

nice - set program priority

SYNOPSIS

nice(incr)

DESCRIPTION

The scheduling priority of the process is augmented by *incr*. Positive priorities get less service than normal. Priority 10 is recommended to users who wish to execute long-running programs without flak from the administration.

Negative increments are ignored except on behalf of the super-user. The priority is limited to the range -20 (most urgent) to 20 (least).

The priority of a process is passed to a child process by *fork*(2). For a privileged process to return to normal priority from an unknown state, *nice* should be called successively with arguments -40 (goes to priority -20 because of truncation), 20 (to get to 0), then 0 (to maintain compatibility with previous versions of this call).

SEE ALSO

nice(1)

ASSEMBLER

```
(nice = 34.)
(priority in r0)
sys nice
```

NAME

open - open for reading or writing

SYNOPSIS

```
open(name, mode)
char *name;
```

DESCRIPTION

Open opens the file *name* for reading (if *mode* is 0), writing (if *mode* is 1) or for both reading and writing (if *mode* is 2). *Name* is the address of a string of ASCII characters representing a path name, terminated by a null character.

The file is positioned at the beginning (byte 0). The returned file descriptor must be used for subsequent calls for other input-output functions on the file.

SEE ALSO

creat(2), read(2), write(2), dup(2), close(2)

DIAGNOSTICS

The value -1 is returned if the file does not exist, if one of the necessary directories does not exist or is unreadable, if the file is not readable (resp. writable), or if too many files are open.

ASSEMBLER

```
(open = 5.)
sys open; name; mode
(file descriptor in r0)
```

NAME

pause - stop until signal

SYNOPSIS

```
pause( )
```

DESCRIPTION

Pause never returns normally. It is used to give up control while waiting for a signal from *kill*(2) or *alarm*(2).

SEE ALSO

kill(1), kill(2), alarm(2), signal(2), setjmp(3)

ASSEMBLER

```
(pause = 29.)
sys pause
```

NAME

phys - allow a process to access physical addresses

SYNOPSIS

phys(segreg, size, physadr)

DESCRIPTION

The argument *segreg* specifies a process virtual (data-space) address range of 8K bytes starting at virtual address *segreg*x8K bytes. This address range is mapped into physical address *physadr*x64 bytes. Only the first *size*x64 bytes of this mapping is addressable. If *size* is zero, any previous mapping of this virtual address range is nullified. For example, the call

```
phys(6, 1, 0177775);
```

will map virtual addresses 0160000-0160077 into physical addresses 017777500-017777577. In particular, virtual address 0160060 is the PDP-11 console located at physical address 017777560.

This call may only be executed by the super-user.

SEE ALSO

PDP-11 segmentation hardware

DIAGNOSTICS

The function value zero is returned if the physical mapping is in effect. The value -1 is returned if not super-user, if *segreg* is not in the range 0-7, if *size* is not in the range 0-127, or if the specified *segreg* is already used for other than a previous call to *phys*.

BUGS

This system call is obviously very machine dependent and very dangerous. This system call is not considered a permanent part of the system.

ASSEMBLER

(phys = 52.)
sys phys; segreg; size; physadr

NAME

pipe - create an interprocess channel

SYNOPSIS

pipe(fildes)
int fildes[2];

DESCRIPTION

The *pipe* system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor *fildes*[1] up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor *fildes*[0] will pick up the data. Writes with a count of 4096 bytes or less are atomic; no other process can intersperse data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent *fork* calls) will pass data through the pipe with *read* and *write* calls.

The Shell has a syntax to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors closed) returns an end-of-file.

SEE ALSO

sh(1), read(2), write(2), fork(2)

DIAGNOSTICS

The function value zero is returned if the pipe was created; -1 if too many files are already open. A signal is generated if a write on a pipe with only one end is attempted.

BUGS

Should more than 4096 bytes be necessary in any pipe among a loop of processes, deadlock will occur.

ASSEMBLER

(pipe = 42.)
sys pipe
(read file descriptor in r0)
(write file descriptor in r1)

NAME

pkon, pkoff - establish packet protocol

SYNOPSIS

```
pkon(fd, size)

pkoff(fd)
```

DESCRIPTION

Pkon establishes packet protocol (see *pk*(4)) on the open character special file whose file descriptor is *fd*. *Size* is a desired packet size, a power of 2 in the range 32<*size*<4096. The size is negotiated with a remote packet driver, and a possibly smaller actual packet size is returned.

An asynchronous line used for packet communication should be in raw mode; see *tty*(4).

Pkoff turns off the packet driver on the channel whose file descriptor is *fd*.

SEE ALSO

pk(4), pkopen(3), tty(4), signal(2)

DIAGNOSTICS

Pkon returns -1 if *fd* does not describe an open file, or if packet communication cannot be established.

Pkoff returns -1 for an unknown file descriptor.

Writing on a packet driver link that has been shut down by *close* or *pkoff* at the other end raises signal SIGPIPE in the writing process.

NAME

profil - execution time profile

SYNOPSIS

profil(buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;

DESCRIPTION

Buff points to an area of core whose length (in bytes) is given by _bufsiz_. After this call, the user's program counter (_pc_) is examined each clock tick (60th second); _offset_ is subtracted from it, and the result multiplied by _scale_. If the resulting number corresponds to a word inside _buff_, that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777(8) gives a 1-1 mapping of pc's to words in _buff_; 077777(8) maps each pair of instruction words together. 02(8) maps all instructions onto the beginning of _buff_ (producing a non-interrupting core clock).

Profiling is turned off by giving a _scale_ of 0 or 1. It is rendered ineffective by giving a _bufsiz_ of 0. Profiling is turned off when an _exec_ is executed, but remains on in child and parent both after a _fork_. Profiling may be turned off if an update in _buff_ would cause a memory fault.

SEE ALSO

monitor(3), prof(1)

ASSEMBLER

(profil = 44.)
sys profil; buff; bufsiz; offset; scale

NOTES

You should use the monitor(3) call. The prof(1) program may require the buffer size to be equal to or smaller than the program size.

NAME

ptrace - process trace

SYNOPSIS

```
#include <signal.h>

ptrace(request, pid, addr, data)
int *addr;
```

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging. There are four arguments whose interpretation depends on a *request* argument. Generally, *pid* is the process ID of the traced process, which must be a child (no more distant descendant) of the tracing process. A process being traced behaves normally until it encounters some signal whether internally generated like `illegal instruction' or externally generated like `interrupt.' See *signal*(2) for the list. Then the traced process enters a stopped state and its parent is notified via *wait*(2). When the child is in the stopped state, its core image can be examined and modified using *ptrace*. If desired, another *ptrace* request can then cause the child either to terminate or to continue, possibly ignoring the signal.

The value of the *request* argument determines the precise action of the call:

0 This request is the only one used by the child process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

1,2 The word in the child process's address space at *addr* is returned. If I and D space are separated, request 1 indicates I space, 2 D space. *Addr* must be even. The child must be stopped. The input *data* is ignored.

3 The word of the system's per-process data area corresponding to *Addr* is returned. *Addr* must be even and less than 512. This space contains the registers and other information about the process; its layout corresponds to the *user* structure in the system.

4,5 The given *data* is written at the word in the process's address space corresponding to *addr*, which must be even. No useful value is returned. If I and D space are separated, request 4 indicates I space, 5 D space.

Attempts to write in pure procedure fail if another process is executing the same file.

6 The process's system data is written, as it is read with request 3. Only a few locations can be written in this way: the general registers, the floating point status and registers, and certain bits of the processor status word.

7 The _data_ argument is taken as a signal number and the child's execution continues at location _addr_ as if it had incurred that signal. Normally the _signal_ number will be either 0 to indicate that the signal that caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop. If _addr_ is (int *)1 then execution continues from where it stopped.

8 The traced process terminates.

9 Execution continues as in request 7; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. On the PDP-11 the T-bit is used and just one instruction is executed. This is part of the mechanism for implementing breakpoints. On other processors, appropriate machine-dependent strategies are used.

As indicated, these calls (except for request 0) can be used only when the subject process has stopped. The _wait_ call is used to determine when a process stops; in such a case the `termination' status returned by _wait_ has the value 0177 to indicate stoppage rather than genuine termination.

To forestall possible fraud, _ptrace_ inhibits the set-user-id facility on subsequent _exec_(2) calls. If a traced process calls _exec_, it will stop before executing the first instruction of the new image showing signal SIGTRAP.

SEE ALSO
wait(2), signal(2), adb(1)

DIAGNOSTICS
The value -1 is returned if _request_ is invalid, _pid_ is not a traceable process, _addr_ is out of bounds, or _data_ specifies an illegal signal number.

BUGS
The request 0 call should be able to specify signals which are to be treated normally and not cause a stop. In this way, for example, programs with simulated floating point

(which use `illegal instruction' signals at a very high rate) could be efficiently debugged.
The error indication, -1, is a legitimate function value; errno, see intro(2), can be used to disambiguate.

It should be possible to stop a process on occurrence of a system call; in this way a completely controlled environment could be provided.

Differing processors and configurations will necessarily create some differences in functionality.

PDP-11 ASSEMBLER

```
(ptrace = 26.)
(data in r0)
sys ptrace; pid; addr; request
(value in r0)
```

NAME

read - read from file

SYNOPSIS

read(fildes, buffer, nbytes)
char *buffer;

DESCRIPTION

A file descriptor is a word returned from a successful *open*, *creat*, *dup*, or *pipe* call. *Buffer* is the location of *nbytes* contiguous bytes into which the input will be placed. It is not guaranteed that all *nbytes* bytes will be read; for example if the file refers to a typewriter at most one line will be returned. In any event the number of characters read is returned.

If the returned value is 0, then end-of-file has been reached.

SEE ALSO

open(2), creat(2), dup(2), pipe(2)

DIAGNOSTICS

As mentioned, 0 is returned when the end of the file has been reached. If the read was otherwise unsuccessful the return value is -1. Many conditions can generate an error: physical I/O errors, bad buffer address, preposterous *nbytes*, file descriptor not that of an input file.

ASSEMBLER

(read = 3.)
(file descriptor in r0)
sys read; buffer; nbytes
(byte count in r0)

NAME

setuid, setgid - set user and group ID

SYNOPSIS

setuid(uid)

setgid(gid)

DESCRIPTION

The user ID (group ID) of the current process is set to the argument. Both the effective and the real ID are set. These calls are only permitted to the super-user or if the argument is the real ID.

SEE ALSO

getuid(2)

DIAGNOSTICS

Zero is returned if the user (group) ID is set; -1 is returned otherwise.

ASSEMBLER

```
(setuid = 23.)
(user ID in r0)
sys setuid

(setgid = 46.)
(group ID in r0)
sys setgid
```

NAME

shutdn - flush block I/O and halt CPU

SYNOPSIS

```
#include <sys/filsys.h>

shutdn (sblk, ~sblk)
struct filsys *sblk;
```

DESCRIPTION

Shutdn causes all information in core memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O. The super-blocks of all writable file systems are flagged `clean', so that they can be remounted without cleaning when XENIX is rebooted. *Shutdn* then prints ``Normal System Shutdown'' on the console and halts the cpu.

If *sblk* is non-zero, it specifies the address of a super block which will be written to the root device as the last I/O before the halt. This facility is provided to allow file system repair programs to supercede the system's copy of the root super block with one of their own. The address of the new super block is boolean inverted to form the second argument. This is an attempt to reduce the liklihood of a program accidentally invoking *shutdn* and destroying the root file system.

Shutdn locks out all other processes while it is doing it's work. However, it is recommended that user processes be killed off (see *kill*(1)) before calling *shutdn* as some types of disk activity could cause file systems to not be flagged `clean'.

The caller must be the super-user.

SEE ALSO

fsck(1m), haltsys(8), shutdown(8), mount(2)

ASSEMBLER

```
(cxenix = 63.)
(shutdn = 0.)
(`shutdn' in r0)
(sblk in r1)
sys cxenix; ~sblk; ..; ..
```

NAME

signal - catch or ignore signals

SYNOPSIS

```
#include <signal.h>

(*signal(sig, func))()
(*func)();
```

DESCRIPTION

A signal is generated by some abnormal event, initiated either by user at a typewriter (quit, interrupt), by a program error (bus error, etc.), or by request of another program (kill). Normally all signals cause termination of the receiving process, but a signal call allows them either to be ignored or to cause an interrupt to a specified location. Here is the list of signals with names as in the include file.

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction (not reset when caught)
SIGTRAP	5*	trace trap (not reset when caught)
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGPIPE	13	write on a pipe or link with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
	16	unassigned

The starred signals in the list above cause a core image if not caught or ignored.

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination, sometimes with a core image. If *func* is SIG_IGN the signal is ignored. Otherwise when the signal occurs *func* will be called with the signal number as argument. A return from the function will continue the process at the point it was interrupted. Except as indicated, a signal is reset to SIG_DFL after being caught. Thus if it is desired to catch every such signal, the catching routine must issue another *signal* call.

When a caught signal occurs during certain system calls, the call terminates prematurely. In particular this can occur during a *read* or *write*(2) on a slow device (like a typewriter; but not a file); and during *pause* or *wait*(2). When such a signal occurs, the saved user status is arranged in such a way that when return from the signal-catching takes place, it will appear that the system call returned an error status. The user's program may then, if it wishes, re-execute the call.

The value of *signal* is the previous (or initial) value of *func* for the particular signal.

After a *fork*(2) the child inherits all signals. *Exec*(2) resets all caught signals to default action.

SEE ALSO

kill(1), kill(2), ptrace(2), setjmp(3)

DIAGNOSTICS

The value (int)-1 is returned if the given signal is out of range.

BUGS

If a repeated signal arrives before the last one can be reset, there is no chance to catch it.

The type specification of the routine and its *func* argument are problematical.

ASSEMBLER

```
(signal = 48.)
sys signal; sig; label
(old label in r0)
```

If *label* is 0, default action is reinstated. If *label* is odd, the signal is ignored. Any other even *label* specifies an address in the process where an interrupt is simulated. An RTI or RTT instruction will return from the interrupt.

NAME

stat, fstat - get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

stat(name, buf)
char *name;
struct stat *buf;

fstat(fildes, buf)
struct stat *buf;
```

DESCRIPTION

Stat obtains detailed information about a named file. *Fstat* obtains the same information about an open file known by the file descriptor from a successful *open*, *creat*, *dup* or *pipe*(2) call.

Name points to a null-terminated string naming a file; *buf* is the address of a buffer into which information is placed concerning the file. It is unnecessary to have any permissions at all with respect to the file, but all directories leading to the file must be searchable. The layout of the structure pointed to by buf as defined in <*stat.h*> is given below. *St_mode* is encoded according to the '#define' statements.

```
struct    stat
{
     dev_t     st_dev;
     ino_t     st_ino;
     unsigned short st_mode;
     short     st_nlink;
     short     st_uid;
     short     st_gid;
     dev_t     st_rdev;
     off_t     st_size;
     time_t    st_atime;
     time_t    st_mtime;
     time_t    st_ctime;
};

#define   S_IFMT    0170000        /* type of file */
#define        S_IFDIR   0040000   /* directory */
#define        S_IFCHR   0020000   /* character special */
#define        S_IFBLK   0060000   /* block special */
#define        S_IFREG   0100000   /* regular */
#define        S_IFMPC   0030000   /* multiplexed char special */
#define        S_IFMPB   0070000   /* multiplexed block special */
#define   S_ISUID   0004000        /* set user id on execution */
```

```
#define   S_ISGID   0002000        /* set group id on execution */
#define   S_ISVTX   0001000        /* save swapped text even after
#define   S_IREAD   0000400        /* read permission, owner */
#define   S_IWRITE  0000200        /* write permission, owner */
#define   S_IEXEC   0000100        /* execute/search permission, ow
```

The mode bits 0000070 and 0000007 encode group and others permissions (see *chmod*(2)). The defined types, *ino t*, *off t*, *time t*, name various width integer values; *dev t* encodes major and minor device numbers; their exact definitions are in the include file <sys/types.h> (see *types*(5).

When *fildes* is associated with a pipe, *fstat* reports an ordinary file with restricted permissions. The size is the number of bytes queued in the pipe.

st atime is the file was last read. For reasons of efficiency, it is not set when a directory is searched, although this would be more logical. *st mtime* is the time the file was last written or created. It is not set by changes of owner, group, link count, or mode. *st ctime* is set both both by writing and changing the i-node.

SEE ALSO

ls(1), filsys(5)

DIAGNOSTICS

Zero is returned if a status is available; -1 if the file cannot be found.

ASSEMBLER

```
(stat = 18.)
sys stat; name; buf

(fstat = 28.)
(file descriptor in r0)
sys fstat; buf
```

NAME

stime - set time

SYNOPSIS

```
stime(tp)
long *tp;
```

DESCRIPTION

Stime sets the system's idea of the time and date. Time, pointed to by _tp_, is measured in seconds from 0000 GMT Jan 1, 1970. Only the super-user may use this call.

SEE ALSO

date(1), time(2), ctime(3)

DIAGNOSTICS

Zero is returned if the time was set; -1 if user is not the super-user.

ASSEMBLER

```
(stime = 25.)
(time in r0-r1)
sys stime
```

NAME

sync - update super-block

SYNOPSIS

sync( )

DESCRIPTION

Sync causes all information in core memory that should be on _disk_ to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system, for example _icheck_, _df_, etc. It is mandatory before a boot.

SEE ALSO

sync(1), update(8)

BUGS

The writing, although scheduled, is not necessarily complete upon return from _sync_.

ASSEMBLER

(sync = 36.)
sys sync

NAME

time, ftime - get date and time

SYNOPSIS

```
long time(0)

long time(tloc)
long *tloc;

#include <sys/types.h>
#include <sys/timeb.h>
ftime(tp)
struct timeb *tp;
```

DESCRIPTION

Time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds.

If _tloc_ is nonnull, the return value is also stored in the place to which _tloc_ points.

The _ftime_ entry fills in a structure pointed to by its argument, as defined by <_sys_/_timeb_._h_>:

```
/*
 * Structure returned by ftime system call
 */
struct timeb {
     time_t   time;
     unsigned short millitm;
     short    timezone;
     short    dstflag;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local timezone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

SEE ALSO

date(1), stime(2), ctime(3)

ASSEMBLER

```
(ftime = 35.)
sys ftime; bufptr

(time = 13.; obsolete call)
sys time
(time since 1970 in r0-r1)
```

NAME

times - get process times

SYNOPSIS

```
times(buffer)
struct tbuffer *buffer;
```

DESCRIPTION

Times returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/HZ seconds, where HZ=60 in North America.

After the call, the buffer will appear as follows:

```
struct tbuffer {
     long proc_user_time;
     long proc_system_time;
     long child_user_time;
     long child_system_time;
};
```

The children times are the sum of the children's process times and their children's times.

SEE ALSO

time(1), time(2)

ASSEMBLER

```
(times = 43.)
sys times; buffer
```

NAME

umask - set file creation mode mask

SYNOPSIS

```
umask(complmode)
```

DESCRIPTION

Umask sets a mask used whenever a file is created by *creat*(2) or *mknod*(2): the actual mode (see *chmod*(2)) of the newly-created file is the logical **and** of the given mode and the complement of the argument. Only the low-order 9 bits of the mask (the protection bits) participate. In other words, the mask shows the bits to be turned off when files are created.

The previous value of the mask is returned by the call. The value is initially 0 (no restrictions). The mask is inherited by child processes.

SEE ALSO

creat(2), mknod(2), chmod(2)

ASSEMBLER

```
(umask = 60.)
sys umask; complmode
```

NAME

unlink - remove directory entry

SYNOPSIS

```
unlink(name)
char *name;
```

DESCRIPTION

Name points to a null-terminated string. _Unlink_ removes the entry for the file pointed to by _name_ from its directory. If this entry was the last link to the file, the contents of the file are freed and the file is destroyed. If, however, the file was open in any process, the actual destruction is delayed until it is closed, even though the directory entry has disappeared.

