DO NOT USE THIS SOFTWARE UNTIL YOU HAVE READ THIS LICENSE AGREEMENT. BY USING THE SOFTWARE (OR AUTHORIZING ANY OTHER PERSON TO DO SO), YOU AGREE TO ACCEPT AND ABIDE BY ALL THE TERMS OF THIS LICENSE AGREEMENT. THE LICENSE AGREEMENT REFERS TO YOU AS "CUSTOMER".

1 DEFINITIONS

LICENSED SOFTWARE shall mean solely the object code of the following versions of SCO's UNIX operating system software: Versions V5, V6, and V7 adapted for PDP-11 computer systems.

SCO INTELLECTUAL PROPERTY RIGHTS shall mean patent, copyright and trade secret rights of The Santa Cruz Operation, Inc. ("SCO") in the LICENSED SOFTWARE.

2 LICENSE GRANT

SCO grants to CUSTOMER a worldwide, non-exclusive, royalty-free license under SCO INTELLECTUAL PROPERTY RIGHTS to reproduce, modify, and use the LICENSED SOFTWARE solely for non-commercial uses, and to distribute the LICENSED SOFTWARE to a third party who is also bound by the terms and conditions of this License Agreement. For this purpose, any distribution of a copy of LICENSED SOFTWARE shall be accompanied by a verbatim copy of this License Agreement. CUSTOMER shall not reverse compile, reverse assemble, or otherwise reverse engineer the LICENSED SOFTWARE or any portion thereof, or remove or alter any copyright or other proprietary notices appearing in the LICENSED SOFTWARE.

3 "AS IS" DELIVERY

CUSTOMER acknowledges that it accepts the LICENSED SOFTWARE "AS IS", and that SCO is under no obligation to supply any support, training, consulting, or other services for the LICENSED SOFTWARE.

4 WARRANTY DISCLAIMER/LIMITATION OF LIABILITY

SCO DISCLAIMS ALL WARRANTIES WITH REGARD TO ANY PORTION OF THE LICENSED SOFTWARE, INCLUDING BUT NOT LIMITED TO ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, AND ANY WARRANTY THAT THE LICENSED SOFTWARE DOES NOT VIOLATE ANY PROPRIETARY RIGHT OF A THIRD PARTY. IN NO EVENT SHALL SCO BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE, OR ANY OTHER THEORY ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF ANY PORTION OF THE LICENSED SOFTWARE.

5 INDEMNITY FROM CUSTOMER

CUSTOMER will hold SCO harmless against all liabilities, demands, damages, expenses, or losses arising out of use by CUSTOMER of LICENSED SOFTWARE.

6 TERMINATION

6.1 This Agreement shall be effective unless and until terminated.

6.2 Either party may terminate this Agreement at any time upon 30 days written notice. In addition, if CUSTOMER shall fail to perform or observe any of the terms and conditions to be performed or observed by it under this License Agreement, SCO may in its sole discretion thereafter elect, by written notice, to

immediately terminate this License Agreement, together with all rights granted herein to CUSTOMER.

6.3 Termination of this Agreement shall not release CUSTOMER from any liability which shall have accrued in favor of SCO at the time such termination becomes effective, nor from CUSTOMER's obligation not to reverse engineer any portion of the LICENSED SOFTWARE.

6.4 In the event of any termination of this Agreement for any reason, CUSTOMER shall immediately destroy all whole or partial copies of the LICENSED SOFTWARE in CUSTOMER's possession, custody, or control. In addition, CUSTOMER shall cease to use the LICENSED SOFTWARE as of the date of termination.

7 GENERAL TERMS

7.1 This License Agreement shall be governed by the substantive laws of the State of California.

7.2 CUSTOMER shall not assign any rights or delegate any of its obligations under this License Agreement without the written consent of SCO.

7.3 CUSTOMER assures SCO that, with respect to any LICENSED SOFTWARE obtained under this Agreement, CUSTOMER will comply with all applicable governmental export laws and regulations.

7.4 Waiver of any breach hereunder by CUSTOMER may be effected only by a writing signed by SCO and shall not constitute a waiver of any other breach.

7.5 CUSTOMER understands and agrees that this License Agreement is the complete and exclusive statement of the understanding between the parties with respect to its subject matter, and supersedes all communications and understanding between the parties relating to such subject matter.