SEE ALSO

rm(1), link(2)

DIAGNOSTICS

Zero is normally returned; -1 indicates that the file does not exist, that its directory cannot be written, or that the file contains pure procedure text that is currently in use. Write permission is not required on the file itself. It is also illegal to unlink a directory (except for the super-user).

ASSEMBLER

```
(unlink = 10.)
sys unlink; name
```

NAME

utime - set file times

SYNOPSIS

```
#include <sys/types.h>
utime(file, timep)
char *file;
time_t timep[2];
```

DESCRIPTION

The _utime_ call uses the `accessed' and `updated' times in that order from the _timep_ vector to set the corresponding recorded times for _file_.

The caller must be the owner of the file or the super-user. The `inode-changed' time of the file is set to the current time.

SEE ALSO

stat (2)

ASSEMBLER

```
(utime = 30.)
sys utime; file; timep
```

NAME

wait - wait for process to terminate

SYNOPSIS

wait(status)
int *status;

wait(0)

DESCRIPTION

Wait causes its caller to delay until a signal is received or one of its child processes terminates. If any child has died since the last *wait*, return is immediate; if there are no children, return is immediate with the error bit set (resp. with a value of -1 returned). The normal return yields the process ID of the terminated child. In the case of several children several *wait* calls are needed to learn of all the deaths.

If (int)*status* is nonzero, the high byte of the word pointed to receives the low byte of the argument of *exit* when the child terminated. The low byte receives the termination status of the process. See *signal*(2) for a list of termination statuses (signals); 0 status indicates normal termination. A special status (0177) is returned for a stopped process which has not terminated and can be restarted. See *ptrace*(2). If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

If the parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

SEE ALSO

exit(2), fork(2), signal(2)

DIAGNOSTICS

Returns -1 if there are no children not previously waited for.

ASSEMBLER

(wait = 7.)
sys wait
(process ID in r0)
(status in r1)

The high byte of the status is the low byte of r0 in the child at termination.

NAME

write - write on a file

SYNOPSIS

```
write(fildes, buffer, nbytes)
char *buffer;
```

DESCRIPTION

A file descriptor is a word returned from a successful _open_, _creat_, _dup_, or _pipe_(2) call.

Buffer is the address of _nbytes_ contiguous bytes which are written on the output file. The number of characters actually written is returned. It should be regarded as an error if this is not the same as requested.

Writes which are multiples of 512 characters long and begin on a 512-byte boundary in the file are more efficient than any others.

SEE ALSO

creat(2), open(2), pipe(2)

DIAGNOSTICS

Returns -1 on error: bad descriptor, buffer address, or count; physical I/O errors.

ASSEMBLER

```
(write = 4.)
(file descriptor in r0)
sys write; buffer; nbytes
(byte count in r0)
```

NAME

intro - introduction to library functions

SYNOPSIS

#include <stdio.h>

#include <math.h>

DESCRIPTION

This section describes functions that may be found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in section 2. Functions are divided into various libraries distinguished by the section number at the top of the page:

(3) These functions, together with those of section 2 and those marked (3S), constitute library _libc_, which is automatically loaded by the C compiler _cc_(1) and the Fortran compiler _f77_(1). The link editor _ld_(1) searches this library under the `-lc' option. Declarations for some of these functions may be obtained from include files indicated on the appropriate pages.

(3M) These functions constitute the math library, _libm_. They are automatically loaded as needed by the Fortran compiler _f77_(1). The link editor searches this library under the `-lm' option. Declarations for these functions may be obtained from the include file <math.h>.

(3S) These functions constitute the `standard I/O package', see _stdio_(3). These functions are in the library _libc_ already mentioned. Declarations for these functions may be obtained from the include file <stdio.h>.

(3X) Various specialized libraries have not been given distinctive captions. The files in which these libraries are found are named on the appropriate pages.

FILES

/lib/libc.a
/lib/libm.a, /usr/lib/libm.a (one or the other)

SEE ALSO

stdio(3), nm(1), ld(1), cc(1), f77(1), intro(2)

DIAGNOSTICS

Functions in the math library (3M) may return conventional values when the function is undefined for the given arguments or when the value is not representable. In these cases the external variable _errno_ (see _intro_(2)) is set to the value EDOM or ERANGE. The values of EDOM and ERANGE are defined in the include file <math.h>.

ASSEMBLER

In assembly language these functions may be accessed by simulating the C calling sequence. For example, ecvt(3) might be called this way:

```
setd
mov  $sign,-(sp)
mov  $decpt,-(sp)
mov  ndigit,-(sp)
movf value,-(sp)
jsr  pc,_ecvt
add  $14.,sp
```

NAME

abort - generate IOT fault

SYNOPSIS

abort()

DESCRIPTION

Abort executes the PDP11 IOT instruction. This causes a signal that normally terminates the process with a core dump, which may be used for debugging.

SEE ALSO

adb(1), signal(2), exit(2)

DIAGNOSTICS

Usually `IOT trap - core dumped' from the shell.

NAME
    abs - integer absolute value

SYNOPSIS
    abs(i)

DESCRIPTION
    Abs returns the absolute value of its integer operand.

SEE ALSO
    floor(3) for _fabs_

BUGS
    You get what the hardware gives on the largest negative integer.

NAME

atof, atoi, atol - convert ASCII to numbers

SYNOPSIS

```
double atof(nptr)
char *nptr;

atoi(nptr)
char *nptr;

long atol(nptr)
char *nptr;
```

DESCRIPTION

These functions convert a string pointed to by _nptr_ to floating, integer, and long integer representation respectively. The first unrecognized character ends the string.

Atof recognizes an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional `e' or `E' followed by an optionally signed integer.

Atoi and _atol_ recognize an optional string of tabs and spaces, then an optional sign, then a string of digits.

SEE ALSO

scanf(3)

BUGS

There are no provisions for overflow.

NAME

crypt, setkey, encrypt - DES encryption

SYNOPSIS

```
char *crypt(key, salt)
char *key, *salt;

setkey(key)
char *key;

encrypt(block, edflag)
char *block;
```

DESCRIPTION

Crypt is the password encryption routine. It is based on the NBS Data Encryption Standard, with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

The first argument to *crypt* is a user's typed password. The second is a 2-character string chosen from the set [a-zA-Z0-9./]. The *salt* string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password, in the same alphabet as the salt. The first two characters are the salt itself.

The other entries provide (rather primitive) access to the actual DES algorithm. The argument of *setkey* is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored, leading to a 56-bit key which is set into the machine.

The argument to the *encrypt* entry is likewise a character array of length 64 containing 0's and 1's. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by *setkey*. If *edflag* is 0, the argument is encrypted; if non-zero, it is decrypted.

SEE ALSO

passwd(1), passwd(5), login(1), getpass(3)

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

ctime, localtime, gmtime, asctime, timezone - convert date and time to ASCII

SYNOPSIS

```
char *ctime(clock)
long *clock;

#include <time.h>

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;

char *timezone(zone, dst)
```

DESCRIPTION

Ctime converts a time pointed to by *clock* such as returned by *time*(2) into ASCII and returns a pointer to a 26-character string in the following form. All the fields have constant width.

```
Sun Sep 16 01:03:52 1973\n\0
```

Localtime and *gmtime* return pointers to structures containing the broken-down time. *Localtime* corrects for the time zone and possible daylight savings time; *gmtime* converts directly to GMT, which is the time UNIX uses. *Asctime* converts a broken-down time to ASCII and returns a pointer to a 26-character string.

The structure declaration from the include file is:

```
struct tm { /* see ctime(3) */
     int  tm_sec;
     int  tm_min;
     int  tm_hour;
     int  tm_mday;
     int  tm_mon;
     int  tm_year;
     int  tm_wday;
     int  tm_yday;
     int  tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday =

0), year - 1900, day of year (0-365), and a flag that is nonzero if daylight saving time is in effect.

When local time is called for, the program consults the system to determine the time zone and whether the standard U.S.A. daylight saving time adjustment is appropriate. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

Timezone returns the name of the time zone associated with its first argument, which is measured in minutes westward from Greenwich. If the second argument is 0, the standard name is used, otherwise the Daylight Saving version. If the required name does not appear in a table built into the routine, the difference from GMT is produced; e.g. in Afghanistan *timezone*(-(*60***4*+*30*), *0*) is appropriate because it is 4:30 ahead of GMT and the string GMT+4:30 is produced.

SEE ALSO

time(2)

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii - character classification
toupper, tolower, toascii - character transformation

SYNOPSIS

#include <ctype.h>

isalpha(c)

. . .

DESCRIPTION

These macros classify ASCII-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. *Isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value EOF (see *stdio*(3)).

isalpha	*c* is a letter
isupper	*c* is an upper case letter
islower	*c* is a lower case letter
isdigit	*c* is a digit
isxdigit	*c* is a hexadecimal digit
isalnum	*c* is an alphanumeric character
isspace	*c* is a space, tab, carriage return, newline, or formfeed
ispunct	*c* is a punctuation character (neither control nor alphanumeric)
isprint	*c* is a printing character, code 040(8) (space) through 0176 (tilde)
iscntrl	*c* is a delete character (0177) or ordinary control character (less than 040).
isascii	*c* is an ASCII character, code less than 0200

These macros transform ASCII-coded integer values and non-ascii characters in a repeatable way.

toupper
c transforms lower case letters to upper case, but is undefined for other values.

<u>tolower</u>
    <u>c</u> transforms upper case letters to lower case, but is undefined for other values.

<u>toascii</u>
    <u>c</u> transforms a non-ascii character to the corresponding ascii character, without disturbing ascii characters. Since EOF is not a character, mapping to ascii has undesirable properties.

SEE ALSO
    ascii(7), stdio(3), getc(3), putc(3)

NAME

curses - screen functions with ``optimal'' cursor motion

SYNOPSIS

cc [flags] files -*lcurses* -*ltermlib* [libraries]

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the *refresh*() tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine *initscr*() must be called before any of the other routines that deal with windows and screens are used.

SEE ALSO

Screen Updating and Cursor Movement Optimization: A Library Package, Ken Arnold,
termcap (5), stty (2), setenv (3), setenv (3),

AUTHOR

Ken Arnold

FUNCTIONS

addch(ch)	add a character to *stdscr*
addstr(str)	add a string to *stdscr*
box(win,vert,hor)	draw a box around a window
crmode()	set cbreak mode
clear()	clear *stdscr*
clearok(scr,boolf)	set clear flag for *scr*
clrtobot()	clear to bottom on *stdscr*
clrtoeol()	clear to end of line on *stdscr*
delwin(win)	delete *win*
echo()	set echo mode
erase()	erase *stdscr*
getch()	get a char through *stdscr*
getstr(str)	get a string through *stdscr*
gettmode()	get tty modes
getyx(win,y,x)	get (y,x) co-ordinates
inch()	get char at current (y,x) co-ordinate
initscr()	initialize screens
leaveok(win,boolf)	set leave flag for *win*
longname(termbuf,name)	get long name from *termbuf*
move(y,x)	move to (y,x) on *stdscr*
mvcur(lasty,lastx,newy,newx)	actually move cursor
newwin(lines,cols,begin_y,begin_x)	create a new window
nl()	set newline mapping
nocrmode()	unset cbreak mode
noecho()	unset echo mode
nonl()	unset newline mapping
noraw()	unset raw mode

overlay(win1,win2)	overlay win1 on win2
overwrite(win1,win2)	overwrite win1 on top of win2
printw(fmt,arg1,arg2,...)	printf on *stdscr*
raw()	set raw mode
refresh()	make current screen look like *stdscr*
restty()	reset tty flags to stored value
savetty()	stored current tty flags
scanw(fmt,arg1,arg2,...)	scanf through *stdscr*
scroll(win)	scroll *win* one line
scrollok(win,boolf)	set scroll flag
setterm(name)	set term variables for name
unctrl(ch)	printable version of *ch*
waddch(win,ch)	add char to *win*
waddstr(win,str)	add string to *win*
wclear(win)	clear *win*
wclrtobot(win)	clear to bottom of *win*
wclrtoeol(win)	clear to end of line on *win*
werase(win)	erase *win*
wgetch(win)	get a char through *win*
wgetstr(win,str)	get a string through *win*
winch(win)	get char at current (y,x) in *win*
wmove(win,y,x)	set current (y,x) co-ordinates on *win*
wprintw(win,fmt,arg1,arg2,...)	printf on *win*
wrefresh(win)	make screen look like *win*
wscanw(win,fmt,arg1,arg2,...)	scanf through *win*

NAME

ecvt, fcvt, gcvt - output conversion

SYNOPSIS

```
char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt(value, ndigit, buf)
double value;
char *buf;
```

DESCRIPTION

Ecvt converts the _value_ to a null-terminated string of _ndigit_ ASCII digits and returns a pointer thereto. The position of the decimal point relative to the beginning of the string is stored indirectly through _decpt_ (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by _sign_ is non-zero, otherwise it is zero. The low-order digit is rounded.

Fcvt is identical to _ecvt_, except that the correct digit has been rounded for Fortran F-format output of the number of digits specified by _ndigits_.

Gcvt converts the _value_ to a null-terminated ASCII string in _buf_ and returns a pointer to _buf_. It attempts to produce _ndigit_ significant digits in Fortran F format if possible, otherwise E format, ready for printing. Trailing zeros may be suppressed.

SEE ALSO

printf(3)

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

end, etext, edata - last locations in program

SYNOPSIS

extern end;
extern etext;
extern edata;

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of *etext* is the first address above the program text, *edata* above the initialized data region, and *end* above the uninitialized data region.

When execution begins, the program break coincides with *end*, but many functions reset the program break, among them the routines of *brk*(2), *malloc*(3), standard input/output (*stdio*(3)), the profile (-p) option of *cc*(1), etc. The current value of the program break is reliably returned by `sbrk(0)', see *brk*(2).

SEE ALSO

brk(2), malloc(3)

NAME

frexp, ldexp, modf - split into mantissa and exponent

SYNOPSIS

```
double frexp(value, eptr)
double value;
int *eptr;

double ldexp(value, exp)
double value;

double modf(value, iptr)
double value, *iptr;
```

DESCRIPTION

Frexp returns the mantissa of a double _value_ as a double quantity, _x_, of magnitude less than 1 and stores an integer _n_ such that _value_ = _x_*2**_n_ indirectly through _eptr_.

Ldexp returns the quantity _value_*2**_exp_.

Modf returns the positive fractional part of _value_ and stores the integer part indirectly through _iptr_.

NAME

getenv - value for environment name

SYNOPSIS

```
char *getenv(name)
char *name;
```

DESCRIPTION

Getenv searches the environment list (see *environ*(5)) for a string of the form *name*=*value* and returns *value* if such a string is present, otherwise 0 (NULL).

SEE ALSO

environ(5), exec(2)

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent - get group file entry

SYNOPSIS

```
#include <grp.h>

struct group *getgrent();

struct group *getgrgid(gid) int gid;

struct group *getgrnam(name) char *name;

int setgrent();

int endgrent();
```

DESCRIPTION

Getgrent, _getgrgid_ and _getgrnam_ each return pointers to an object with the following structure containing the broken-out fields of a line in the group file.

```
struct    group { /* see getgrent(3) */
          char *gr_name;
          char *gr_passwd;
          int  gr_gid;
          char **gr_mem;
};
```

The members of this structure are:

gr_name
    The name of the group.

gr_passwd
    The encrypted password of the group.

gr_gid
    The numerical group-ID.

gr_mem
    Null-terminated vector of pointers to the individual member names.

Getgrent simply reads the next line while _getgrgid_ and _getgrnam_ search until a matching _gid_ or _name_ is found (or until EOF is encountered). Each routine picks up where the others leave off so successive calls may be used to search the entire file.

A call to _setgrent_ has the effect of rewinding the group file to allow repeated searches. _Endgrent_ may be called to close the group file when processing is complete.

FILES

/etc/group

SEE ALSO

getlogin(3), getpwent(3), group(5)

DIAGNOSTICS

A null pointer (0) is returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

getlogin - get login name

SYNOPSIS

```
char *getlogin();
```

DESCRIPTION

Getlogin returns a pointer to the login name as found in */etc/utmp*. It may be used in conjunction with *getpwnam* to locate the correct password file entry when the same userid is shared by several login names.

If *getlogin* is called within a process that is not attached to a typewriter, it returns NULL. The correct procedure for determining the login name is to first call *getlogin* and if it fails, to call *getpwuid*.

FILES

/etc/utmp

SEE ALSO

getpwent(3), getgrent(3), utmp(5)

DIAGNOSTICS

Returns NULL (0) if name not found.

BUGS

The return values point to static data whose content is overwritten by each call.

NAME

getpass - read a password

SYNOPSIS

```
char *getpass(prompt)
char *prompt;
```

DESCRIPTION

Getpass reads a password from the file */dev/tty*, or if that cannot be opened, from the standard input, after prompting with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters.

FILES

/dev/tty

SEE ALSO

crypt(3)

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

getpw - get name from UID

SYNOPSIS

```
getpw(uid, buf)
char *buf;
```

DESCRIPTION

Getpw searches the password file for the (numerical) *uid*, and fills in *buf* with the corresponding line; it returns non-zero if *uid* could not be found. The line is null-terminated.

FILES

/etc/passwd

SEE ALSO

getpwent(3), passwd(5)

DIAGNOSTICS

Non-zero return on error.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent - get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent();

struct passwd *getpwuid(uid) int uid;

struct passwd *getpwnam(name) char *name;

int setpwent();

int endpwent();
```

DESCRIPTION

Getpwent, *getpwuid* and *getpwnam* each return a pointer to an object with the following structure containing the broken-out fields of a line in the password file.

```
struct    passwd { /* see getpwent(3) */
          char *pw_name;
          char *pw_passwd;
          int  pw_uid;
          int  pw_gid;
          int  pw_quota;
          char *pw_comment;
          char *pw_gecos;
          char *pw_dir;
          char *pw_shell;
};
```

The fields *pw quota* and *pw comment* are unused; the others have meanings described in *passwd*(5).

Getpwent reads the next line (opening the file if necessary); *setpwent* rewinds the file; *endpwent* closes it.

Getpwuid and *getpwnam* search from the beginning until a matching *uid* or *name* is found (or until EOF is encountered).

FILES

/etc/passwd

SEE ALSO

getlogin(3), getgrent(3), passwd(5)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

l3tol, ltol3 - convert between 3-byte integers and long integers

SYNOPSIS

```
l3tol(lp, cp, n)
long *lp;
char *cp;

ltol3(cp, lp, n)
char *cp;
long *lp;
```

DESCRIPTION

L3tol converts a list of _n_ three-byte integers packed into a character string pointed to by _cp_ into a list of long integers pointed to by _lp_.

Ltol3 performs the reverse conversion from long integers (_lp_) to three-byte integers (_cp_).

These functions are useful for file-system maintenance; disk addresses are three bytes long.

SEE ALSO

filsys(5)

NAME

malloc, free, realloc, calloc - main memory allocator

SYNOPSIS

```
char *malloc(size)
unsigned size;

free(ptr)
char *ptr;

char *realloc(ptr, size)
char *ptr;
unsigned size;

char *calloc(nelem, elsize)
unsigned nelem, elsize;
```

DESCRIPTION

Malloc and *free* provide a simple general-purpose memory allocation package. *Malloc* returns a pointer to a block of at least *size* bytes beginning on a word boundary.

The argument to *free* is a pointer to a block previously allocated by *malloc*; this space is made available for further allocation, but its contents are left undisturbed.

Needless to say, grave disorder will result if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

Malloc allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls *sbrk* (see *break*(2)) to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

Realloc also works if *ptr* points to a block freed since the last call of *malloc*, *realloc* or *calloc*; thus sequences of *free*, *malloc* and *realloc* can exploit the search strategy of *malloc* to do storage compaction.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

DIAGNOSTICS

Malloc, *realloc* and *calloc* return a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. *Malloc* may be recompiled to check the arena very stringently on every transaction; see the source code.

BUGS

When *realloc* returns 0, the block pointed to by *ptr* may be destroyed.

NAME

mktemp - make a unique file name

SYNOPSIS

```
char *mktemp(template)
char *template;
```

DESCRIPTION

Mktemp replaces template by a unique file name, and returns the address of the template. The template should look like a file name with six trailing X's, which will be replaced with the current process id and a unique letter.

SEE ALSO

getpid(2)

NAME

monitor - prepare execution profile

SYNOPSIS

```
monitor(lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)( ), (*highpc)( );
short buffer[ ];
```

DESCRIPTION

An executable program created by `cc -p' automatically includes calls for _monitor_ with default parameters; _monitor_ needn't be called explicitly except to gain fine control over profiling.

Monitor is an interface to _profil_(2). _Lowpc_ and _highpc_ are the addresses of two functions; _buffer_ is the address of a (user supplied) array of _bufsize_ short integers. _Monitor_ arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of _lowpc_ and the highest is just below _highpc_. At most _nfunc_ call counts can be kept; only calls of functions compiled with the profiling option -p of _cc_(1) are recorded. For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext();
...
monitor((int)2, etext, buf, bufsize, nfunc);
```

Etext lies just above all the program text, see _end_(3).

To stop execution monitoring and write the results on the file _mon.out_, use

```
monitor(0);
```

then _prof_(1) can be used to examine the results.

FILES

mon.out

SEE ALSO

prof(1), profil(2), cc(1)

NOTES

The _prof_(1) program may require the buffer size to be equal to or smaller than the program size. If you did not use the profiling option **-p** of _cc_(1) you will want to use 0 for the nfunc argument.

NAME

nlist - get entries from name list

SYNOPSIS

```
#include <a.out.h>
nlist(filename, nl)
char *filename;
struct nlist nl[ ];
```

DESCRIPTION

Nlist examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See *a.out*(5) for the structure declaration.

This subroutine is useful for examining the system name list kept in the file /unix. In this way programs can obtain system addresses that are up to date.

SEE ALSO

a.out(5)

DIAGNOSTICS

All type entries are set to 0 if the file cannot be found or if it is not a valid namelist.

NAME

perror, sys_errlist, sys_nerr - system error messages

SYNOPSIS

```
perror(s)
char *s;

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

DESCRIPTION

Perror produces a short error message on the standard error file describing the last error encountered during a call to the system from a C program. First the argument string *s* is printed, then a colon, then the message and a new-line. Most usefully, the argument string is the name of the program which incurred the error. The error number is taken from the external variable *errno* (see *intro*(2)), which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings *sys errlist* is provided; *errno* can be used as an index in this table to get the message string without the newline. *Sys nerr* is the number of messages provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

SEE ALSO

intro(2)

NAME

pkopen, pkclose, pkread, pkwrite, pkfail - packet driver simulator

SYNOPSIS

```
char *pkopen(fd)

pkclose(ptr)
char *ptr;

pkread(ptr, buffer, count)
char *ptr, *buffer;

pkwrite(ptr, buffer, count)
char *ptr, *buffer;

pkfail()
```

DESCRIPTION

These routines are a user-level implementation of the full-duplex end-to-end communication protocol described in _pk_(4). If _fd_ is a file descriptor open for reading and writing, _pkopen_ carries out the initial synchronization and returns an identifying pointer. The pointer is used as the first parameter to _pkread_, _pkwrite_, and _pkclose_.

Pkread, _pkwrite_ and _pkclose_ behave analogously to _read_, _write_ and _close_(2). However, a write of zero bytes is meaningful and will produce a corresponding read of zero bytes.

SEE ALSO

pk(4), pkon(2)

DIAGNOSTICS

Pkfail is called upon persistent breakdown of communication. _Pkfail_ must be supplied by the user.

Pkopen returns a null (0) pointer if packet protocol can not be established.

Pkread returns -1 on end of file, 0 in correspondence with a 0-length write.

BUGS

This simulation of _pk_(4) leaves something to be desired in needing special read and write routines, and in not being inheritable across calls of _exec_(2). Its prime use is on systems that lack _pk_.
These functions use _alarm_(2); simultaneous use of _alarm_ for other puposes may cause trouble.

NAME

qsort - quicker sort

SYNOPSIS

```
qsort(base, nel, width, compar)
char *base;
int (*compar)( );
```

DESCRIPTION

Qsort is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine to be called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

SEE ALSO

sort(1)

NAME

rand, srand - random number generator

SYNOPSIS

```
srand(seed)
int seed;

rand( )
```

DESCRIPTION

Rand uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to 2^{15}-1.

The generator is reinitialized by calling _srand_ with 1 as argument. It can be set to a random starting point by calling _srand_ with whatever you like as argument.

NAME

setjmp, longjmp - non-local goto

SYNOPSIS

```
#include <setjmp.h>

setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;
```

DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* for later use by *longjmp*. It returns value 0.

Longjmp restores the environment saved by the last call of *setjmp*. It then returns in such a way that execution continues as if the call of *setjmp* had just returned the value *val* to the function that invoked *setjmp*, which must not itself have returned in the interim. All accessible data have values as of the time *longjmp* was called except for register variables whose values are undefined.

SEE ALSO

signal(2)

NAME

sleep - suspend execution for interval

SYNOPSIS

```
sleep(seconds)
unsigned seconds;
```

DESCRIPTION

The current process is suspended from execution for the number of seconds specified by the argument. The actual suspension time may be up to 1 second less than that requested, because scheduled wakeups occur at fixed 1-second intervals, and an arbitrary amount longer because of other activity in the system.

The routine is implemented by setting an alarm clock signal and pausing until it occurs. The previous state of this signal is saved and restored. If the sleep time exceeds the time to the alarm signal, the process sleeps only until the signal would have occurred, and the signal is sent 1 second later.

SEE ALSO

alarm(2), pause(2)

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, index, rindex - string operations

SYNOPSIS

```
char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s1, s2, n)
char *s1, *s2;

strcmp(s1, s2)
char *s1, *s2;

strncmp(s1, s2, n)
char *s1, *s2;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;

strlen(s)
char *s;

char *index(s, c)
char *s, c;

char *rindex(s, c)
char *s;
```

DESCRIPTION

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

Strcat appends a copy of string _s2_ to the end of string _s1_. _Strncat_ copies at most _n_ characters. Both return a pointer to the null-terminated result.

Strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, according as _s1_ is lexicographically greater than, equal to, or less than _s2_. _Strncmp_ makes the same comparison but looks at at most _n_ characters.

Strcpy copies string _s2_ to _s1_, stopping after the null character has been moved. _Strncpy_ copies exactly _n_ characters, truncating or null-padding _s2_; the target may not be null-terminated if the length of _s2_ is _n_ or more. Both return _s1_.

Strlen returns the number of non-null characters in _s_.

Index (_rindex_) returns a pointer to the first (last) occurrence of character _c_ in string _s_, or zero if _c_ does not occur in the string.

BUGS

Strcmp uses native character comparison, which is signed on _PDP11_'s, unsigned on other machines.

NAME

swab - swap bytes

SYNOPSIS

```
swab(from, to, nbytes)
char *from, *to;
```

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between PDP11's and other machines. *Nbytes* should be even.

NAME

system - issue a shell command

SYNOPSIS

```
system(string)
char *string;
```

DESCRIPTION

System causes the *string* to be given to *sh*(1) as input as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

SEE ALSO

popen(3), exec(2), wait(2)

DIAGNOSTICS

Exit status 127 indicates the shell couldn't be executed.

NAME

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs - terminal independent operation routines

SYNOPSIS

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *
tgetstr(id, area)
char *id, **area;

char *
tgoto(cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

DESCRIPTION

These functions extract and use capabilities from the terminal capability data base *termcap*(5). These are low level routines; see *curses*(3) for a higher level package.

Tgetent extracts the entry for terminal *name* into the buffer at *bp*. *Bp* should be a character buffer of size 1024 and must be retained through all subsequent calls to *tgetnum*, *tgetflag*, and *tgetstr*. *Tgetent* returns -1 if it cannot open the *termcap* file, 0 if the terminal name given does not have an entry, and 1 if all goes well. It will look in the environment for a TERMCAP variable. If found, and the value does not begin with a slash, and the terminal type **name** is the same as the environment string TERM, the TERMCAP string is used instead of reading the termcap file. If it does begin with a slash, the string is used as a path name rather than /*etc*/*termcap*. This can speed up entry into programs that call *tgetent*, as well as to help debug new terminal descriptions or to make one for your terminal if you can't write the file /*etc*/*termcap*.

Tgetnum gets the numeric value of capability *id*, returning -1 if is not given for the terminal. *Tgetflag* returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. *Tgetstr* gets the string value of capability *id*, placing it in the buffer at *area*, advancing the *area* pointer. It decodes the abbreviations for this field described in *termcap*(5), except for cursor addressing and padding information.

Tgoto returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables **UP** (from the **up** capability) and **BC** (if **bc** is given rather than **bs**) if necessary to avoid placing **\n**, **^D** or **^@** in the returned string. (Programs which call tgoto should be sure to turn off the XTABS bit(s), since tgoto may now output a tab. Note that programs using termcap should in general turn off XTABS anyway since some terminals use control I for other functions, such as nondestructive space.) If a % sequence is given which is not understood, then *tgoto* returns OOPS.

Tputs decodes the leading padding information of the string *cp*; *affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable, *outc* is a routine which is called with each character in turn. The external variable *ospeed* should contain the output speed of the terminal as encoded by *stty* (*2*). The external variable **PC** should contain a pad character to be used (from the **pc** capability) if a null (^@) is inappropriate.

FILES

/usr/lib/libtermcap.a	-ltermcap library
/etc/termcap	data base

SEE ALSO

ex(1), curses(3), termcap(5)

AUTHOR

William Joy

BUGS

NAME

ttyname, isatty, ttyslot - find name of a terminal

SYNOPSIS

char *ttyname(fildes)

isatty(fildes)

ttyslot()

DESCRIPTION

Ttyname returns a pointer to the null-terminated path name of the terminal device associated with file descriptor *fildes*.

Isatty returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

Ttyslot returns the number of the entry in the *ttys*(5) file for the control terminal of the current process.

FILES

/dev/*
/etc/ttys

SEE ALSO

ioctl(2), ttys(5)

DIAGNOSTICS

Ttyname returns a null pointer (0) if *fildes* does not describe a terminal device in directory \`/dev'.

Ttyslot returns 0 if \`/etc/ttys' is inaccessible or if it cannot determine the control terminal.

BUGS

The return value points to static data whose content is overwritten by each call.

NAME

mem, kmem - core memory

DESCRIPTION

Mem is a special file that is an image of the core memory of the computer. It may be used, for example, to examine, and even to patch the system. *Kmem* is the same as *mem* except that kernel virtual memory rather than physical memory is accessed.

Byte addresses are interpreted as memory addresses. References to non-existent locations return errors.

On the Z8000, the per-process data for the current process begins at 0xF800.

Inout and *inoutb* may be used to access the I/O space on the Z8000. Reads from these devices return values read from the device register addressed. Writes to these devices send data to the device register addressed. The file offset is not updated for these devices, so subsequent reads or writes refer to the same device register.

FILES

/dev/mem, /dev/kmem, /dev/inout, /dev/inoutb

BUGS

NAME

null - data sink

DESCRIPTION

Data written on a null special file is discarded.

Reads from a null special file always return 0 bytes.

FILES

/dev/null

NAME

pk - packet driver

DESCRIPTION

The packet driver implements a full-duplex end-to-end flow control strategy for machine-to-machine communication. Packet driver protocol is established by calling *pkon*(2) with a character device file descriptor and a desired packet size in bytes. The packet size must be a power of 2, $32 \leq size \leq 4096$. The file descriptor must represent an 8-bit data path. This is normally obtained by setting the device in raw mode (see *ioctl*(2)).

The actual packet size, which may be smaller than the desired packet size, is arrived at by negotiation with the packet driver at the remote end of the data link.

The packet driver maintains two data areas for incoming and outgoing packets. The output area is needed to implement retransmission on errors, and arriving packets are queued in the input area. Data arriving for a file not open for reading is discarded. Initially the size of both areas is set to two packets.

It is not necessary that reads and writes be multiples of the packet size although there is less system overhead if they are. Read operations return the maximum amount of data available from the input area up to the number of bytes specified in the system call. The buffer sizes in write operations are not normally transmitted across the link. However, writes of zero length are treated specially and are reflected at the remote end as a zero-length read. This facilitates marking the serial byte stream, usually for delimiting files.

When one side of a packet driver link is shut down by *close*(2)*or* *pkoff* (see *pkon*(2)), *read*(2) on the other side will return 0, and *write* on the other side will raise a SIGPIPE signal.

SEE ALSO

pkon(2), pkopen(3)

NAME

tty - general terminal interface

DESCRIPTION

This section describes both a particular special file, and the general nature of the terminal interface.

The file /*dev*/*tty* is, in each process, a synonym for the control terminal associated with that process. It is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected. It can also be used for programs that demand a file name for output, when typed output is desired and it is tiresome to find out which terminal is currently in use.

As for terminals in general: all of the low-speed asynchronous communications ports use the same general interface, no matter what hardware is involved. The remainder of this section discusses the common features of the interface.

When a terminal file is opened, it causes the process to wait until a connection is established. In practice user's programs seldom open these files; they are opened by *init* and become a user's input and output file. The very first terminal file open in a process becomes the *control terminal* for that process. The control terminal plays a special role in handling quit or interrupt signals, as discussed below. The control terminal is inherited by a child process during a *fork*, even if the control terminal is closed. The set of processes that thus share a control terminal is called a *process group*; all members of a process group receive certain signals together, see DEL below and *kill*(2).

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters may be typed at any time, even while output is occurring, and are only lost when the system's character input buffers become completely choked, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently this limit is 256 characters. When the input limit is reached all the saved characters are thrown away without notice.

Normally, terminal input is processed in units of lines. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not however necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information. There are special modes, discussed below, that permit the program to read each character as typed without waiting for a full line.

During input, erase and kill processing is normally done. By default, the character `#' erases the last character typed, except that it will not erase beyond the beginning of a line or an EOT. By default, the character `@' kills the entire line up to the point where it was typed, but not beyond an EOT. Both these characters operate on a keystroke basis independently of any backspacing or tabbing that may have been done. Either `@' or `#' may be entered literally by preceding it by `\'; the erase or kill character remains, but the `\' disappears. These two characters may be changed to others.

When desired, all upper-case letters are mapped into the corresponding lower-case letter. The upper-case letter may be generated by preceding it by `\'. In addition, the following escape sequences can be generated on output and accepted on input:

for	use
`	\'
|	\!
~	\^
{	\(
}	\)

Certain ASCII control characters have special meaning. These characters are not passed to a reading program except in raw mode where they lose their special character. Also, it is possible to change these characters from the default; see below.

EOT (Control-D) may be used to generate an end of file from a terminal. When an EOT is received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOT is discarded. Thus if there are no characters waiting, which is to say the EOT occurred at the beginning of a line, zero characters will be passed back, and this is the standard end-of-file indication.

DEL (Rubout) is not passed to a program but generates an *interrupt* signal which is sent to all processes with the associated control terminal. Normally each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location. See *signal*(2).

FS (Control-\ or control-shift-L) generates the *quit* signal. Its treatment is identical to the interrupt signal except that unless a receiving process has made other arrangements it will not only be terminated but a core image file will be generated.

DC3 (Control-S) delays all printing on the terminal until something is typed in.

DC1 (Control-Q) restarts printing after DC3 without generating any input to a program.

When the carrier signal from the dataset drops (usually because the user has hung up his terminal) a _hangup_ signal is sent to all processes with the terminal as control terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hangup signal is ignored, any read returns with an end-of-file indication. Thus programs that read a terminal and test for end-of-file on their input can terminate appropriately when hung up on.

When one or more characters are written, they are actually transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. When a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold the program is resumed. Even parity is always generated on output. The EOT character is not transmitted (except in raw mode) to prevent terminals that respond to it from hanging up.

Several _ioctl_(2) calls apply to terminals. Most of them use the following structure, defined in <_sgtty.h_>:

```
struct sgttyb {
     char sg_ispeed;
     char sg_ospeed;
     char sg_erase;
     char sg_kill;
     int  sg_flags;
};
```

The _sg ispeed_ and _sg ospeed_ fields describe the input and output speeds of the device according to the following table, which corresponds to the DEC DH-11 interface. If other hardware is used, impossible speed changes are ignored. Symbolic values in the table are as defined in <_sgtty.h_>.

B0	0	(hang up dataphone)
B50	1	50 baud
B75	2	75 baud
B110	3	110 baud

B134	4	134.5 baud
B150	5	150 baud
B200	6	200 baud
B300	7	300 baud
B600	8	600 baud
B1200	9	1200 baud
B1800	10	1800 baud
B2400	11	2400 baud
B4800	12	4800 baud
B9600	13	9600 baud
EXTA	14	External A
EXTB	15	External B

In the current configuration, only 110, 150, 300 and 1200 baud are really supported on dial-up lines. Code conversion and line control required for IBM 2741's (134.5 baud) must be implemented by the user's program. The half-duplex line discipline required for the 202 dataset (1200 baud) is not supplied; full-duplex 212 datasets work fine.

The _sg erase_ and _sg kill_ fields of the argument structure specify the erase and kill characters respectively. (Defaults are # and @.)

The _sg flags_ field of the argument structure contains several bits that determine the system's treatment of the terminal:

ALLDELAY	0177400	Delay algorithm selection
BSDELAY	0100000	Select backspace delays (not implemented):
BS0	0	
BS1	0100000	
VTDELAY	0040000	Select form-feed and vertical-tab delays:
FF0	0	
FF1	0100000	
CRDELAY	0030000	Select carriage-return delays:
CR0	0	
CR1	0010000	
CR2	0020000	
CR3	0030000	
TBDELAY	0006000	Select tab delays:
TAB0	0	
TAB1	0001000	
TAB2	0004000	
XTABS	0006000	
NLDELAY	0001400	Select new-line delays:
NL0	0	
NL1	0000400	
NL2	0001000	
NL3	0001400	
EVENP	0000200	Even parity allowed on input (most terminals)
ODDP	0000100	Odd parity allowed on input

RAW	0000040	Raw mode: wake up on all characters, 8-bit interfa
CRMOD	0000020	Map CR into LF; echo LF or CR as CR-LF
ECHO	0000010	Echo (full duplex)
LCASE	0000004	Map upper case to lower on input
CBREAK	0000002	Return each character as soon as typed
TANDEM	0000001	Automatic flow control

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay.

Backspace delays are currently ignored but might be used for Terminet 300's.

If a form-feed/vertical tab delay is specified, it lasts for about 2 seconds.

Carriage-return delay type 1 lasts about .08 seconds and is suitable for the Terminet 300. Delay type 2 lasts about .16 seconds and is suitable for the VT05 and the TI 700. Delay type 3 is unimplemented and is 0.

New-line delay type 1 is dependent on the current column and is tuned for Teletype model 37's. Type 2 is useful for the VT05 and is about .10 seconds. Type 3 is unimplemented and is 0.

Tab delay type 1 is dependent on the amount of movement and is tuned to the Teletype model 37. Type 3, called XTABS, is not a delay at all but causes tabs to be replaced by the appropriate number of spaces on output.

Characters with the wrong parity, as determined by bits 200 and 100, are ignored.

In raw mode, every character is passed immediately to the program without waiting until a full line has been typed. No erase or kill processing is done; the end-of-file indicator (EOT), the interrupt character (DEL) and the quit character (FS) are not treated specially. There are no delays and no echoing, and no replacement of one character for another; characters are a full 8 bits for both input and output (parity is up to the program).

Mode 020 causes input carriage returns to be turned into new-lines; input of either CR or LF causes LF-CR both to be echoed (for terminals with a new-line function).

CBREAK is a sort of half-cooked (rare?) mode. Programs can read each character as soon as typed, instead of waiting for a full line, but quit and interrupt work, and output delays,

NAME

iSBC 215/218/220 Driver

DESCRIPTION

The **iSBC 215/218/220** driver controls various mixes of winchester and floppy disk drives. The iSBC 218 is supported _only_ when used as a multi-module on the iSBC 215 controller. The driver can alternatively be used for **iSBC 220** (SMD) devices. The driver supports various features:

(1) Handles multiple iSBC 215/218 and iSBC 220 boards.