-

UNIX PROGRAMMER'S MANUAL

Sixth Edition

K. Thompson

D. M. Ritchie

May, 1975

This manual was set by a Graphic Systems phototypesetter driven by the *troff* formatting program operating under the UNIX system. The text of the manual was prepared using the *ed* text editor.

-

PREFACE
to the Sixth Edition

We are grateful to L. L. Cherry, R. C. Haight, S. C. Johnson, B. W. Kernighan, M. E. Lesk, and E. N. Pinson for their contributions to the system software, and to L. E. McMahon for software and for his contributions to this manual. We are particularly appreciative of the invaluable technical, editorial, and administrative efforts of J. F. Ossanna, M. D. McIlroy, and R. Morris. They all contributed greatly to the stock of UNIX software and to this manual. Their inventiveness, thoughtful criticism, and ungrudging support increased immeasurably not only whatever success the UNIX system enjoys, but also our own enjoyment in its creation.

-

INTRODUCTION TO THIS MANUAL

This manual gives descriptions of the publicly available features of UNIX. It provides neither a general overview − see "The UNIX Time-sharing System" (Comm. ACM **17** 7, July 1974, pp. 365-375) for that − nor details of the implementation of the system, which remain to be disclosed.

Within the area it surveys, this manual attempts to be as complete and timely as possible. A conscious decision was made to describe each program in exactly the state it was in at the time its manual section was prepared. In particular, the desire to describe something as it should be, not as it is, was resisted. Inevitably, this means that many sections will soon be out of date.

This manual is divided into eight sections:

I. Commands
II. System calls
III. Subroutines
IV. Special files
V. File formats and conventions
VI. User-maintained programs
VII. User-maintained subroutines
VIII. Maintenance

Commands are programs intended to be invoked directly by the user, in contradistinction to subroutines, which are intended to be called by the user's programs. Commands generally reside in directory */bin* (for *bin*ary programs). Some programs also reside in */usr/bin,* to save space in */bin.* These directories are searched automatically by the command interpreter.

System calls are entries into the UNIX supervisor. In assembly language, they are coded with the use of the opcode *sys*, a synonym for the *trap* instruction. In this edition, the C language interface routines to the system calls have been incorporated in section II.

A small assortment of subroutines is available; they are described in section III. The binary form of most of them is kept in the system library */lib/liba.a.* The subroutines available from C and from Fortran are also included; they reside in */lib/libc.a* and */lib/libf.a* respectively.

The special files section IV discusses the characteristics of each system "file" which actually refers to an I/O device. The names in this section refer to the DEC device names for the hardware, instead of the names of the special files themselves.

The file formats and conventions section V documents the structure of particular kinds of files; for example, the form of the output of the loader and assembler is given. Excluded are files used by only one command, for example the assembler's intermediate files.

User-maintained programs and subroutines (sections VI and VII) are not considered part of the UNIX system, and the principal reason for listing them is to indicate their existence without necessarily giving a complete description. The authors of the individual programs should be consulted for more information.

Section VIII discusses commands which are not intended for use by the ordinary user, in some cases because they disclose information in which he is presumably not interested, and in others because they perform privileged functions.

Each section consists of a number of independent entries of a page or so each. The name of the entry is in the upper corners of its pages, its preparation date in the upper middle. Entries within each section are alphabetized. The page numbers of each entry start at 1. (The earlier hope for frequent, partial updates of the manual is clearly in vain, but in any event it is not feasible to maintain consecutive page numbering in a document like this.)

-

All entries are based on a common format, not all of whose subsections will always appear.

The *name* section repeats the entry name and gives a very short description of its purpose.

The *synopsis* summarizes the use of the program being described. A few conventions are used, particularly in the Commands section:

Boldface words are considered literals, and are typed just as they appear.

Square brackets ([ ]) around an argument indicate that the argument is optional. When an argument is given as ''name'', it always refers to a file name.

Ellipses ''. . .'' are used to show that the previous argument-prototype may be repeated.

A final convention is used by the commands themselves. An argument beginning with a minus sign ''−'' is often taken to mean some sort of flag argument even if it appears in a position where a file name could appear. Therefore, it is unwise to have files whose names begin with ''−''.

The *description* section discusses in detail the subject at hand.

The *files* section gives the names of files which are built into the program.

A *see also* section gives pointers to related information.

A *diagnostics* section discusses the diagnostic indications which may be produced. Messages which are intended to be self-explanatory are not listed.

The *bugs* section gives known bugs and sometimes deficiencies. Occasionally also the suggested fix is described.