(2) Handles 5.25" or 8" floppies on a 215; single- or double-sided.

(3) Has configurable device-characteristics.

(4) Has configurable partition tables.

(5) Handles non-BSIZE sector sizes.

(6) Handles format of 215/218/220 (no automatic alternate-tracking).

Media change is handled as follows: A unit becomes "ready" on its first successful open. A media-change (units 4-7) resets the ready status. The driver insists on "ready" status for I/O requests to be valid. When closed, the open and ready status is reset. Thus, if a media change occurs, the unit must be completely closed before it may be used again. This is to insure cached data for one volume is **not** written to another volume. This is primarily intended for use with floppy disk.

On an error condition, the driver returns either the soft-status or high-byte of the hard-status, whichever has the most information.

Partitions are configured on a sector basis, but must reflect whole tracks. A set of partitions for winchester disk are given below (in 1024-byte sectors):

Priam 3450 Partitions

disk	start	length	tracks
w?t0	0	12	0
w?a	12	6000	1-500
w?b	6012	1188	501-599
w?c	7200	23400	600-2549

Pertec D8020 Partitions

disk	start	length	tracks
pw?t0	0	12	0

The following codes affect characters that are special to the terminal interface. The third argument is a pointer to the following structure, defined in <sgtty.h>:

```
struct tchars {
    char    t_intrc;        /* interrupt */
    char    t_quitc;        /* quit */
    char    t_startc;       /* start output */
    char    t_stopc;        /* stop output */
    char    t_eofc;         /* end-of-file */
    char    t_brkc;         /* input delimiter (like nl) */
};
```

The default values for these characters are DEL, FS, DC1, DC3, EOT, and -1. A character value of -1 eliminates the effect of that character. The t_brkc character, by default -1, acts like a new-line in that it terminates a `line,' is echoed, and is passed to the program. The `stop' and `start' characters may be the same, to produce a toggle effect. It is probably counterproductive to make other special characters (especially erase and kill) identical.

The calling codes for the tchars structure are:

TIOCSETC
: Copy the pointed to tchars structure into the systems working copy, so that the values in the structure take effect, replacing (and destroying) the previous values.

TIOCGETC
: Get the special character structure and store it in the pointed to tchars structure.

FILES

/dev/tty
/dev/tty*
/dev/console

SEE ALSO

getty(8), stty (1), signal(2), ioctl(2)

BUGS

Half-duplex terminals are not supported.

The terminal handler has clearly entered the race for ever-greater complexity and generality. It's still not complex and general enough for TENEX fans.

NAME

iSBC 215/218/220 Driver

DESCRIPTION

The iSBC 215/218/220 driver controls various mixes of winchester and floppy disk drives. The iSBC 218 is supported <u>only</u> when used as a multi-module on the iSBC 215 controller. The driver can alternatively be used for iSBC 220 (SMD) devices. The driver supports various features:

(1) Handles multiple iSBC 215/218 and iSBC 220 boards.

(2) Handles 5.25" or 8" floppies on a 215; single- or double-sided.

(3) Has configurable device-characteristics.

(4) Has configurable partition tables.

(5) Handles non-BSIZE sector sizes.

(6) Handles format of 215/218/220 (no automatic alternate-tracking).

Media change is handled as follows: A unit becomes "ready" on its first successful open. A media-change (units 4-7) resets the ready status. The driver insists on "ready" status for I/O requests to be valid. When closed, the open and ready status is reset. Thus, if a media change occurs, the unit must be completely closed before it may be used again. This is to insure cached data for one volume is **not** written to another volume. This is primarily intended for use with floppy disk.

On an error condition, the driver returns either the soft-status or high-byte of the hard-status, whichever has the most information.

Partitions are configured on a sector basis, but must reflect whole tracks. A set of partitions for winchester disk are given below (in 1024-byte sectors):

Priam 3450 Partitions

disk	start	length	tracks
w?t0	0	12	0
w?a	12	6000	1-500
w?b	6012	1188	501-599
w?c	7200	23400	600-2549

Pertec D8020 Partitions

disk	start	length	tracks
pw?t0	0	12	0

pw?a	12	6000	1-500
pw?b	6012	1188	501-599
pw?c	7200	9360	600-1379

Floppy disks are, by convention, formatted with a single-density 128-byte sectored track 0. This device is **/dev/sft0** and **/dev/rsft0**. Partitions for floppies sectored with various sizes are given below (note that partitions are given in sectors):

1024-byte Single-Sided Floppy

disk	start	length	tracks
f0	26	608	76

1024-byte Double-Sided Floppy

disk	start	length	tracks
df0	26	1224	153

256-byte Single-Sided Floppy

disk	start	length	tracks
xf0	26	1976	76

256-byte Double-Sided Floppy

disk	start	length	tracks
dxf0	26	3978	153

Note that the above partitions do <u>not</u> include the <u>alternate</u> tracks on the winchesters. The driver does not yet support alternate-track formatting.

Raw interfaces exist for all of the above; their names are the same with a prefixed **r**. The raw interfaces include an ioctl that will format a track. This is invoked with the function code I215_IOC_FMT and a structure of the form given below. For more details on track formatting, consult the iSBC 215 manual.

```
struct    i215ftk    {
     int  f_track;
     int  f_intl;
     int  f_skew;
     char f_type;
     char f_pat[4];
};
```

The driver is configured into **c.c** in <u>bdevsw</u>, <u>cdevsw</u>, <u>dinitsw</u>, <u>vecintsw</u>, using the entry points **i215open**, **i215close**, **i215strat**, **i215intr**, **i215ioctl**, **i215read**, **i215write**, and **i215probe**. Device-characteristics and partitions are configured via several tables in a file such as **c215.c**. A different file is necessary to avoid clashes on some symbols with other devices being configured. The configuration consists of filling out the following structures (one i215cfg structure per iSBC 215/218 or 220 controller):

```
struct    i215cfg    {
     unsigned  c_wua;
     char      c_devcod;
     char      c_level;
     struct i215cdrt     *c_drtab[8];
}    i215cfg[N215];

struct    i215cdrt {
     unsigned  cdr_ncyl;
     char      cdr_nfhead;
     char      cdr_nrhead;
     char      cdr_nsec;
     unsigned  cdr_secsiz;
     char      cdr_nalt;
     struct i215part     *cdr_part;
};

struct    i215part {
     daddr_t   p_fsec;
     daddr_t   p_nsec;
};
```

Where:

c_wua
: gives the physical wake-up address of the controller.

c_devcod
: is **DEV8FLPY** for an iSBC 215 with 8" floppies on a 218, **DEV5FLPY** for an iSBC 215 with 5.25" floppies on a 218, **DEV220** for an iSBC 220 controller.

c_level
: is the interrupt level the controller is configured to interrupt at.

c_drtab
: points at the <u>device-characteristics</u> table for each of the 8 possible units connected to the 215/218/220. See the 215 manual for details.

i215cdrt
: gives the device-characteristics for a unit. This is a vector, that allows <u>different</u> device-characteristics to be selected dynamically.

cdr_part
: points at the partition-table for each device-

description. There may be up to 4 partitions per unit.

p_fsec
: gives the first sector number of a partition.

p_nsec
: gives the number of sectors in the partition.

The <u>i215probe</u> procedure is called very early in the initialization of Xenix. It checks the configuration tables and decides which configured boards do actually exist in the hardware configuration. For each board listed in the configuration, an indication of whether or not it exists is printed on the console. This message looks like:

```
     iSBC 215 @ WUA xxxx level y found.
```

or,

```
     iSBC 215 @ WUA xxxx level y NOT found.
```

Where <u>xxxx</u> is the wakeup-address (hex) and <u>y</u> is the interrupt <u>level</u>. If <u>i215probe</u> decides a board doesn't exist, no I/O will be allowed to the units it defines. In paticular, <u>open</u> will fail and return **ENXIO**.

The minor number is divided into several fields:

Bits	Meaning
0-2	Unit Number
3-5	Drtab Number
6-7	Partition Number

Thus, the minor number = 64 * partition + 8 * drtab + unit. In the above, <u>drtab</u> means the device-characteristics table and/or index.

FILES

i215.h	Defines structures
/dev/w*	Block Priam Partitions
/dev/rw*	Raw Priam Partitions
/dev/pw*	Block Pertec Partitions
/dev/rpw*	Raw Pertec Partitions
/dev/*f*	Miscellaneous Floppy Devices
c215.c	iSBC 215/218/220 Specific Configuration

SEE ALSO

format(8)

iSBC 215(Reg.) Winchester Disk Controller Hardware Reference Manual, Intel Corporaton, Order Number 121593-002, Rev A.

iSBX 218(Reg.) Flexible Disk Controller Hardware Reference Manual, **Intel Corporaton,** Order Number 121583-001.

iSBC 220(Reg.) SMD Disk Controller Hardware Reference Manual, **Intel Corporaton,** Order Number 121597-001, Rev A.

BUGS

iSBC 220 support not tested.

14" Winchester disk support not tested.

8" Winchester tested only on Priam 3450 and Pertec D8020.

5.25" Winchester and Floppy not tested.

The error information returned can't properly be fit into one byte.

Concurrent seek is not supported.

Alternate-track formatting is supported, just not tested.

Not tested with multiple winchester drives or multiple floppy drives.

NAME

iSBC 534 Driver

DESCRIPTION

The **iSBC 534** driver controls one or more iSBC 534 four-channel communication expansion boards, and the iSBC 86/xx On-Board USART. Each iSBC 534 has four USARTs. Supported baud rates are 50, 75, 110, 150, 200, 300, 600, 1200, 2400, 4800, and 9600.

The driver is configured into **c.c** in cdevsw, dinitsw, vecintsw, using the entry points **i534open, i534close, i534read, i534write, i534ioctl, i534init, i534intr, i534ciintr, i534cointr.** The 534 and On-Board USART share all procedures except the interrupt handlers; i534ciintr and i534cointr are the interrupt handlers for the On-Board USART. The On-Board USART requires no specific configuration, since its addresses are defined by the iSBC 86/xx boards. The iSBC 534 boards are configured via filling out a structure of the form:

```
struct    i534cfg    {
     int  c_level;
     int  c_base;
};
```

Where:

c_level
: is the interrupt level the controller is configured to interrupt at.

c_base
: specifies the base address of the I/O ports on the board. See the iSBC 534 manual for further details.

The i534init procedure is called very early in the initialization of Xenix. It checks the configuration tables and decides which configured boards do actually exist in the hardware configuration. For each board listed in the configuration, an indication of whether or not it exists is printed on the console. This message looks like:

```
iSBC 534 Based xxxx level y found.
```

or,

```
iSBC 534 Based xxxx level y NOT found.
```

Where xxxx is the base-address of the board (hex), and y is the interrupt level. If i534init decides a board doesn't exist, no I/O will be allowed to the units it defines. In paticular, open will fail and return **ENXIO**.

The minor number is interpreted as:

Minor	Meaning
0-3	1st 534 Board
4-7	2nd 534 Board
.	.
.	.
.	.
0x80	On-Board USART

FILES

i534.h	Defines Structures
/dev/tty?[0-3]	iSBC 534 USARTS
/dev/console	On-Board USART

SEE ALSO

iSBC 534(Reg.) Four Channel Communications Expansion Board Hardware Reference Manual, **Intel Corporaton**, Order Number 9800450-02.

BUGS

Modem answering/calling is not supported.

Baud rates 50, 75, 150, 200, and 600 are not tested.

NAME

a.out - assembler and link editor output

SYNOPSIS

#include <a.out.h>

DESCRIPTION

A.out is the output file of the assembler _as_(1) and the link editor _ld_(1). Both programs make _a.out_ executable if there were no errors and no unresolved external references. Layout information as given in the include file for the PDP11 is:

```
struct  exec {    /* a.out header */
        int       a_magic;    /* magic number */
        unsigned  a_text;     /* size of text segment */
        unsigned  a_data;     /* size of initialized data */
        unsigned  a_bss;      /* size of unitialized data */
        unsigned  a_syms;     /* size of symbol table */
        unsigned  a_entry;    /* entry point */
        unsigned  a_unused;   /* not used */
        unsigned  a_flag;     /* relocation info stripped */
};

#define A_MAGIC1 0407         /* normal */
#define A_MAGIC2 0410         /* read-only text */
#define A_MAGIC3 0411         /* separated I&D */
#define A_MAGIC4 0405         /* overlay */

struct  nlist {   /* symbol table entry */
        char      n_name[8]; /* symbol name */
        int       n_type;    /* type flag */
        unsigned  n_value;   /* value */
};

                  /* values for type flag */
#define N_UNDF    0          /* undefined */
#define N_ABS     01         /* absolute */
#define N_TEXT    02         /* text symbol */
#define N_DATA    03         /* data symbol */
#define N_BSS     04         /* bss symbol */
#define N_TYPE    037
#define N_REG     024        /* register name */
#define N_FN      037        /* file name symbol */
#define N_EXT     040        /* external bit, or'ed in */
#define FORMAT    "%06o"     /* to print a value */
#define FWIDTH    6          /* width of FORMAT */
```

The file has four sections: a header, the program and data text, relocation information, and a symbol table (in that order). The last two may be empty if the program was loaded with the `-s' option of _ld_ or if the symbols and relocation have been removed by _strip_(1).

In the header the sizes of each section are given in bytes, but are even. The size of the header is not included in any of the other sizes.

When an _a.out_ file is loaded into core for execution, three logical segments are set up: the text segment, the data segment (with uninitialized data, which starts off as all 0, following initialized), and a stack. The text segment begins at 0 in the core image; the header is not loaded. If the magic number in the header is 0407(8), it indicates that the text segment is not to be write-protected and shared, so the data segment is immediately contiguous with the text segment. If the magic number is 0410, the data segment begins at the first 0 mod 8K byte boundary following the text segment, and the text segment is not writable by the program; if other processes are executing the same file, they will share the text segment. If the magic number is 411, the text segment is normally pure, write-protected, and shared, and moreover instruction and data space are separated; the text and data segment both begin at location 0. However, if the entry point value is 1, the file is a 23fixed program. If the magic number is 0405, the text segment is overlaid on an existing (0411 or 0405) text segment and the existing data segment is preserved.

The stack will occupy the highest possible locations in the core image: from 0177776(8) and growing downwards. The stack is automatically extended as required. The data segment is only extended as requested by _brk_(2).

The start of the text segment in the file is 020(8); the start of the data segment is $020+S_t$ (the size of the text) the start of the relocation information is $020+S_t+S_d$; the start of the symbol table is $020+2(S_t+S_d)$ if the relocation information is present, $020+S_t+S_d$ if not.

The layout of a symbol table entry and the principal flag values that distinguish symbol types are given in the include file. Other flag values may occur if an assembly language program defines machine instructions.

If a symbol's type is undefined external, and the value field is non-zero, the symbol is interpreted by the loader _ld_ as the name of a common region whose size is indicated by the value of the symbol.

The value of a word in the text or data portions which is not a reference to an undefined external symbol is exactly that value which will appear in core when the file is executed. If a word in the text or data portion involves a reference to an undefined external symbol, as indicated by the relocation information for that word, then the value of the word as stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added into the word in the file.

If relocation information is present, it amounts to one word per word of program text or initialized data. There is no relocation information if the `relocation info stripped' flag in the header is on.

Bits 3-1 of a relocation word indicate the segment referred to by the text or data word associated with the relocation word:

000	absolute number
002	reference to text segment
004	reference to initialized data
006	reference to uninitialized data (bss)
010	reference to undefined external symbol

Bit 0 of the relocation word indicates, if 1, that the reference is relative to the pc (e.g. `clr x'); if 0, that the reference is to the actual symbol (e.g., `clr *$x').

The remainder of the relocation word (bits 15-4) contains a symbol number in the case of external references, and is unused otherwise. The first symbol is numbered 0, the second 1, etc.

SEE ALSO

as(1), ld(1), nm(1), 23fix(8), file(1), cc(1), adb(1)

NAME

acct - execution accounting file

SYNOPSIS

```
#include <sys/acct.h>
```

DESCRIPTION

Acct(2) causes entries to be made into an accounting file for each process that terminates. The accounting file is a sequence of entries whose layout, as defined by the include file is:

```
/*
 * Accounting structures
 */

typedef unsigned short comp_t;/* "floating pt": 3 bits base 8 exp,
struct  acct
{
        char    ac_comm[10]; /* Accounting command name */
        comp_t  ac_utime;    /* Accounting user time */
        comp_t  ac_stime;    /* Accounting system time */
        comp_t  ac_etime;    /* Accounting elapsed time */
        time_t  ac_btime;    /* Beginning time */
        short   ac_uid;      /* Accounting user ID */
        short   ac_gid;      /* Accounting group ID */
        short   ac_mem;      /* average memory usage */
        comp_t  ac_io;       /* number of disk IO blocks */
        dev_t   ac_tty;      /* control typewriter */
        char    ac_flag;     /* Accounting flag */
#ifdef MSLOCAL
        char    ac_nice;                /* nice value from proc tab
        comp_t  ac_fpsim;               /* floating point simulatio
        comp_t  ac_sysc;                /* system calls */
#endif
};

extern  struct  acct         acctbuf;
extern  struct  inode        *acctp;/* inode of accounting file */

#define AFORK   01           /* has executed fork, but no exec */
#define ASU     02           /* used super-user privileges */
```

If the process does an exec(2), the first 10 characters of the filename appear in ac_comm. The accounting flag contains bits indicating whether exec(2) was ever accomplished, and whether the process ever had super-user privileges.

SEE ALSO

acct(2), sa(1)

NAME

ar - archive (library) file format

SYNOPSIS

#include <ar.h>

DESCRIPTION

The archive command *ar* is used to combine several files into one. Archives are used mainly as libraries to be searched by the link-editor *ld*.

A file produced by *ar* has a magic number at the start, followed by the constituent files, each preceded by a file header. The magic number and header layout as described in the include file are:

```
#define ARMAG 0177545
struct  ar_hdr {
        char  ar_name[14];
        long  ar_date;
        char  ar_uid;
        char  ar_gid;
        int   ar_mode;
        long  ar_size;
};
```

The name is a null-terminated string; the date is in the form of *time*(2); the user ID and group ID are numbers; the mode is a bit pattern per *chmod*(2); the size is counted in bytes.

Each file begins on a word boundary; a null byte is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file.

SEE ALSO

ar(1), ld(1), nm(1)

BUGS

Coding user and group IDs as characters is a botch.

NAME

core - format of core image file

DESCRIPTION

UNIX writes out a core image of a terminated process when any of various errors occur. See signal(2) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called `core' and is written in the process's working directory (provided it can be; normal access controls apply).

The first 1024 bytes of the core image are a copy of the system's per-user data for the process, including the registers as they were at the time of the fault; see the system listings for the format of this area. The remainder represents the actual contents of the user's core area when the core image was written. If the text segment is write-protected and shared, it is not dumped; otherwise the entire address space is dumped.

In general the debugger adb(1) is sufficient to deal with core images.

SEE ALSO

adb(1), signal(2)

NAME

dir - format of directories

SYNOPSIS

#include <sys/dir.h>

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry see, filsys(5). The structure of a directory entry as given in the include file is:

```
#ifndef DIRSIZ
#define DIRSIZ14
#endif
struct  direct
{
        ino_t d_ino;
        char  d_name[DIRSIZ];
};
```

By convention, the first two entries in each directory are for `.' and `..'. The first is an entry for the directory itself. The second is for the parent directory. The meaning of `..' is modified for the root directory of the master file system and for the root directories of removable file systems. In the first case, there is no parent, and in the second, the system does not permit off-device references. Therefore in both cases `..' has the same meaning as `.'.

SEE ALSO

filsys(5)

NAME

dump, ddate - incremental dump format

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ino.h>
# include <dumprestor.h>
```

DESCRIPTION

Tapes used by _dump_ and _restor_(1) contain:

a header record
two groups of bit map records
a group of records describing directories
a group of records describing files

The format of the header record and of the first record of each description as given in the include file <_dumprestor.h_> is:

```
#define NTREC     20
#define MLEN      16
#ifdef            SMALL
#define           MSIZ3072
#else
#define MSIZ      4096
#endif

#define TS_TAPE   1
#define TS_INODE  2
#define TS_BITS   3
#define TS_ADDR   4
#define TS_END    5
#define TS_CLRI   6
#define MAGIC     (int)60011
#define CHECKSUM  (int)84446
struct            spcl
{
     int          c_type;
     time_t       c_date;
     time_t       c_ddate;
     int          c_volume;
     daddr_t      c_tapea;
     ino_t        c_inumber;
     int          c_magic;
     int          c_checksum;
     struct       dinodec_dinode;
     int          c_count;
     char         c_addr[BSIZE];
} spcl;

struct            idates
```

```
{
     char          id_name[16];
     char          id_incno;
     time_t        id_ddate;
};
```

NTREC is the number of 512 byte records in a physical tape block. MLEN is the number of bits in a bit map word. MSIZ is the number of bit map words.

The TS_ entries are used in the c_type field to indicate what sort of header this is. The types and their meanings are as follows:

TS_TAPE	Tape volume label
TS_INODE	A file or directory follows. The c_dinode field is a copy of the disk inode and contains bits telling what sort of file this is.
TS_BITS	A bit map follows. This bit map has a one bit for each inode that was dumped.
TS_ADDR	A subrecord of a file description. See c_addr below.
TS_END	End of tape record.
TS_CLRI	A bit map follows. This bit map contains a zero bit for all inodes that were empty on the file system when dumped.
MAGIC	All header records have this number in c_magic.
CHECKSUM	Header records checksum to this value.

The fields of the header structure are as follows:

c_type	The type of the header.
c_date	The date the dump was taken.
c_ddate	The date the file system was dumped from.
c_volume	The current volume number of the dump.
c_tapea	The current number of this (512-byte) record.
c_inumber	The number of the inode being dumped if this is of type TS_INODE.
c_magic	This contains the value MAGIC above, truncated as needed.
c_checksum	This contains whatever value is needed to make the record sum to CHECKSUM.
c_dinode	This is a copy of the inode as it appears on the file system; see filsys(5).
c_count	The count of characters in c_addr.
c_addr	An array of characters describing the blocks of the dumped file. A character is zero if the block associated with that character was not present on

the file system, otherwise the character is non-zero. If the block was not present on the file system, no block was dumped; the block will be restored as a hole in the file. If there is not sufficient space in this record to describe all of the blocks in a file, *TS ADDR* records will be scattered through the file, each one picking up where the last left off.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a *TS END* record and then the tapemark.

The structure *idates* describes an entry of the file /*etc*/*ddate* where dump history is kept. The fields of the structure are:

id_name	The dumped filesystem is `/dev/*id nam*'.
id_incno	The level number of the dump tape; see *dump*(1).
id_ddate	The date of the incremental dump in system format see *types*(5).

FILES

/etc/ddate

SEE ALSO

dump(1), dumpdir(1), restor(1), filsys(5), types(5)

NAME

environ - user environment

SYNOPSIS

extern char **environ;

DESCRIPTION

An array of strings called the `environment' is made available by *exec*(2) when a process begins. By convention these strings have the form `name=value'. The following names are used by various commands:

PATH The sequence of directory prefixes that *sh*, *time*, *nice*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by `:'. *Login*(1) sets PATH=:/bin:/usr/bin.

HOME A user's login directory, set by *login*(1) from the password file *passwd*(5).

TERM The kind of terminal for which output is to be prepared. This information is used by commands, such as *nroff* or *plot*(1), which may exploit special terminal capabilities. See *term*(7) for a list of terminal types.

Further names may be placed in the environment by the *export* command and `name=value' arguments in *sh*(1), or by *exec*(2). It is unwise to conflict with certain Shell variables that are frequently exported by `.profile' files: MAIL, PS1, PS2, IFS.

SEE ALSO

exec(2), sh(1), term(7), login(1)

NAME

filsys, fblk, ino - format of file system volume

SYNOPSIS

#include <sys/param.h>
#include <sys/fblk.h>
#include <sys/filsys.h>
#include <sys/ino.h>

DESCRIPTION

Every file system storage volume (e.g. RF disk, RK disk, RP disk, DECtape reel) has a common format for certain vital information. Every such volume is divided into a certain number of 512-byte blocks. Block 0 is unused and is available to contain a bootstrap program, pack label, or other information.

Block 1 is the super block. The layout of the super block as defined by the include file <sys/filsys.h> is:

```
/*
 * Structure of the super-block
 */
struct filsys {
        unsigned short s_isize;     /* size in blocks of i-list */
        daddr_t  s_fsize;           /* size in blocks of entire volum
        short    s_nfree;           /* number of addresses in s_free
        daddr_t  s_free[NICFREE];/* free block list */
        short    s_ninode;          /* number of i-nodes in s_inode *
        ino_t    s_inode[NICINOD];/* free i-node list */
        char     s_flock;           /* lock during free list manipula
        char     s_ilock;           /* lock during i-list manipulatio
        char     s_fmod;            /* super block modified flag */
        char     s_ronly;           /* mounted read-only flag */
        time_t   s_time;            /* last super block update */
        /* remainder not maintained by this version of the system */
        daddr_t  s_tfree;           /* total free blocks*/
        ino_t    s_tinode;          /* total free inodes */
        short    s_m;               /* interleave factor */
        short    s_n;               /* " " */
        char     s_fname[6];        /* file system name */
        char     s_fpack[6];        /* file system pack name */
        /* remainder is maintained for xenix */
        char     s_clean;           /* S_CLEAN if structure is proper
};

#define         S_CLEAN             0106        /* arbitrary magic va
```

S_isize is the address of the first block after the i-list, which starts just after the super-block, in block 2. Thus the i-list is s_isize-2 blocks long. S_fsize is the address of the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block addresses; if an `impossible' block address is allocated from the free list or is freed, a diagnostic is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The *s free* array contains, in *s free*[*1*], ... , *s free*[*s nfree-1*], up to NICFREE free block numbers. NICFREE is a configuration constant. *S free*[*0*] is the block address of the head of a chain of blocks constituting the free list. The layout of each block of the free chain as defined in the include file <*sys/fblk.h*> is:

```
struct fblk
{
        int       df_nfree;
        daddr_t   df_free[NICFREE];
};
```

The fields *df nfree* and *df free* in a free block are used exactly like *s nfree* and *s free* in the super block. To allocate a block: decrement *s nfree*, and the new block number is *s free*[*s nfree*]. If the new block address is 0, there are no blocks left, so give an error. If *s nfree* became 0, read the new block into *s nfree* and *s free*. To free a block, check if *s nfree* is NICFREE; if so, copy *s nfree* and the *s free* array into it, write it out, and set *s nfree* to 0. In any event set *s free*[*s nfree*] to the freed block's address and increment *s nfree*.

S ninode is the number of free i-numbers in the *s inode* array. To allocate an i-node: if *s ninode* is greater than 0, decrement it and return *s inode*[*s ninode*]. If it was 0, read the i-list and place the numbers of all free inodes (up to NICINOD) into the *s inode* array, then try again. To free an i-node, provided *s ninode* is less than NICINODE, place its number into *s inode*[*s ninode*] and increment *s ninode*. If *s ninode* is already NICINODE, don't bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

S flock and *s ilock* are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of *s fmod* on disk is likewise immaterial; it is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information. *S ronly* is a write-protection indicator; its disk value is also immaterial.

S time is the last time the supttem
was changed. During a reboot, s time of the super-block for the root file system is used to set the system's idea of the time.

The fields s tfree, s tinode, s fname and s fpack are not currently maintained.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. I-nodes are 64 bytes long, so 8 of them fit into a block. I-node 2 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. The format of an i-node as given in the include file <sys/ino.h> is:

```
/*
 * Inode structure as it appears on
 * a disk block.
 */
struct dinode
{
        unsigned short      di_mode;     /* mode and type of fil
        short   di_nlink;    /* number of links to file */
        short   di_uid;      /* owner's user id */
        short   di_gid;      /* owner's group id */
        off_t   di_size;     /* number of bytes in file */
        char    di_addr[40]; /* disk block addresses */
        time_t  di_atime;    /* time last accessed */
        time_t  di_mtime;    /* time last modified */
        time_t  di_ctime;    /* time created */
};
#define INOPB   8            /* 8 inodes per block */
/*
 * the 40 address bytes:
 *      39 used; 13 addresses
 *      of 3 bytes each.
 */
```

Di mode tells the kind of file; it is encoded identically to the st mode field of stat(2). Di nlink is the number of directory entries (links) that refer to this i-node. Di uid and di gid are the owner's user and group IDs. Size is the number of bytes in the file. Di atime and di mtime are the times of last access and modification of the file contents (read, write or create) (see times(2)); Di ctime records the time of last modification to the inode or to the file, and is used to determine whether it should be dumped.

Special files are recognized by their modes and not by i-number. A block-type special file is one which can potentially be mounted as a file system; a character-type special file cannot, though it is not necessarily character-oriented. For special files, the _di addr_ field is occupied by the device code (see _types_(5)). The device codes of block and character special files overlap.

Disk addresses of plain files and directories are kept in the array _di addr_ packed into 3 bytes each. The first 10 addresses specify device blocks directly. The last 3 addresingly, doubly, and triply indirect and point to blocks of 128 block pointers. Pointers in indirect blocks have the type _daddr t_ (see _types_(5)).

For block a file to exist, it is not necessary that all blocks less than _b_ exist. A zero block number either in the address words of the i-node or in an indirect block indicates that the corresponding block has never been allocated. Such a missing block reads as if it contained all zero words.

SEE ALSO

icheck(1), dcheck(1), shutdn(2), dir(5), mount(1), stat(2), types(5), fsck(1M),

NAME

group - group file

DESCRIPTION

Group contains for each group the following information:

group name
encrypted password
numerical group ID
a comma separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; Each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

FILES

/etc/group

SEE ALSO

newgrp(1), crypt(3), passwd(1), passwd(5)

NAME

mpxio - multiplexed i/o

SYNOPSIS

#include <sys/mx.h>

#include <sgtty.h>

DESCRIPTION

Data transfers on mpx files (see mpx(2)) are multiplexed by imposing a record structure on the io stream. Each record represents data from/to a particular channel or a control or status message associated with a particular channel.

The prototypical data record read from an mpx file is as follows

```
struct input_record {
     short     index;
     short     count;
     short     ccount;
     char data[];
};
```

where index identifies the channel, and count specifies the number of characters in data. If count is zero, ccount gives the size of data, and the record is a control or status message. Although count or ccount might be odd, the operating system aligns records on short (i.e. 16-bit) boundaries by skipping bytes when necessary.

Data written to an mpx file must be formatted as an array of record structures defined as follows

```
struct output_record {
     short     index;
     short     count;
     short     ccount;
     char *data;
};
```

where the data portion of the record is referred to indirectly and the other cells have the same interpretation as in input record.

The control messages listed below may be read from a multiplexed file descriptor. They are presented as two 16-bit integers: the first number is the message code (defined in <sys/mx.h>), the second is an optional parameter meaningful only with M_WATCH and M_BLK.

M_WATCH - a process `wants to attach' on this channel. The second parameter is the 16-bit user-id of the process that executed the open.

M_CLOSE - the channel is closed. This message is generated when the last file descriptor referencing a channel is closed. The *detach* command (see *mpx*(2) should be used in response to this message.

M_EOT - indicates logical end of file on a channel. If the channel is joined to a typewriter, EOT (control-d) will cause the M_EOT message under the conditions specified in *tty*(4) for end of file. If the channel is attached to a process, M_EOT will be generated whenever the process writes zero bytes on the channel.

M_BLK - if non-blocking mode has been enabled on an mpx file descriptor *xd* by executing *ioctl*(*xd*, *MXNBLK*, *0*), write operations on the file are truncated in the kernel when internal queues become full. This is done on a per-channel basis: the parameter is a count of the number of characters not transferred to the channel on which M_BLK is received.

M_UBLK - is generated for a channel after M_BLK when the internal queues have drained below a threshold.

Two other messages may be generated by the kernel. As with other messages, the first 16-bit quantity is the message code.

M_OPEN - is generated in conjunction with `listener' mode (see *mpx*(2)). The uid of the calling process follows the message code as with M_WATCH. This is followed by a null-terminated string which is the name of the file being opened.

M_IOCTL - is generated for a channel connected to a process when that process executes the *ioctl*(*fd*, *cmd*, &*vec*) call on the channel file descriptor. The M_IOCTL code is followed by the *cmd* argument given to *ioctl* followed by the contents of the structure *vec*. It is assumed, not needing a better compromise at this time, that the length of *vec* is determined by *sizeof* (*struct sgttyb*) as declared in <*sgtty.h*>.

Two control messages are understood by the operating system. M_EOT may be sent through an mpx file to a channel. It is equivalent to propagating a zero-length record through the channel; i.e. the channel is allowed to drain and the process or device at the other end receives a zero-length transfer before data starts flowing through the channel again. M_IOCTL can also be sent through a channel. The format is identical to that described above.

NAME

mtab - mounted file system table

DESCRIPTION

Mtab resides in directory _/etc_ and contains a table of devices mounted by the _mount_ command. _Umount_ removes entries.

Each entry is 64 bytes long; the first 32 are the null-padded name of the place where the special file is mounted; the second 32 are the null-padded name of the special file. The special file has all its directories stripped away; that is, everything through the last \`/' is thrown away.

This table is present only so people can look at it. It does not matter to _mount_ if there are duplicated entries nor to _umount_ if a name cannot be found.

FILES

/etc/mtab

SEE ALSO

mount(1)

NAME

passwd - password file

DESCRIPTION

Passwd contains for each user the following information:

name (login name, contains no upper case)
encrypted password
numerical user ID
numerical group ID
GCOS job number, box number, optional GCOS user-id
initial working directory
program to use as Shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. The GCOS field is used only when communicating with that system, and in other installations can contain any desired information. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, the Shell itself is used.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user ID's to names.

Some programs depend on certain entries (such as "daemon" and "root"); these should not be removed.

FILES

/etc/passwd

SEE ALSO

getpwent(3), login(1), crypt(3), passwd(1), group(5), cron(8)

NAME

plot - graphics interface

DESCRIPTION

Files of this format are produced by routines described in plot(3), and are interpreted for various devices by commands described in plot(1). A graphics file is a stream of plotting instructions. Each instruction consists of an ASCII letter usually followed by bytes of binary information. The instructions are executed in order. A point is designated by four bytes representing the x and y values; each value is a signed integer. The last designated point in an l, m, n, or p instruction becomes the `current point' for the next instruction.

Each of the following descriptions begins with the name of the corresponding routine in plot(3).

m move: The next four bytes give a new current point.

n cont: Draw a line from the current point to the point given by the next four bytes. See plot(1).

p point: Plot the point given by the next four bytes.

l line: Draw a line from the point given by the next four bytes to the point given by the following four bytes.

t label: Place the following ASCII string so that its first character falls on the current point. The string is terminated by a newline.

a arc: The first four bytes give the center, the next four give the starting point, and the last four give the end point of a circular arc. The least significant coordinate of the end point is used only to determine the quadrant. The arc is drawn counter-clockwise.

c circle: The first four bytes give the center of the circle, the next two the radius.

e erase: Start another frame of output.

f linemod: Take the following string, up to a newline, as the style for drawing further lines. The styles are `dotted,' `solid,' `longdashed,' `shortdashed,' and `dotdashed.' Effective only in plot 4014 and plot ver.

s space: The next four bytes give the lower left corner of the plotting area; the following four give the upper right corner. The plot will be magnified or reduced to fit the device as closely as possible.

Space settings that exactly fill the plotting area with unity scaling appear below for devices supported by the filters of *plot*(1). The upper limit is just outside the plotting area. In every case the plotting area is taken to be square; points outside may be displayable on devices whose face isn't square.

```
4014       space(0, 0, 3120, 3120);
ver        space(0, 0, 2048, 2048);
300, 300s  space(0, 0, 4096, 4096);
450        space(0, 0, 4096, 4096);
```

SEE ALSO

plot(1), plot(3), graph(1)

NAME

termcap - terminal capability data base

SYNOPSIS

/etc/termcap

DESCRIPTION

Termcap is a data base describing terminals, used, *e.g.*, by *vi*(1) and *curses*(3). Terminals are described in *termcap* by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in *termcap*.

Entries in *termcap* consist of a number of `:' separated fields. The first entry for each terminal gives the names which are known for the terminal, separated by `|' characters. The first name is always 2 characters long and is used by older version 6 systems which store the terminal type in a 16 bit word in a systemwide data base. The second name given is the most common abbreviation for the terminal, and the last name given should be a long name fully identifying the terminal. The second name should contain no blanks; the last name may well contain blanks for readability.

CAPABILITIES

(P) indicates padding may be specified
(P*) indicates that padding may be based on no. lines affected

Name	Type	Pad?	Description
ae	str	(P)	End alternate character set
al	str	(P*)	Add new blank line
am	bool		Terminal has automatic margins
as	str	(P)	Start alternate character set
bc	str		Backspace if not ^H
bs	bool		Terminal can backspace with ^H
bt	str	(P)	Back tab
bw	bool		Backspace wraps from column 0 to last column
CC	str		Command character in prototype if terminal setta
cd	str	(P*)	Clear to end of display
ce	str	(P)	Clear to end of line
ch	str	(P)	Like cm but horizontal motion only, line stays s
cl	str	(P*)	Clear screen
cm	str	(P)	Cursor motion
co	num		Number of columns in a line
cr	str	(P*)	Carriage return, (default ^M)
cs	str	(P)	Change scrolling region (vt100), like cm
cv	str	(P)	Like ch but vertical only.
da	bool		Display may be retained above
dB	num		Number of millisec of bs delay needed
db	bool		Display may be retained below
dC	num		Number of millisec of cr delay needed

dc	str	(P*)	Delete character
dF	num		Number of millisec of ff delay needed
dl	str	(P*)	Delete line
dm	str		Delete mode (enter)
dN	num		Number of millisec of nl delay needed
do	str		Down one line
dT	num		Number of millisec of tab delay needed
ed	str		End delete mode
ei	str		End insert mode; give :ei=: if **ic**
eo	str		Can erase overstrikes with a blank
ff	str	(P*)	Hardcopy terminal page eject (default ^L)
hc	bool		Hardcopy terminal
hd	str		Half-line down (forward 1/2 linefeed)
ho	str		Home cursor (if no **cm**)
hu	str		Half-line up (reverse 1/2 linefeed)
hz	str		Hazeltine; can't print ~'s
ic	str	(P)	Insert character
if	str		Name of file containing **is**
im	bool		Insert mode (enter); give :im=: if **ic**
in	bool		Insert mode distinguishes nulls on display
ip	str	(P*)	Insert pad after character inserted
is	str		Terminal initialization string
k0-k9	str		Sent by other function keys 0-9
kb	str		Sent by backspace key
kd	str		Sent by terminal down arrow key
ke	str		Out of keypad transmit mode
kh	str		Sent by home key
kl	str		Sent by terminal left arrow key
kn	num		Number of other keys
ko	str		Termcap entries for other non-function keys
kr	str		Sent by terminal right arrow key
ks	str		Put terminal in keypad transmit mode
ku	str		Sent by terminal up arrow key
l0-l9	str		Labels on other function keys
li	num		Number of lines on screen or page
ll	str		Last line, first column (if no **cm**)
ma	str		Arrow key map, used by vi version 2 only
mi	bool		Safe to move while in insert mode
ml	str		Memory lock on above cursor.
mu	str		Memory unlock (turn off memory lock).
nc	bool		No correctly working carriage return (DM250
nd	str		Non-destructive space (cursor right)
nl	str	(P*)	Newline character (default **\n**)
ns	bool		Terminal is a CRT but doesn't scroll.
os	bool		Terminal overstrikes
pc	str		Pad character (rather than null)
pt	bool		Has hardware tabs (may need to be set with
se	str		End stand out mode
sf	str	(P)	Scroll forwards
sg	num		Number of blank chars left by so or se
so	str		Begin stand out mode
sr	str	(P)	Scroll reverse (backwards)

ta	str	(P)	Tab (n ^I or with padding)
tc	str		Entry of similar terminal - must be last
te	str		String to end programs that use **cm**
ti	str		String to begin programs that use **cm**
uc	str		Underscore one char and move past it
ue	str		End underscore mode
ug	num		Number of blank chars left by us or ue
ul	bool		Terminal underlines even though it doesn't overs
up	str		Upline (cursor up)
us	str		Start underscore mode
vb	str		Visible bell (may not move cursor)
ve	str		Sequence to end open/visual mode
vs	str		Sequence to start open/visual mode
xb	bool		Beehive (f1=escape, f2=ctrl C)
xn	bool		A newline is ignored after a wrap (Concept)
xr	bool		Return acts like **ce** \r \n (Delta Data)
xs	bool		Standout not erased by writing over it (HP 264?)
xt	bool		Tabs are destructive, magic so char (Teleray 106

A Sample Entry

The following entry, which describes the Concept-100, is among the more complex entries in the termcap file as of this writing. (This particular concept entry is outdated, and is used as an example only.)