At the beginning of this document is a table of contents, organized by section and alphabetically within each section. There is also a permuted index derived from the table of contents. Within each index entry, the title of the writeup to which it refers is followed by the appropriate section number in parentheses. This fact is important because there is considerable name duplication among the sections, arising principally from commands which exist only to exercise a particular system call.

This manual was prepared using the UNIX text editor *ed* and the formatting program *troff*.

-

HOW TO GET STARTED

This section provides the basic information you need to get started on UNIX: how to log in and log out, how to communicate through your terminal, and how to run a program. See ''UNIX for Beginners'' by Brian W. Kernighan for a more complete introduction to the system.

Logging in. You must call UNIX from an appropriate terminal. UNIX supports ASCII terminals typified by the TTY 37, the GE Terminet 300, the Dasi 300, and various graphical terminals. You must also have a valid user name, which may be obtained, together with the telephone number, from the system administrators. The same telephone number serves terminals operating at all the standard speeds. After a data connection is established, the login procedure depends on what kind of terminal you are using.

> *300-baud terminals:* Such terminals include the GE Terminet 300, most display terminals, Execuport, TI, GSI, and certain Anderson-Jacobson terminals. These terminals generally have a speed switch which should be set at ''300'' (or ''30'' for 30 characters per second) and a half/full duplex switch which should be set at full-duplex. (This switch will often have to be changed since many other systems require half-duplex). When a connection is established, the system types ''login:''; you type your user name, followed by the ''return'' key. If you have a password, the system asks for it and turns off the printer on the terminal so the password will not appear. After you have logged in, the ''return'', ''new line'', or ''linefeed'' keys will give exactly the same results.
>
> *TTY 37 terminal:* When you have established a data connection, the system types out a few garbage characters (the ''login:'' message at the wrong speed). Depress the ''break'' (or ''interrupt'') key; this is a speed-independent signal to UNIX that a 150-baud terminal is in use. The system then will type ''login:,'' this time at the correct speed; you respond with your user name. From the TTY 37 terminal, and any other which has the ''new-line'' function (combined carriage return and linefeed), terminate each line you type with the ''new-line'' key (*not* the ''return'' key).

For all these terminals, it is important that you type your name in lower-case if possible; if you type upper-case letters, UNIX will assume that your terminal cannot generate lower-case letters and will translate all subsequent upper-case letters to lower case.

The evidence that you have successfully logged in is that the Shell program will type a ''%'' to you. (The Shell is described below under ''How to run a program.'')

For more information, consult *getty* (VIII), which discusses the login sequence in more detail, and *tty* (IV), which discusses typewriter I/O.

Logging out. There are three ways to log out:

> You can simply hang up the phone.
>
> You can log out by typing an end-of-file indication (EOT character, control ''d'') to the Shell. The Shell will terminate and the ''login: '' message will appear again.
>
> You can also log in directly as another user by giving a *login* command (I).

How to communicate through your terminal. When you type to UNIX, a gnome deep in the system is gathering your characters and saving them in a secret place. The characters will not be given to a program until you type a return (or new-line), as described above in *Logging in.*

UNIX typewriter I/O is full-duplex. It has full read-ahead, which means that you can type at any time, even while a program is typing at you. Of course, if you type during output, the output will have the input characters interspersed. However, whatever you type will be saved up and interpreted in correct sequence. There is a limit to the amount of read-ahead, but it is generous and not likely to be exceeded unless the system is in trouble. When the read-ahead limit is exceeded, the system throws away all the saved characters.

On a typewriter input line, the character ''@'' kills all the characters typed before it, so typing mistakes can be repaired on a single line. Also, the character ''#'' erases the last character typed. Successive uses of

-

"#" erase characters back to, but not beyond, the beginning of the line. "@" and "#" can be transmitted to a program by preceding them with "\". (So, to erase "\", you need two "#"s).

The ASCII "delete" (a.k.a. "rubout") character is not passed to programs but instead generates an *interrupt signal.* This signal generally causes whatever program you are running to terminate. It is typically used to stop a long printout that you don't want. However, programs can arrange either to ignore this signal altogether, or to be notified when it happens (instead of being terminated). The editor, for example, catches interrupts and stops what it is doing, instead of terminating, so that an interrupt can be used to halt an editor printout without losing the file being edited.

The *quit* signal is generated by typing the ASCII FS character. It not only causes a running program to terminate but also generates a file with the core image of the terminated process. Quit is useful for debugging.