```
cl|c100|concept100:is=\EU\Ef\E7\E5\E8\El\ENH\EK\E\200\Eo&\200:\
        :al=3*\E^R:am:bs:cd=16*\E^C:ce=16\E^S:cl=2*^L:cm=\Ea%+ %+ :
        :dc=16\E^A:dl=3*\E^B:ei=\E\200:eo:im=\E^P:in:ip=16*:li#24:m
        :se=\Ed\Ee:so=\ED\EE:ta=8\t:ul:up=\E;:vb=\Ek\EK:xn:
```

Entries may continue onto multiple lines by giving a \ as the last character of a line, and that empty fields may be included for readability (here between the last field on a line and the first field on the next). Capabilities in termcap are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have two letter codes. For instance, the fact that the Concept has automatic margins (i.e. an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**. Numeric capabilities are followed by the character `#' and then the value. Thus **co** which indicates the number of columns the terminal has gives the value `80' for the Concept.

Finally, string valued capabilities, such as **ce** (clear to end of line sequence) are given by the two character code, an \`=', and then a string ending at the next following \`:'. A delay in milliseconds may appear after the \`=' in such a capability, and padding characters are supplied by the editor after the remainder of the string is sent to provide this delay. The delay can be either a integer, e.g. \`20', or an integer followed by an \`*', i.e. \`3*'. A \`*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. When a \`*' is specified, it is sometimes useful to give a delay of the form \`3.5' specify a delay per unit to tenths of milliseconds.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. A \E maps to an ESCAPE character, ^x maps to a control-x for any appropriate x, and the sequences \n \r \t \b \f give a newline, return, tab, backspace and formfeed. Finally, characters may be given as three octal digits after a \, and the characters ^ and \ may be given as \^ and \\. If it is necessary to place a : in a capability it must be escaped in octal as \072. If it is necessary to place a null character in a string capability it must be encoded as \200. The routines which deal with termcap use C strings, and strip the high bits of the output very late so that a \200 comes out as a \000 would.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in termcap and to build up a description gradually, using partial descriptions with ex to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the termcap file to describe it or bugs in ex. To easily test a new terminal description you can set the environment variable TERMCAP to a pathname of a file containing the description you are working on and the editor will look there rather than in /etc/termcap. TERMCAP can also be set to the termcap entry itself to avoid reading the file when starting up the editor. (This only works on version 7 systems.)

Basic capabilities

The number of columns on each line for the terminal is given by the **co** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **li** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, then this is given by the **cl** string capability. If the terminal can backspace, then it should have the **bs** capability, unless a backspace is accomplished by a character other than ^H (ugh) in which case you should give this character as the **bc** string capability. If it overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability.

A very important point here is that the local cursor motions encoded in termcap are undefined at the left and top edges of a CRT terminal. The editor will never attempt to backspace around the left edge, nor will it attempt to go up locally off the top. The editor assumes that feeding off the bottom of the screen will cause the screen to scroll up, and the **am** capability tells whether the cursor sticks at the right edge of the screen. If the terminal has switch selectable automatic margins, the termcap file usually assumes that this is on, i.e. **am**.

These capabilities suffice to describe hardcopy and glass-tty terminals. Thus the model 33 teletype is described as

```
t3|33|tty33:co#72:os
```

while the Lear Siegler ADM-3 is described as

```
cl|adm3|3|lsi adm3:am:bs:cl=^Z:li#24:co#80
```

Cursor addressing

Cursor addressing in the terminal is described by a **cm** string capability, with printf(3s) like escapes **%x** in it. These substitute to encodings of the current line or column position, while other characters are passed through unchanged. If the **cm** string is thought of as being a function, then its arguments are the line and then the column to which motion is desired, and the % encodings have the following meanings:

%d	as in printf, 0 origin
%2	like %2d
%3	like %3d
%.	like %c
%+x	adds x to value, then %.
%>xy	if value > x adds y, no output.
%r	reverses order of line and column, no output
%i	increments line/column (for 1 origin)
%%	gives a single %
%n	exclusive or row and column with 0140 (DM2500)
%B	BCD (16*(x/10)) + (x%10), no output.

%D Reverse coding (x-2*(x%16)), no output. (Delta Data).

Consider the HP2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cm** capability is cm=6\E&%r%2c%2Y. The Microterm ACT-IV needs the current row and column sent preceded by a ^T, with the row and column simply encoded in binary, cm=^T%.%.. Terminals which use %. need to be able to backspace the cursor (**bs** or **bc**), and to move the cursor up one line on the screen (**up** introduced below). This is necessary because it is not always safe to transmit **\t**, **\n** **^D** and **\r**, as the system may change or discard them.

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus cm=\E=%+ %+ .

Cursor motions

If the terminal can move the cursor one position to the right, leaving the character at the current position unchanged, then this sequence should be given as **nd** (non-destructive space). If it can move the cursor up a line on the screen in the same column, this should be given as **up**. If the terminal has no cursor addressing capability, but can home the cursor (to very upper left corner of screen) then this can be given as **ho**; similarly a fast way of getting to the lower left hand corner can be given as **ll**; this may involve going up with **up** from the home position, but the editor will never do this itself (unless **ll** does) because it makes no assumption about the effect of moving up from the home position.

Area clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **ce**. If the terminal can clear from the current position to the end of the display, then this should be given as **cd**. The editor only uses **cd** from the first column of a line.

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **al**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl**; this is done only from the first position on the line to be deleted. If the terminal can scroll the screen backwards, then this can be given as **sb**, but just **al** suffices. If the terminal can retain display memory above then the **da** capability should be given; if display memory can be retained below then **db** should be given. These let the editor understand that deleting a line on the screen may bring non-blank lines up from below or that scrolling back with **sb** may bring down non-blank lines.

Insert/delete character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using termcap. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can find out which kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type abc   def using local cursor motions (not spaces) between the abc and the def. Then position the cursor before the abc and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the abc shifts over to the def which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for insert null. If your terminal does something different and unusual then you may have to modify the editor to get it to use the insert mode your terminal defines. We have seen no terminals which have an insert mode not not falling into one of these two classes.

The editor can handle both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **im** the sequence to get into insert mode, or give it an empty value if your terminal uses a sequence to insert a blank position. Give as **ei** the sequence to leave insert mode (give this, with an empty value also if you gave **im** so). Now give as **ic** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ic**, terminals which send a sequence to open a screen position should give it here. (Insert mode is preferable to the sequence to open a position on the screen if your terminal has both.) If post insert padding is needed, give this as a number of milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g. if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mi** to speed up inserting in this case. Omitting **mi** will affect only speed. Some terminals (notably Datamedia's) must not have **mi** because of the way their insert mode works.

Finally, you can specify delete mode by giving **dm** and **ed** to enter and exit delete mode, and **dc** to delete a single character while in delete mode.

Highlighting, underlining, and visible bells

If your terminal has sequences to enter and exit standout mode these can be given as **so** and **se** respectively. If there are several flavors of standout mode (such as inverse video, blinking, or underlining - half bright is not usually an acceptable standout mode unless the terminal is in inverse video mode constantly) the preferred mode is inverse video by itself. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, this is acceptable, and although it may confuse some programs slightly, it can't be helped.

Codes to begin underlining and end underlining can be given as **us** and **ue** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as **uc**. (If the underline code does not move the cursor to the right, give the code followed by a nondestructive space.)

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **vb**; it must not move the cursor. If the terminal should be placed in a different mode during open and visual modes of _ex_, this can be given as **vs** and **ve**, sent at the start and end of these modes respectively. These can be used to change, e.g., from a underline to a block cursor and back.

If the terminal needs to be in a special mode when running a program that addresses the cursor, the codes to enter and exit this mode can be given as **ti** and **te**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly.

If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted HP 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **ks** and **ke**. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kl**, **kr**, **ku**, **kd**, and **kh** respectively. If there are function keys such as f0, f1, ..., f9, the codes they send can be given as **k0**, **k1**, **...**, **k9**. If these keys have labels other than the default f0 through f9, the labels can be given as **l0**, **l1**, **...**, **l9**. If there are other keys that transmit the same code as the terminal expects for the corresponding function, such as clear screen, the termcap 2 letter codes can be given in the **ko** capability, for example, :ko=cl,ll,sf,sb:, which says that the terminal has clear, home down, scroll down, and scroll up keys that transmit the same thing as the cl, ll, sf, and sb entries.

The **ma** entry is also used to indicate arrow keys on terminals which have single character arrow keys. It is obsolete but still in use in version 2 of vi, which must be run on some minicomputers due to memory limitations. This field is redundant with **kl**, **kr**, **ku**, **kd**, and **kh**. It consists of groups of two characters. In each group, the first character is what an arrow key sends, the second character is the corresponding vi command. These commands are **h** for **kl**, **j** for **kd**, **k** for **ku**, **l** for **kr**, and H for **kh**. For example, the mime would be :ma=^Kj^Zk^Xl: indicating arrow keys left (^H), down (^K), up (^Z), and right (^X). (There is no home key on the mime.)

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pc**.

If tabs on the terminal require padding, or if the terminal uses a character other than ^I to tab, then this can be given as **ta**.

Hazeltine terminals, which don't allow `~' characters to be printed should indicate **hz**. Datamedia terminals, which echo carriage-return linefeed for carriage return and then ignore a following linefeed should indicate **nc**. Early Concept terminals, which ignore a linefeed immediately after an **am** wrap, should indicate **xn**. If an erase-eol is required to get rid of standout (instead of merely writing on top of it), **xs** should be given. Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt**. Other specific terminal problems may be corrected by adding more capabilities of the form **x*x***.

Other capabilities include **is**, an initialization string for the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to properly clear and then set the tabs on the terminal, if the terminal has settable tabs. If both are given, is will be printed before **if**. This is useful where **if** is /*usr*/*lib*/*tabset*/*std* but **is** clears the tabs first.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **tc** can be given with the name of the similar terminal. This capability must be *last* and the combined length of the two entries must not exceed 1024. Since *termlib* routines search the entry from left to right, and since the tc capability is replaced by the corresponding entry, the capabilities given at the left override the ones in the similar terminal. A capability can be cancelled with **xx@** where xx is the capability. For example, the entry

```
hn|2621nl:ks@:ke@:tc=2621:
```

defines a 2621nl that does not have the **ks** or **ke** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES

/etc/termcap	file containing terminal descriptions

SEE ALSO

ex(1), curses(3), termcap(3), tset(1), vi(1), ul(1), more(1)

AUTHOR

William Joy
Mark Horton added underlining and keypad support

BUGS

Ex allows only 256 characters for string capabilities, and the routines in *termcap*(*3*) do not check for overflow of this buffer. The total length of a single entry (excluding only escaped newlines) may not exceed 1024.

The **ma**, **vs**, and **ve** entries are specific to the *vi* program.

Not all programs support all entries. There are entries that are not supported by any program.

NAME

tp - DEC/mag tape formats

DESCRIPTION

The command tp dumps files to and extracts files from DECtape and magtape. The formats of these tapes are the same except that magtapes have larger directories.

Block zero contains a copy of a stand-alone bootstrap program. See bproc(8).

Blocks 1 through 24 for DECtape (1 through 62 for magtape) contain a directory of the tape. There are 192 (resp. 496) entries in the directory; 8 entries per block; 64 bytes per entry. Each entry has the following format:

```
struct {
     char pathname[32];
     int  mode;
     char uid;
     char gid;
     char unused1;
     char size[3];
     long modtime;
     int  tapeaddr;
     char unused2[16];
     int  checksum;
};
```

The path name entry is the path name of the file when put on the tape. If the pathname starts with a zero word, the entry is empty. It is at most 32 bytes long and ends in a null byte. Mode, uid, gid, size and time modified are the same as described under i-nodes (see file system filsys(5)). The tape address is the tape block number of the start of the contents of the file. Every file starts on a block boundary. The file occupies (size+511)/512 blocks of continuous tape. The checksum entry has a value such that the sum of the 32 words of the directory entry is zero.

Blocks above 25 (resp. 63) are available for file storage.

A fake entry has a size of zero.

SEE ALSO

filsys(5), tp(1)

BUGS

The pathname, uid, gid, and size fields are too small.

NAME

ttys - terminal initialization data

DESCRIPTION

The *ttys* file is read by the *init* program and specifies which terminal special files are to have a process created for them which will allow people to log in. It contains one line per special file.

The first character of a line is either `0' or `1'; the former causes the line to be ignored, the latter causes it to be effective. The second character is used as an argument to *getty*(8), which performs such tasks as baud-rate recognition, reading the login name, and calling *login*. For normal lines, the character is `0'; other characters can be used, for example, with hard-wired terminals where speed recognition is unnecessary or which have special characteristics. (*Getty* will have to be fixed in such cases.) The remainder of the line is the terminal's entry in the device directory, /dev.

FILES

/etc/ttys

SEE ALSO

init(8), getty(8), login(1)

NAME

types - primitive system data types

SYNOPSIS

#include <sys/types.h>

DESCRIPTION

The data types defined in the include file are used in UNIX system code; some data of these types are accessible to user code:

```
typedef   long           daddr_t;       /* disk address */
typedef   char *         caddr_t;       /* core address */
typedef   unsigned int   ino_t;         /* i-node number */
typedef   long           time_t;        /* a time */
/* HCR Jan 81 -- add 2 words to label_t */
typedef   int            label_t[6+2];  /* program status */
typedef   int            dev_t;         /* device code */
typedef   long           off_t;         /* offset in file */
     /* selectors and constructor for device code */
#define   major(x)       (int)(((unsigned)x>>8))
#define   minor(x)       (int)(x&0377)
#define   makedev(x,y)   (dev_t)((x)<<8|(y))
```

The form daddr_t is used for disk addresses except in an i-node on disk, see filsys(5). Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The label_t variables are used to save the processor state while another process is running.

SEE ALSO

filsys(5), time(2), lseek(2), adb(1)

NAME

utmp, wtmp - login records

SYNOPSIS

#include <utmp.h>

DESCRIPTION

The *utmp* file allows one to discover information about who is currently using UNIX. The file is a sequence of entries with the following structure declared in the include file:

```
struct utmp {
     char ut_line[8];          /* tty name */
     char ut_name[8];          /* user id */
     long ut_time;        /* time on */
};
```

This structure gives the name of the special file associated with the user's terminal, the user's login name, and the time of the login in the form of *time*(2).

The *wtmp* file records all logins and logouts. Its format is exactly like *utmp* except that a null user name indicates a logout on the associated terminal. Furthermore, the terminal name `~' indicates that the system was rebooted at the indicated time; the adjacent pair of entries with terminal names `|' and `}' indicate the system-maintained time just before and just after a *date* command has changed the system's idea of the time.

Wtmp is maintained by *login*(1) and *init*(8). Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by *ac*(1).

FILES

/etc/utmp
/usr/adm/wtmp

SEE ALSO

login(1), init(8), who(1), ac(1)

APPENDIX A: C Reference Manual

What follows is the _C Reference Manual_ from Kernighan and Ritchie's _The C Programming Language_. It is provided here for reference and is not intended to teach C programming.

The C Programming Language — Reference Manual

Dennis M. Ritchie

Bell Laboratories, Murray Hill, New Jersey

This manual is reprinted, with minor changes, from *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1978.

1. Introduction

This manual describes the C language on the DEC PDP-11, the DEC VAX-11, the Honeywell 6000, the IBM System/370, and the Interdata 8/32. Where differences exist, it concentrates on the PDP-11, but tries to point out implementation-dependent details. With few exceptions, these dependencies follow directly from the underlying properties of the hardware; the various compilers are generally quite compatible.

2. Lexical conventions

There are six classes of tokens: identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, newlines, and comments (collectively, "white space") as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1 Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`. Comments do not nest.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper and lower case letters are different. No more than the first eight characters are significant, although more may be used. External identifiers, which are used by various assemblers and loaders, are more restricted:

DEC PDP-11	7 characters, 2 cases
DEC VAX-11	8 characters, 2 cases
Honeywell 6000	6 characters, 1 case
IBM 360/370	7 characters, 1 case
Interdata 8/32	8 characters, 2 cases

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
int         extern      else
char        register    for
float       typedef     do
double      static      while
struct      goto        switch
union       return      case
long        sizeof      default
short       break       entry
unsigned    continue
auto        if
```

The `entry` keyword is not currently implemented by any compiler but is reserved for future use. Some

† UNIX is a Trademark of Bell Laboratories.

implementations also reserve the words `fortran` and `asm`.

2.4 Constants

There are several kinds of constants, as listed below. Hardware characteristics which affect sizes are summarized in §2.6.

2.4.1 Integer constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with 0 (digit zero), decimal otherwise. The digits `8` and `9` have octal value 10 and 11 respectively. A sequence of digits preceded by `0x` or `0X` (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include `a` or `A` through `f` or `F` with values 10 through 15. A decimal constant whose value exceeds the largest signed machine integer is taken to be `long`; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be `long`.

2.4.2 Explicit long constants

A decimal, octal, or hexadecimal integer constant immediately followed by `l` (letter ell) or `L` is a long constant. As discussed below, on some machines integer and long values may be considered identical.

2.4.3 Character constants

A character constant is a character enclosed in single quotes, as in `'x'`. The value of a character constant is the numerical value of the character in the machine's character set.

Certain non-graphic characters, the single quote ' and the backslash \, may be represented according to the following table of escape sequences:

newline	NL (LF)	`\n`
horizontal tab	HT	`\t`
backspace	BS	`\b`
carriage return	CR	`\r`
form feed	FF	`\f`
backslash	\	`\\`
single quote	'	`\'`
bit pattern	*ddd*	\ddd

The escape \ddd consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is \0 (not followed by a digit), which indicates the character NUL. If the character following a backslash is not one of those specified, the backslash is ignored.

2.4.4 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an `e` or `E`, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the `e` and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.

2.5 Strings

A string is a sequence of characters surrounded by double quotes, as in `"..."`. A string has type "array of characters" and storage class `static` (see §4 below) and is initialized with the given characters. All strings, even when written identically, are distinct. The compiler places a null byte \0 at the end of each string so that programs which scan the string can find its end. In a string, the double quote character " must be preceded by a \; in addition, the same escapes as described for character constants may be used. Finally, a \ and an immediately following newline are ignored.

2.6 Hardware characteristics

The following table summarizes certain hardware properties which vary from machine to machine. Although these affect program portability, in practice they are less of a problem than might be thought *a priori*.

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	EBCDIC	ASCII
`char`	8 bits	9 bits	8 bits	8 bits
`int`	16	36	32	32
`short`	16	36	16	16
`long`	32	36	32	32
`float`	32	36	32	32
`double`	64	72	64	64
range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 76}$

The VAX-11 is identical to the PDP-11 except that integers have 32 bits.

3. Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in **bold** type. Alternative categories are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript "opt," so that

{ *expression*$_{opt}$ }

indicates an optional expression enclosed in braces. The syntax is summarized in §18.

4. What's in a name?

C bases the interpretation of an identifier upon two attributes of the identifier: its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

There are four declarable storage classes: automatic, static, external, and register. Automatic variables are local to each invocation of a block (§9.2), and are discarded upon exit from the block; static variables are local to a block, but retain their values upon reentry to a block even after control has left the block; external variables exist and retain their values throughout the execution of the entire program, and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables they are local to each block and disappear on exit from the block.

C supports several fundamental types of objects:

Objects declared as characters (`char`) are large enough to store any member of the implementation's character set, and if a genuine character from that character set is stored in a character variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine-dependent.

Up to three sizes of integer, declared `short int`, `int`, and `long int`, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers, or long integers, or both, equivalent to plain integers. "Plain" integers have the natural size suggested by the host machine architecture; the other sizes are provided to meet special needs.

Unsigned integers, declared `unsigned`, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. (On the PDP-11, unsigned long quantities are not supported.)

Single-precision floating point (`float`) and double-precision floating point (`double`) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. Types `char` and `int` of all sizes will collectively be called *integral* types. `float` and `double` will collectively be called *floating* types.

Besides the fundamental arithmetic types there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

arrays of objects of most types;
functions which return objects of a given type;
pointers to objects of a given type;
structures containing a sequence of objects of various types;
unions capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

5. Objects and lvalues

An *object* is a manipulatable region of storage; an *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier There are operators which yield lvalues: for example, if E is an expression of pointer type, then *E is an lvalue expression referring to the object to which E points. The name "lvalue" comes from the assignment expression E1 = E2 in which the left operand E1 must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

6. Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions. §6.6 summarizes the conversions demanded by most ordinary operators; it will be supplemented as required by the discussion of each operator.

6.1 Characters and integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer always involves sign extension; integers are signed quantities. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. Of the machines treated by this manual, only the PDP-11 sign-extends. On the PDP-11, character variables range in value from -128 to 127; the characters of the ASCII alphabet are all positive. A character constant specified with an octal escape suffers sign extension and may appear negative; for example, '\377' has the value -1.

When a longer integer is converted to a shorter or to a char, it is truncated on the left; excess bits are simply discarded.

6.2 Float and double

All floating arithmetic in C is carried out in double-precision; whenever a float appears in an expression it is lengthened to double by zero-padding its fraction. When a double must be converted to float, for example by an assignment, the double is rounded before truncation to float length.

6.3 Floating and integral

Conversions of floating values to integral type tend to be rather machine-dependent; in particular the direction of truncation of negative numbers varies from machine to machine. The result is undefined if the value will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of precision occurs if the destination lacks sufficient bits.

6.4 Pointers and integers

An integer or long integer may be added to or subtracted from a pointer; in such a case the first is converted as specified in the discussion of the addition operator.

Two pointers to objects of the same type may be subtracted; in this case the result is converted to an integer as specified in the discussion of the subtraction operator.

6.5 Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo $2^{wordsize}$). In a 2's complement representation, this conversion is conceptual and there is no actual change in the bit pattern.

When an unsigned integer is converted to long, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

6.6 Arithmetic conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions."

First, any operands of type char or short are converted to int, and any of type float are converted to double.

Then, if either operand is `double`, the other is converted to `double` and that is the type of the result.
Otherwise, if either operand is `long`, the other is converted to `long` and that is the type of the result.
Otherwise, if either operand is `unsigned`, the other is converted to `unsigned` and that is the type of the result.
Otherwise, both operands must be `int`, and that is the type of the result.

7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of + (§7.4) are those expressions defined in §§7.1-7.3. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in the grammar of §18.

Otherwise the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects. The order in which side effects take place is unspecified. Expressions involving a commutative and associative operator (*, +, &, |, ^) may be rearranged arbitrarily, even in the presence of parentheses; to force a particular order of evaluation an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is machine-dependent. All existing implementations of C ignore integer overflows; treatment of division by 0, and all floating-point exceptions, varies between machines, and is usually adjustable by a library function.

7.1 Primary expressions

Primary expressions involving `.`, `->`, subscripting, and function calls group left to right.

primary-expression:
- *identifier*
- *constant*
- *string*
- (*expression*)
- *primary-expression* [*expression*]
- *primary-expression* (*expression-list*$_{opt}$)
- *primary-lvalue* . *identifier*
- *primary-expression* -> *identifier*

expression-list:
- *expression*
- *expression-list* , *expression*

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is "array of ...", however, then the value of the identifier-expression is a pointer to the first object in the array, and the type of the expression is "pointer to ...". Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared "function returning ...", when used except in the function-name position of a call, is converted to "pointer to function returning ...".

A constant is a primary expression. Its type may be `int`, `long`, or `double` depending on its form. Character constants have type `int`; floating constants are `double`.

A string is a primary expression. Its type is originally "array of `char`"; but following the same rule given above for identifiers, this is modified to "pointer to `char`" and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see §8.6.)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to ...", the subscript expression is `int`, and the type of the result is "...". The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`. All the clues needed to understand this notation are contained in this section together with the discussions in §§ 7.1, 7.2, and 7.4 on identifiers, *, and + respectively; §14.3 below summarizes the implications.

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning ...", and the result of the function call is of type "...". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type `float` are converted to `double` before the call; any of type `char` or `short` are converted to `int`; and as usual, array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see §7.2, 8.7.

In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. On the other hand, it is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ.

Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be an lvalue naming a structure or a union, and the identifier must name a member of the structure or union. The result is an lvalue referring to the named member of the structure or union.

A primary expression followed by an arrow (built from a - and a >) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points.

Thus the expression `E1->MOS` is the same as `(*E1).MOS`. Structures and unions are discussed in §8.5. The rules given here for the use of structures and unions are not enforced strictly, in order to allow an escape from the typing mechanism. See §14.1.

7.2 Unary operators

Expressions with unary operators group right-to-left.

unary-expression:
  `*` *expression*
  `&` *lvalue*
  `-` *expression*
  `!` *expression*
  `~` *expression*
  `++` *lvalue*
  `--` *lvalue*
  lvalue `++`
  lvalue `--`
  (*type-name*) *expression*
  `sizeof` *expression*
  `sizeof` (*type-name*)

The unary `*` operator means *indirection*: the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to ...", the type of the result is "...".

The result of the unary `&` operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is "...", the type of the result is "pointer to ...".

The result of the unary - operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n, where n is the number of bits in an `int`. There is no unary + operator.

The result of the logical negation operator ! is 1 if the value of its operand is 0, 0 if the value of its operand is non-zero. The type of the result is `int`. It is applicable to any arithmetic type or to pointers.

The ~ operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix ++ is incremented. The value is the new value of the operand, but is not an lvalue. The expression `++x` is equivalent to `x+=1`. See the discussions of addition (§7.4) and assignment operators (§7.14) for information on conversions.

The lvalue operand of prefix -- is decremented analogously to the prefix ++ operator.

When postfix ++ is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix -- is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix -- operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast*. Type names are described in §8.7.

The `sizeof` operator yields the size, in bytes, of its operand. (A *byte* is undefined by the language except in terms of the value of `sizeof`. However, in all existing implementations a byte is the space required to hold a `char`.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an integer constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The `sizeof` operator may also be applied to a parenthesized type name. In that case it yields the size, in bytes, of an object of the indicated type.

The construction `sizeof(`*type*`)` is taken to be a unit, so the expression `sizeof(`*type*`)-2` is the same as `(sizeof(`*type*`))-2`.

7.3 Multiplicative operators

The multiplicative operators `*`, `/`, and `%` group left-to-right. The usual arithmetic conversions are performed.

multiplicative-expression:
  expression `*` *expression*
  expression `/` *expression*
  expression `%` *expression*

The binary `*` operator indicates multiplication. The `*` operator is associative and expressions with several multiplications at the same level may be rearranged by the compiler.

The binary `/` operator indicates division. When positive integers are divided truncation is toward 0, but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that `(a/b)*b + a%b` is equal to `a` (if `b` is not 0).

The binary `%` operator yields the remainder from the division of the first expression by the second. The usual arithmetic conversions are performed. The operands must not be `float`.

7.4 Additive operators

The additive operators `+` and `-` group left-to-right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
  expression `+` *expression*
  expression `-` *expression*

The result of the `+` operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer, and which points to another object in the same array, appropriately offset from the original object. Thus if `P` is a pointer to an object in an array, the expression `P+1` is a pointer to the next object in the array.

No further type combinations are allowed for pointers.

The `+` operator is associative and expressions with several additions at the same level may be rearranged by the compiler.

The result of the `-` operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions as for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an `int` representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object-length.

7.5 Shift operators

The shift operators << and >> group left-to-right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to `int`; the type of the result is that of the left operand. The result is undefined if the right operand is negative, or greater than or equal to the length of the object in bits.

> *shift-expression:*
> > *expression* << *expression*
> > *expression* >> *expression*

The value of `E1<<E2` is `E1` (interpreted as a bit pattern) left-shifted `E2` bits; vacated bits are 0-filled. The value of `E1>>E2` is `E1` right-shifted `E2` bit positions. The right shift is guaranteed to be logical (0-fill) if `E1` is `unsigned`; otherwise it may be (and is, on the PDP-11) arithmetic (fill by a copy of the sign bit).

7.6 Relational operators

The relational operators group left-to-right, but this fact is not very useful; `a<b<c` does not mean what it seems to.

> *relational-expression:*
> > *expression* < *expression*
> > *expression* > *expression*
> > *expression* <= *expression*
> > *expression* >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is `int`. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

7.7 Equality operators

> *equality-expression:*
> > *expression* == *expression*
> > *expression* != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus `a<b == c<d` is 1 whenever `a<b` and `c<d` have the same truth-value).

A pointer may be compared to an integer, but the result is machine dependent unless the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object, and will appear to be equal to 0; in conventional usage, such a pointer is considered to be null.

7.8 Bitwise AND operator

> *and-expression:*
> > *expression* & *expression*

The & operator is associative and expressions involving & may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands. The operator applies only to integral operands.

7.9 Bitwise exclusive OR operator

> *exclusive-or-expression:*
> > *expression* ^ *expression*

The ^ operator is associative and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

7.10 Bitwise inclusive OR operator

inclusive-or-expression:
  expression | *expression*

The | operator is associative and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

7.11 Logical AND operator

logical-and-expression:
  expression && *expression*

The && operator groups left-to-right. It returns 1 if both its operands are non-zero, 0 otherwise. Unlike &, && guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

7.12 Logical OR operator

logical-or-expression:
  expression || *expression*

The || operator groups left-to-right. It returns 1 if either of its operands is non-zero, and 0 otherwise. Unlike |, || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

7.13 Conditional operator

conditional-expression:
  expression ? *expression* : *expression*

Conditional expressions group right-to-left. The first expression is evaluated and if it is non-zero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type; otherwise, if both are pointers of the same type, the result has the common type; otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

7.14 Assignment operators

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:
  lvalue = *expression*
  lvalue += *expression*
  lvalue −= *expression*
  lvalue *= *expression*
  lvalue /= *expression*
  lvalue %= *expression*
  lvalue >>= *expression*
  lvalue <<= *expression*
  lvalue &= *expression*
  lvalue ^= *expression*
  lvalue |= *expression*

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment.

The behavior of an expression of the form `E1` *op*= `E2` may be inferred by taking it as equivalent to `E1 = E1` *op* `(E2)`; however, `E1` is evaluated only once. In `+=` and `-=`, the left operand may be a pointer, in which case the (integral) right operand is converted as explained in §7.4; all right operands and all non-pointer left operands must have arithmetic type.

The compilers currently allow a pointer to be assigned to an integer, an integer to a pointer, and a pointer to a pointer of another type. The assignment is a pure copy operation, with no conversion. This usage is nonportable, and may produce pointers which cause addressing exceptions when used. However, it is guaranteed that assignment of the constant 0 to a pointer will produce a null pointer distinguishable from a pointer to any object.

7.15 Comma operator

comma-expression:
  expression , expression

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right. In contexts where comma is given a special meaning, for example in a list of actual arguments to functions (§7.1) and lists of initializers (§8.6), the comma operator as described in this section can only appear in parentheses; for example,

```
f(a, (t=3, t+2), c)
```

has three arguments, the second of which has the value 5.

8. Declarations

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:
  decl-specifiers declarator-list$_{opt}$;

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

decl-specifiers:
  type-specifier decl-specifiers$_{opt}$
  sc-specifier decl-specifiers$_{opt}$

The list must be self-consistent in a way described below.

8.1 Storage class specifiers

The sc-specifiers are:

sc-specifier:
```
auto
static
extern
register
typedef
```

The `typedef` specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience; it is discussed in §8.8. The meanings of the various storage classes were discussed in §4.

The `auto`, `static` and `register` declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the `extern` case there must be an external definition (§10) for the given identifiers somewhere outside the function in which they are declared.

A `register` declaration is best thought of as an `auto` declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are `int`, `char`, or pointer. One other restriction applies to register variables: the address-of operator `&` cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

At most one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be `auto` inside a function, `extern` outside. Exception: functions are never automatic.

8.2 Type specifiers

The type-specifiers are

type-specifier:
  `char`
  `short`
  `int`
  `long`
  `unsigned`
  `float`
  `double`
  struct-or-union-specifier
  typedef-name

The words `long`, `short`, and `unsigned` may be thought of as adjectives; the following combinations are acceptable.

```
short int
long int
unsigned int
long float
```

The meaning of the last is the same as `double`. Otherwise, at most one type-specifier may be given in a declaration. If the type-specifier is missing from a declaration, it is taken to be `int`.

Specifiers for structures and unions are discussed in §8.5; declarations with `typedef` names are discussed in §8.8.

8.3 Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

declarator-list:
  init-declarator
  init-declarator , *declarator-list*

init-declarator:
  declarator initializer$_{opt}$

Initializers are discussed in §8.6. The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
  identifier
  (*declarator*)
  * *declarator*
  declarator ( )
  declarator [*constant-expression*$_{opt}$]

The grouping is the same as in expressions.

8.4 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class. Each declarator contains exactly one identifier; it is this identifier that is declared.

If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

```
T D1
```

where T is a type-specifier (like int, etc.) and D1 is a declarator. Suppose this declaration makes the identifier have type "... T," where the "..." is empty if D1 is just a plain identifier (so that the type of x in "int x" is just int). Then if D1 has the form

```
*D
```

the type of the contained identifier is "... pointer to T."

If D1 has the form

```
D()
```

then the contained identifier has the type "... function returning T."

If D1 has the form

D[*constant-expression*]

or

```
D[]
```

then the contained identifier has type "... array of T." In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is int. (Constant expressions are defined precisely in §15.) When several "array of" specifications are adjacent, a multi-dimensional array is created; the constant expressions which specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant-expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays, structures, unions or functions, although they may return pointers to such things; there are no arrays of functions, although there may be arrays of pointers to functions. Likewise a structure or union may not contain a function, but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f(), *fip(), (*pfi)();
```

declares an integer i, a pointer ip to an integer, a function f returning an integer, a function fip returning a pointer to an integer, and a pointer pfi to a function which returns an integer. It is especially useful to compare the last two. The binding of *fip() is *(fip()), so that the declaration suggests, and the same construction in an expression requires, the calling of a function fip, and then using indirection through the (pointer) result to yield an integer. In the declarator (*pfi)(), the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

```
float fa[17], *afp[17];
```

declares an array of float numbers and an array of pointers to float numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank 3×5×7. In complete detail, x3d is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions x3d, x3d[i], x3d[i][j], x3d[i][j][k] may reasonably appear in an expression. The first three have type "array," the last has type int.

8.5 Structure and union declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form

> *struct-or-union-specifier:*
>   *struct-or-union* { *struct-decl-list* }
>   *struct-or-union identifier* { *struct-decl-list* }
>   *struct-or-union identifier*
>
> *struct-or-union:*
>   `struct`
>   `union`

The struct-decl-list is a sequence of declarations for the members of the structure or union:

> *struct-decl-list:*
>   *struct-declaration*
>   *struct-declaration struct-decl-list*
>
> *struct-declaration:*
>   *type-specifier struct-declarator-list ;*
>
> *struct-declarator-list:*
>   *struct-declarator*
>   *struct-declarator , struct-declarator-list*

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field*; its length is set off from the field name by a colon.

> *struct-declarator:*
>   *declarator*
>   *declarator : constant-expression*
>   *: constant-expression*

Within a structure, the objects declared have addresses which increase as their declarations are read left-to-right. Each non-field member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word. Fields are assigned right-to-left on the PDP-11, left-to-right on other machines.

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, an unnamed field with a width of 0 specifies alignment of the next field at a word boundary. The "next field" presumably is a field, not an ordinary structure member, because in the latter case the alignment would have been automatic.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even **`int`** fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values. In all implementations, there are no arrays of fields, and the address-of operator & may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

> `struct` *identifier* { *struct-decl-list* }
> `union` *identifier* { *struct-decl-list* }

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

> `struct` *identifier*
> `union` *identifier*

Structure tags allow definition of self-referential structures; they also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union which contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The names of members and tags may be the same as ordinary variables. However, names of tags and members must be mutually distinct.

Two structures may share a common initial sequence of members; that is, the same member may appear in two different structures if it has the same type in both and if all previous members are the same in both. (Actually, the compiler checks only that a name in two different structures has the same type and offset in both, but if preceding members differ the construction is nonportable.)

A simple example of a structure declaration is

```
struct tnode {
     char tword[20];
     int count;
     struct tnode *left;
     struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares s to be a structure of the given sort and sp to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the count field of the structure to which sp points;

```
s.left
```

refers to the left subtree pointer of the structure s; and

```
s.right->tword[0]
```

refers to the first character of the tword member of the right subtree of s.

8.6 Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by =, and consists of an expression or a list of values nested in braces.

initializer:
  = *expression*
  = { *initializer-list* }
  = { *initializer-list* , }

initializer-list:
  expression
  initializer-list , *initializer-list*
  { *initializer-list* }

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in §15, or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants, and previously declared variables and functions.

Static and external variables which are not initialized are guaranteed to start off as 0; automatic and register variables which are not initialized are guaranteed to start off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array) then the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate, written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with 0's. It is not permitted to initialize unions or automatic aggregates.

Braces may be elided as follows. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a **char** array to be initialized by a string. In this case successive characters of the string initialize the members of the array.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes **x** as a 1-dimensional array which has three members, since no size was specified and there are three initializers.

```
float y[4][3] = {
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array y[0], namely y[0][0], y[0][1], and y[0][2]. Likewise the next two lines initialize y[1] and y[2]. The initializer ends early and therefore y[3] is initialized with 0. Precisely the same effect could have been achieved by

```
float y[4][3] = {
        1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for y begins with a left brace, but that for y[0] does not, therefore 3 elements from the list are used. Likewise the next three are taken successively for y[1] and y[2]. Also,

```
float y[4][3] = {
        { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of y (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

8.7 Type names

In two contexts (to specify type conversions explicitly by means of a cast, and as an argument of sizeof) it is desired to supply the name of a data type. This is accomplished using a "type name," which in essence is a declaration for an object of that type which omits the name of the object.

> *type-name:*
> > *type-specifier abstract-declarator*
>
> *abstract-declarator:*
> > *empty*
> > (*abstract-declarator*)
> > * *abstract-declarator*
> > *abstract-declarator* ( )
> > *abstract-declarator* [*constant-expression*$_{opt}$]

To avoid ambiguity, in the construction

> (*abstract-declarator*)

the abstract-declarator is required to be non-empty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
int *()
int (*)()
```

name respectively the types "integer," "pointer to integer," "array of 3 pointers to integers," "pointer to an array of 3 integers," "function returning pointer to integer," and "pointer to function returning an integer."