Besides adapting to the speed of the terminal, UNIX tries to be intelligent about whether you have a terminal with the new-line function or whether it must be simulated with carriage-return and line-feed. In the latter case, all input carriage returns are turned to new-line characters (the standard line delimiter) and both a carriage return and a line feed are echoed to the terminal. If you get into the wrong mode, the *stty* command (I) will rescue you.

Tab characters are used freely in UNIX source programs. If your terminal does not have the tab function, you can arrange to have them turned into spaces during output, and echoed as spaces during input. The system assumes that tabs are set every eight columns. Again, the *stty* command (I) will set or reset this mode. Also, there is a file which, if printed on TTY 37 or TermiNet 300 terminals, will set the tab stops correctly (*tabs* (V)).

Section *tty* (IV) discusses typewriter I/O more fully.

How to run a program; the Shell. When you have successfully logged into UNIX, a program called the Shell is listening to your terminal. The Shell reads typed-in lines, splits them up into a command name and arguments, and executes the command. A command is simply an executable program. The Shell looks first in your current directory (see next section) for a program with the given name, and if none is there, then in a system directory. There is nothing special about system-provided commands except that they are kept in a directory where the Shell can find them.

The command name is always the first word on an input line; it and its arguments are separated from one another by spaces.

When a program terminates, the Shell will ordinarily regain control and type a "%" at you to indicate that it is ready for another command.

The Shell has many other capabilities, which are described in detail in section *sh* (I).

The current directory. UNIX has a file system arranged in a hierarchy of directories. When the system administrator gave you a user name, he also created a directory for you (ordinarily with the same name as your user name). When you log in, any file name you type is by default in this directory. Since you are the owner of this directory, you have full permissions to read, write, alter, or destroy its contents. Permissions to have your will with other directories and files will have been granted or denied to you by their owners. As a matter of observed fact, few UNIX users protect their files from destruction, let alone perusal, by other users.

To change the current directory (but not the set of permissions you were endowed with at login) use *chdir* (I).

Path names. To refer to files not in the current directory, you must use a path name. Full path names begin with "/", the name of the root directory of the whole file system. After the slash comes the name of each directory containing the next sub-directory (followed by a "/") until finally the file name is reached. E.g.: */usr/lem/filex* refers to the file *filex* in the directory *lem; lem* is itself a subdirectory of *usr; usr* springs directly from the root directory.

If your current directory has subdirectories, the path names of files therein begin with the name of the sub-

directory (no prefixed "/").

Without important exception, a path name may be used anywhere a file name is required.

Important commands which modify the contents of files are *cp* (I), *mv* (I), and *rm* (I), which respectively copy, move (i.e. rename) and remove files. To find out the status of files or directories, use *ls* (I). See *mkdir* (I) for making directories; *rmdir* (I) for destroying them.

For a fuller discussion of the file system, see "The UNIX Time-Sharing System," by the present authors. It may also be useful to glance through section II of this manual, which discusses system calls, even if you don't intend to deal with the system at that level.

Writing a program. To enter the text of a source program into a UNIX file, use *ed* (I). The three principal languages in UNIX are assembly language (see *as* (I)), Fortran (see *fc* (I)), and C (see *cc* (I)). After the program text has been entered through the editor and written on a file, you can give the file to the appropriate language processor as an argument. The output of the language processor will be left on a file in the current directory named "a.out". (If the output is precious, use *mv* to move it to a less exposed name soon.) If you wrote in assembly language, you will probably need to load the program with library subroutines; see *ld* (I). The other two language processors call the loader automatically.

When you have finally gone through this entire process without provoking any diagnostics, the resulting program can be run by giving its name to the Shell in response to the "%" prompt.

Next, you will need *cdb* (I) or *db* (I) to examine the remains of your program. The former is useful for C programs, the latter for assembly-language. No debugger is much help for Fortran.

Your programs can receive arguments from the command line just as system programs do. See *exec* (II).

Text processing. Almost all text is entered through the editor. The commands most often used to write text on a terminal are: *cat, pr, roff, nroff,* and *troff,* all in section I.

The *cat* command simply dumps ASCII text on the terminal, with no processing at all. The *pr* command paginates the text, supplies headings, and has a facility for multi-column output. *Troff* and *nroff* are elaborate text formatting programs, and require careful forethought in entering both the text and the formatting commands into the input file. *Troff* drives a Graphic Systems phototypesetter; it was used to produce this manual. *Nroff* produces output on a typewriter terminal. *Roff* (I) is a somewhat less elaborate text formatting program, and requires somewhat less forethought.