8.8 Typedef

Declarations whose "storage class" is `typedef` do not define storage, but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

> *typedef-name:*
> *identifier*

Within the scope of a declaration involving `typedef`, each identifier appearing as part of any declarator therein become syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in §8.4. For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im; } complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of `distance` is `int`, that of `metricp` is "pointer to `int`," and that of `z` is the specified structure. `zp` is a pointer to such a structure.

`typedef` does not introduce brand new types, only synonyms for types which could be specified in another way. Thus in the example above `distance` is considered to have exactly the same type as any other `int` object.

9. Statements

Except as indicated, statements are executed in sequence.

9.1 Expression statement

Most statements are expression statements, which have the form

> *expression ;*

Usually expression statements are assignments or function calls.

9.2 Compound statement, or block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided:

> *compound-statement:*
> { *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }
>
> *declaration-list:*
> *declaration*
> *declaration declaration-list*
>
> *statement-list:*
> *statement*
> *statement statement-list*

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of `auto` or `register` variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of `static` variables are performed only once when the program begins execution. Inside a block, `extern` declarations do not reserve storage so initialization is not permitted.

9.3 Conditional statement

The two forms of the conditional statement are

> `if` (*expression*) *statement*
> `if` (*expression*) *statement* `else` *statement*

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the "else" ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if`.

9.4 While statement

The `while` statement has the form

> `while` (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

9.5 Do statement

The `do` statement has the form

> `do` *statement* `while` (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes zero. The test takes place after each execution of the statement.

9.6 For statement

The `for` statement has the form

> `for` (*expression-1*$_{opt}$; *expression-2*$_{opt}$; *expression-3*$_{opt}$) *statement*

This statement is equivalent to

```
expression-1 ;
while (expression-2) {
        statement
        expression-3 ;
}
```

Thus the first expression specifies initialization for the loop, the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression often specifies an incrementation which is performed after each iteration.

Any or all of the expressions may be dropped. A missing *expression-2* makes the implied `while` clause equivalent to `while(1)`; other missing expressions are simply dropped from the expansion above.

9.7 Switch statement

The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

> `switch` (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be `int`. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

> `case` *constant-expression* :

where the constant expression must be `int`. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in §15.

There may also be at most one statement prefix of the form

```
        default :
```

When the `switch` statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression, and if there is a `default` prefix, control passes to the prefixed statement. If no case matches and if there is no `default` then none of the statements in the switch is executed.

`case` and `default` prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see `break`, §9.8.

Usually the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective.

9.8 Break statement

The statement

```
        break ;
```

causes termination of the smallest enclosing `while`, `do`, `for`, or `switch` statement; control passes to the statement following the terminated statement.

9.9 Continue statement

The statement

```
        continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing `while`, `do`, or `for` statement; that is to the end of the loop. More precisely, in each of the statements

```
    while (...) {          do {                  for (...) {
       ...                    ...                   ...
    contin: ;              contin: ;             contin: ;
    }                      } while (...);        }
```

a `continue` is equivalent to `goto contin`. (Following the `contin:` is a null statement, §9.13.)

9.10 Return statement

A function returns to its caller by means of the `return` statement, which has one of the forms

```
        return ;
        return expression ;
```

In the first case the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

9.11 Goto statement

Control may be transferred unconditionally by means of the statement

```
        goto identifier ;
```

The identifier must be a label (§9.12) located in the current function.

9.12 Labeled statement

Any statement may be preceded by label prefixes of the form

```
        identifier :
```

which serve to declare the identifier as a label. The only use of a label is as a target of a `goto`. The scope of a label is the current function, excluding any sub-blocks in which the same identifier has been redeclared. See §11.

9.13 Null statement

The null statement has the form

```
;
```

A null statement is useful to carry a label just before the } of a compound statement or to supply a null body to a looping statement such as `while`.

10. External definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class `extern` (by default) or perhaps `static`, and a specified type. The type-specifier (§8.2) may also be empty, in which case the type is taken to be `int`. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations, except that only at this level may the code for functions be given.

10.1 External function definitions

Function definitions have the form

function-definition:
  decl-specifiers$_{opt}$ *function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are `extern` or `static`; see §11.2 for the distinction between them. A function declarator is similar to a declarator for a "function returning ..." except that it lists the formal parameters of the function being defined.

function-declarator:
  declarator (parameter-list$_{opt}$ *)*

parameter-list:
  identifier
  identifier , parameter-list

The function-body has the form

function-body:
  declaration-list compound-statement

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be `int`. The only storage class which may be specified is `register`; if it is specified the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
int a, b, c;
{
	int m;

	m = (a > b) ? a : b;
	return((m > c) ? m : c);
}
```

Here `int` is the type-specifier; `max(a, b, c)` is the function-declarator; `int a, b, c;` is the declaration-list for the formal parameters; { . . . } is the block giving the code for the statement.

C converts all `float` actual parameters to `double`, so formal parameters declared `float` have their declaration adjusted to read `double`. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared "array of ..." are adjusted to read "pointer to ...". Finally, because structures, unions and functions cannot be passed to a function, it is useless to declare a formal parameter to be a structure, union or function (pointers to such objects are of course permitted).

10.2 External data definitions

An external data definition has the form

> *data-definition:*
>
> *declaration*

The storage class of such data may be `extern` (which is the default) or `static`, but not `auto` or `register`.

11. Scope rules

A C program need not all be compiled at the same time: the source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, what may be called the *lexical scope* of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

11.1 Lexical scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of blocks persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

Because all references to the same external identifier refer to the same object (see §11.2) the compiler checks all declarations of the same external identifier for compatibility; in effect their scope is increased to the whole file in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (§8.5) that identifiers associated with ordinary variables on the one hand and those associated with structure and union members and tags on the other form two disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. `typedef` names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
      auto int distance;
      ...
```

The `int` must be present in the second declaration, or it would be taken to be a declaration with no declarators and type `distance`†.

11.2 Scope of externals

If a function refers to an identifier declared to be `extern`, then somewhere among the files or libraries constituting the complete program there must be an external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function which references the data.

The appearance of the `extern` keyword in an external definition indicates that storage for the identifiers being declared will be allocated in another file. Thus in a multi-file program, an external data definition without the `extern` specifier must appear in exactly one of the files. Any other files which wish to give an external definition for the identifier must include the `extern` in the definition. The identifier can be initialized only in the declaration where storage is allocated.

Identifiers declared `static` at the top level in external definitions are not visible in other files. Functions may be declared `static`.

†It is agreed that the ice is thin here.

12. Compiler control lines

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with # communicate with this preprocessor. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

12.1 Token replacement

A compiler-control line of the form

`#define` *identifier token-string*

(note: no trailing semicolon) causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. A line of the form

`#define` *identifier*(*identifier* , ... , *identifier*) *token-string*

where there is no space between the first identifier and the (, is a macro definition with arguments. Subsequent instances of the first identifier followed by a (, a sequence of tokens delimited by commas, and a) are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Text inside a string or a character constant is not subject to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued.

This facility is most valuable for definition of "manifest constants," as in

```
#define TABSIZE 100

int table[TABSIZE];
```

A control line of the form

`#undef` *identifier*

causes the identifier's preprocessor definition to be forgotten.

12.2 File inclusion

A compiler control line of the form

`#include "`*filename*`"`

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the original source file, and then in a sequence of standard places. Alternatively, a control line of the form

`#include <`*filename*`>`

searches only the standard places, and not the directory of the source file.

`#include`'s may be nested.

12.3 Conditional compilation

A compiler control line of the form

`#if` *constant-expression*

checks whether the constant expression (see §15) evaluates to non-zero. A control line of the form

`#ifdef` *identifier*

checks whether the identifier is currently defined in the preprocessor; that is, whether it has been the subject of a `#define` control line. A control line of the form

`#ifndef` *identifier*

checks whether the identifier is currently undefined in the preprocessor.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

```
        #else
```

and then by a control line

```
        #endif
```

If the checked condition is true then any lines between `#else` and `#endif` are ignored. If the checked condition is false then any lines between the test and an `#else` or, lacking an `#else`, the `#endif`, are ignored.

These constructions may be nested.

12.4 Line control

For the benefit of other preprocessors which generate C programs, a line of the form

`#line` *constant identifier*

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by the identifier. If the identifier is absent the remembered file name does not change.

13. Implicit declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be `int`; if a type but no storage class is indicated, the identifier is assumed to be `auto`. An exception to the latter rule is made for functions, since `auto` functions are meaningless (C being incapable of compiling code into the stack); if the type of an identifier is "function returning ...", it is implicitly declared to be `extern`.

In an expression, an identifier followed by (and not already declared is contextually declared to be "function returning `int`".

14. Types revisited

This section summarizes the operations which can be performed on objects of certain types.

14.1 Structures and unions

There are only two things that can be done with a structure or union: name one of its members (by means of the `.` operator); or take its address (by unary `&`). Other operations, such as assigning from or to it or passing it as a parameter, draw an error message. In the future, it is expected that these operations, but not necessarily others, will be allowed.

§7.1 says that in a direct or indirect structure reference (with `.` or `->`) the name on the right must be a member of the structure named or pointed to by the expression on the left. To allow an escape from the typing rules, this restriction is not firmly enforced by the compiler. In fact, any lvalue is allowed before `.`, and that lvalue is then assumed to have the form of the structure of which the name on the right is a member. Also, the expression before a `->` is required only to be a pointer or an integer. If a pointer, it is assumed to point to a structure of which the name on the right is a member. If an integer, it is taken to be the absolute address, in machine storage units, of the appropriate structure.

Such constructions are non-portable.

14.2 Functions

There are only two things that can be done with a function: call it, or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
        int f();
        ...
        g(f);
```

Then the definition of *g* might read

```
g(funcp)
int (*funcp)();
{
        ...
        (*funcp)();
        ...
}
```

Notice that f must be declared explicitly in the calling routine since its appearance in g(f) was not followed by (.

14.3 Arrays, pointers, and subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator [] is interpreted in such a way that E1[E2] is identical to *((E1)+(E2)). Because of the conversion rules which apply to +, if E1 is an array and E2 an integer, then E1[E2] refers to the E2-th member of E1. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multi-dimensional arrays. If E is an n-dimensional array of rank $i \times j \times \cdots \times k$, then E appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \cdots \times k$. If the * operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to $(n-1)$-dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here x is a 3×5 array of integers. When x appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression x[i], which is equivalent to *(x+i), x is first converted to a pointer as described; then i is converted to the type of x, which involves multiplying i by the length the object to which the pointer points, namely 5 integer objects. The results are added and indirection applied to yield an array (of 5 integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer.

It follows from all this that arrays in C are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

14.4 Explicit pointer conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, §§7.2 and 8.7.

A pointer may be converted to any of the integral types large enough to hold it. Whether an int or long is required is machine dependent. The mapping function is also machine dependent, but is intended to be unsurprising to those who know the addressing structure of the machine. Details for some particular machines are given below.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer, but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a char pointer; it might be used in this way.

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

alloc must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to double; then the *use* of the function is portable.

The pointer representation on the PDP-11 corresponds to a 16-bit integer and is measured in bytes. `char`s have no alignment requirements; everything else must have an even address.

On the Honeywell 6000, a pointer corresponds to a 36-bit integer; the word part is in the left 18 bits, and the two bits that select the character in a word just to their right. Thus `char` pointers are measured in units of 2^{16} bytes; everything else is measured in units of 2^{18} machine words. `double` quantities and aggregates containing them must lie on an even word address (0 mod 2^{19}).

The IBM 370 and the Interdata 8/32 are similar. On both, addresses are measured in bytes; elementary objects must be aligned on a boundary equal to their length, so pointers to `short` must be 0 mod 2, to `int` and `float` 0 mod 4, and to `double` 0 mod 8. Aggregates are aligned on the strictest boundary required by any of their constituents.

15. Constant expressions

In several places C requires expressions which evaluate to a constant: after `case`, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, and `sizeof` expressions, possibly connected by the binary operators

```
+   -   *   /   %   &   |   ^   <<   >>   ==   !=   <   >   <=   >=
```

or by the unary operators

```
-   ~
```

or by the ternary operator

```
?:
```

Parentheses can be used for grouping, but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also apply the unary `&` operator to external or static objects, and to external or static arrays subscripted with a constant expression. The unary `&` can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

16. Portability considerations

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive, but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are a nuisance that must be carefully watched. Most of the others are only minor problems.

The number of `register` variables that can actually be placed in registers varies from machine to machine, as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid `register` declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language. It is right to left on the PDP-11, and VAX-11, left to right on the others. The order in which side effects take place is also unspecified.

Since character constants are really objects of type `int`, multi-character character constants may be permitted. The specific implementation is very machine dependent, however, because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right-to-left on the PDP-11 and VAX-11 and left-to-right on other machines. These differences are invisible to isolated programs which do not indulge in type punning (for example, by converting an `int` pointer to a `char` pointer and inspecting the pointed-to storage), but must be accounted for when conforming to externally-imposed storage layouts.

The language accepted by the various compilers differs in minor details. Most notably, the current PDP-11 compiler will not initialize structures containing bit-fields, and does not accept a few assignment operators in certain contexts where the value of the assignment is used.

17. Anachronisms

Since C is an evolving language, certain obsolete constructions may be found in older programs. Although most versions of the compiler support such anachronisms, ultimately they will disappear, leaving only a portability problem behind.

Earlier versions of C used the form *=op* instead of *op=* for assignment operators. This leads to ambiguities, typified by

```
    x=-1
```

which actually decrements x since the = and the - are adjacent, but which might easily be intended to assign -1 to x.

The syntax of initializers has changed: previously, the equals sign that introduces an initializer was not present, so instead of

```
    int  x    = 1;
```

one used

```
    int  x    1;
```

The change was made because the initialization

```
    int  f    (1+2)
```

resembles a function declaration closely enough to confuse the compilers.

18. Syntax Summary

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

18.1 Expressions

The basic expressions are:

```
expression:
        primary
        * expression
        & expression
        - expression
        ! expression
        ~ expression
        ++ lvalue
        -- lvalue
        lvalue ++
        lvalue --
        sizeof expression
        ( type-name ) expression
        expression binop expression
        expression ? expression : expression
        lvalue asgnop expression
        expression , expression

primary:
        identifier
        constant
        string
        ( expression )
        primary ( expression-list_opt )
        primary [ expression ]
        lvalue . identifier
        primary -> identifier

lvalue:
        identifier
        primary [ expression ]
        lvalue . identifier
        primary -> identifier
        * expression
        ( lvalue )
```

The primary-expression operators

```
() [] . ->
```

have highest priority and group left-to-right. The unary operators

```
* & - ! ~ ++ -- sizeof ( type-name )
```

have priority below the primary operators but higher than any binary operator, and group right-to-left. Binary operators group left-to-right; they have priority decreasing as indicated below. The conditional operator groups right to left.

binop:

```
        *     /     %
        +     -
        >>    <<
        <     >     <=     >=
        ==    !=
        &
        ^
        |
        &&
        ||
        ?:
```

Assignment operators all have the same priority, and all group right-to-left.

asgnop:

```
        =   +=   -=   *=   /=   %=   >>=   <<=   &=   ^=   |=
```

The comma operator has the lowest priority, and groups left-to-right.

18.2 Declarations

declaration:
  decl-specifiers init-declarator-list$_{opt}$;

decl-specifiers:
  type-specifier decl-specifiers$_{opt}$
  sc-specifier decl-specifiers$_{opt}$

sc-specifier:
  `auto`
  `static`
  `extern`
  `register`
  `typedef`

type-specifier:
  `char`
  `short`
  `int`
  `long`
  `unsigned`
  `float`
  `double`
  struct-or-union-specifier
  typedef-name

init-declarator-list:
  init-declarator
  init-declarator , *init-declarator-list*

init-declarator:
  declarator initializer$_{opt}$

declarator:
  identifier
  (*declarator*)
  * *declarator*
  declarator ( )
  declarator [*constant-expression*$_{opt}$]

struct-or-union-specifier:
  `struct` { *struct-decl-list* }
  `struct` *identifier* { *struct-decl-list* }
  `struct` *identifier*
  `union` { *struct-decl-list* }
  `union` *identifier* { *struct-decl-list* }
  `union` *identifier*

struct-decl-list:
  struct-declaration
  struct-declaration struct-decl-list

struct-declaration:
  type-specifier struct-declarator-list ;

struct-declarator-list:
  struct-declarator
  struct-declarator , *struct-declarator-list*

struct-declarator:
  declarator
  declarator : *constant-expression*
  : *constant-expression*

initializer:
  = *expression*
  = { *initializer-list* }
  = { *initializer-list* , }

initializer-list:
  expression
  initializer-list , *initializer-list*
  { *initializer-list* }

type-name:
  type-specifier abstract-declarator

abstract-declarator:
  empty
  (*abstract-declarator*)
  * *abstract-declarator*
  abstract-declarator ( )
  abstract-declarator [*constant-expression*$_{opt}$]

typedef-name:
  identifier

18.3 Statements

compound-statement:
  { *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
  declaration
  declaration declaration-list

statement-list:
statement
statement statement-list

statement:
compound-statement
expression ;
`if` (*expression*) *statement*
`if` (*expression*) *statement* `else` *statement*
`while` (*expression*) *statement*
`do` *statement* `while` (*expression*) ;
`for` (*expression-1*$_{opt}$; *expression-2*$_{opt}$; *expression-3*$_{opt}$) *statement*
`switch` (*expression*) *statement*
`case` *constant-expression* : *statement*
`default` : *statement*
`break` ;
`continue` ;
`return` ;
`return` *expression* ;
`goto` *identifier* ;
identifier : *statement*
;

18.4 External definitions

program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition:
type-specifier$_{opt}$ *function-declarator function-body*

function-declarator:
declarator (*parameter-list*$_{opt}$)

parameter-list:
identifier
identifier , *parameter-list*

function-body:
type-decl-list function-statement

function-statement:
{ *declaration-list*$_{opt}$ *statement-list* }

data-definition:
`extern`$_{opt}$ *type-specifier*$_{opt}$ *init-declarator-list*$_{opt}$;
`static`$_{opt}$ *type-specifier*$_{opt}$ *init-declarator-list*$_{opt}$;

18.5 Preprocessor

#define *identifier token-string*
#define *identifier*(*identifier* , ... , *identifier*) *token-string*
#undef *identifier*
#include "*filename*"
#include <*filename*>
#if *constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#else
#endif
#line *constant identifier*

Recent Changes to C

November 15, 1978

A few extensions have been made to the C language beyond what is described in the reference document ("The C Programming Language," Kernighan and Ritchie, Prentice-Hall, 1978).

1. Structure assignment

Structures may be assigned, passed as arguments to functions, and returned by functions. The types of operands taking part must be the same. Other plausible operators, such as equality comparison, have not been implemented.

There is a subtle defect in the PDP-11 implementation of functions that return structures: if an interrupt occurs during the return sequence, and the same function is called reentrantly during the interrupt, the value returned from the first call may be corrupted. The problem can occur only in the presence of true interrupts, as in an operating system or a user program that makes significant use of signals; ordinary recursive calls are quite safe.

2. Enumeration type

There is a new data type analogous to the scalar types of Pascal. To the type-specifiers in the syntax on p. 193 of the C book add

enum-specifier

with syntax

enum-specifier:
  enum { *enum-list* }
  enum *identifier* { *enum-list* }
  enum *identifier*

enum-list:
  enumerator
  enum-list , *enumerator*

enumerator:
  identifier
  identifier = *constant-expression*

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier: it names a particular enumeration. For example,

```
enum color { chartreuse, burgundy, claret, winedark };
...
enum color *cp, col;
```

makes `color` the enumeration-tag of a type describing various colors, and then declares `cp` as a pointer to an object of that type, and `col` as an object of that type.

The identifiers in the enum-list are declared as constants, and may appear wherever constants are required. If no enumerators with = appear, then the values of the constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

Enumeration tags and constants must all be distinct, and, unlike structure tags and members, are drawn from the same set as ordinary identifiers.

Objects of a given enumeration type are regarded as having a type distinct from objects of all other types, and *lint* flags type mismatches. In the PDP-11 implementation all enumeration variables are treated as if they were `int`.