Surprises. Certain commands provide inter-user communication. Even if you do not plan to use them, it would be well to learn something about them, because someone else may aim them at you.

To communicate with another user currently logged in, *write* (I) is used; *mail* (I) will leave a message whose presence will be announced to another user when he next logs in. The write-ups in the manual also suggest how to respond to the two commands if you are a target.

When you log in, a message-of-the-day may greet you before the first "%".

NAME

ar – archive and library maintainer

SYNOPSIS

ar key afile name ...

DESCRIPTION

Ar maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the loader. It can be used, though, for any similar purpose.

Key is one character from the set **drtux,** optionally concatenated with **v.** *Afile* is the archive file. The *names* are constituent files in the archive file. The meanings of the *key* characters are:

d means delete the named files from the archive file.

r means replace the named files in the archive file. If the archive file does not exist, **r** creates it. If the named files are not in the archive file, they are appended.

t prints a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.

u is similar to **r** except that only those files that have been modified are replaced. If no names are given, all files in the archive that have been modified are replaced by the modified version.

x extracts the named files. If no names are given, all files in the archive are extracted. In neither case does **x** alter the archive file.

v means verbose. Under the verbose option, *ar* gives a file-by-file description of the making of a new archive file from the old archive and the constituent files. The following abbreviations are used:

c copy
a append
d delete
r replace
x extract

FILES

/tmp/vtm? temporary

SEE ALSO

ld (I), archive (V)

BUGS

Option **tv** should be implemented as a table with more information.

There should be a way to specify the placement of a new file in an archive. Currently, it is placed at the end.

Since *ar* has not been rewritten to deal properly with the new file system modes, extracted files have mode 666.

For the same reason, only the first 8 characters of file names are significant.

If the same file is mentioned twice in an argument list, it may be put in the archive twice.

NAME

as – assembler

SYNOPSIS

as [–] name ...

DESCRIPTION

As assembles the concatenation of the named files. If the optional first argument – is used, all undefined symbols in the assembly are treated as global.

The output of the assembly is left on the file **a.out.** It is executable if no errors occurred during the assembly, and if there were no unresolved external references.

FILES

/lib/as2	pass 2 of the assembler
/tmp/atm[1-3]?	temporary
a.out	object

SEE ALSO

ld (I), nm (I), db (I), a.out (V), 'UNIX Assembler Manual'.

DIAGNOSTICS

When an input file cannot be read, its name followed by a question mark is typed and assembly ceases. When syntactic or semantic errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

)	Parentheses error
]	Parentheses error
<	String not terminated properly
*	Indirection used illegally
.	Illegal assignment to '.'
A	Error in address
B	Branch instruction is odd or too remote
E	Error in expression
F	Error in local ('f' or 'b') type symbol
G	Garbage (unknown) character
I	End of file inside an if
M	Multiply defined symbol as label
O	Word quantity assembled at odd address
P	'.' different in pass 1 and 2
R	Relocation error
U	Undefined symbol
X	Syntax error

BUGS

Symbol table overflow is not checked. **x** errors can cause incorrect line numbers in following diagnostics.

NAME

bas – basic

SYNOPSIS

bas [file]

DESCRIPTION

Bas is a dialect of Basic. If a file argument is provided, the file is used for input before the console is read. *Bas* accepts lines of the form:

statement
integer statement

Integer numbered statements (known as internal statements) are stored for later execution. They are stored in sorted ascending order. Non-numbered statements are immediately executed. The result of an immediate expression statement (that does not have '=' as its highest operator) is printed.

Statements have the following syntax:

expression
The expression is executed for its side effects (assignment or function call) or for printing as described above.

comment ...
This statement is ignored. It is used to interject commentary in a program.

done
Return to system level.

draw expression expression expression
A line is drawn on the Tektronix 611 display '/dev/vt0' from the current display position to the XY co-ordinates specified by the first two expressions. The scale is zero to one in both X and Y directions. If the third expression is zero, the line is invisible. The current display position is set to the end point.

display list
The list of expressions and strings is concatenated and displayed (i.e. printed) on the 611 starting at the current display position. The current display position is not changed.

dump
The name and current value of every variable is printed.

edit
The UNIX editor, *ed,* is invoked with the *file* argument. After the editor exits, this file is recompiled.

erase
The 611 screen is erased.

for name **=** expression expression statement
for name **=** expression expression
...
next
The *for* statement repetitively executes a statement (first form) or a group of statements (second form) under control of a named variable. The variable takes on the value of the first expression, then is incremented by one on each loop, not to exceed the value of the second expression.

goto expression
The expression is evaluated, truncated to an integer and execution goes to the corresponding integer numbered statment. If executed from immediate mode, the internal statements are compiled first.

if expression statement
if expression
...
[**else**
...]

fi

The statement (first form) or group of statements (second form) is executed if the expression evaluates to non-zero. In the second form, an optional **else** allows for a group of statements to be executed when the first group is not.

list [expression [expression]]

is used to print out the stored internal statements. If no arguments are given, all internal statements are printed. If one argument is given, only that internal statement is listed. If two arguments are given, all internal statements inclusively between the arguments are printed.

print list

The list of expressions and strings are concatenated and printed. (A string is delimited by " characters.)

prompt list

Prompt is the same as *print* except that no newline character is printed.

return [expression]

The expression is evaluated and the result is passed back as the value of a function call. If no expression is given, zero is returned.

run

The internal statements are compiled. The symbol table is re-initialized. The random number generator is reset. Control is passed to the lowest numbered internal statement.

save [expression [expression]]

Save is like *list* except that the output is written on the *file* argument. If no argument is given on the command, **b.out** is used.

Expressions have the following syntax:

name

A name is used to specify a variable. Names are composed of a letter followed by letters and digits. The first four characters of a name are significant.

number

A number is used to represent a constant value. A number is written in Fortran style, and contains digits, an optional decimal point, and possibly a scale factor consisting of an **e** followed by a possibly signed exponent.

(expression **)**

Parentheses are used to alter normal order of evaluation.

_ expression

The result is the negation of the expression.

expression operator expression

Common functions of two arguments are abbreviated by the two arguments separated by an operator denoting the function. A complete list of operators is given below.

expression **(** [expression [**,** expression] ...] **)**

Functions of an arbitrary number of arguments can be called by an expression followed by the arguments in parentheses separated by commas. The expression evaluates to the line number of the entry of the function in the internally stored statements. This causes the internal statements to be compiled. If the expression evaluates negative, a builtin function is called. The list of builtin functions appears below.

name **[** expression [**,** expression] ... **]**

Each expression is truncated to an integer and used as a specifier for the name. The result is syntactically identical to a name. **a[1,2]** is the same as **a[1][2].** The truncated expressions are restricted to values between 0 and 32767.

The following is the list of operators:

=

= is the assignment operator. The left operand must be a name or an array element. The result is the right operand. Assignment binds right to left, all other operators bind left to right.

& |

& (logical and) has result zero if either of its arguments are zero. It has result one if both its arguments are non-zero. | (logical or) has result zero if both of its arguments are zero. It has result one if either of its arguments are non-zero.

< <= > >= == <>

The relational operators (< less than, <= less than or equal, > greater than, >= greater than or equal, == equal to, <> not equal to) return one if their arguments are in the specified relation. They return zero otherwise. Relational operators at the same level extend as follows: a>b>c is the same as a>b&b>c.

+ −

Add and subtract.

* /

Multiply and divide.

^

Exponentiation.

The following is a list of builtin functions:

arg(i)

is the value of the *i* -th actual parameter on the current level of function call.

exp(x)

is the exponential function of *x*.

log(x)

is the natural logarithm of *x*.

sqr(x)

is the square root of *x*.

sin(x)

is the sine of *x* (radians).

cos(x)

is the cosine of *x* (radians).

atn(x)

is the arctangent of *x*. Its value is between $-\pi/2$ and $\pi/2$.

rnd( )

is a uniformly distributed random number between zero and one.

expr( )

is the only form of program input. A line is read from the input and evaluated as an expression. The resultant value is returned.

abs(x)

is the absolute value of *x*.

int(x)

returns *x* truncated (towards 0) to an integer.

FILES

/tmp/btm?	temporary
b.out	save file

DIAGNOSTICS

Syntax errors cause the incorrect line to be typed with an underscore where the parse failed. All other diagnostics are self explanatory.

BUGS

Has been known to give core images.

NAME

bc – arbitrary precision interactive language

SYNOPSIS

bc [**−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 a library of mathematical subroutines which contains sine (named 's'), cosine ('c'), arctangent ('a'), natural logarithm ('l'), and exponential ('e'). The syntax for *bc* programs is as follows; E means expression, S means statement.

Comments
are enclosed in /* and */.

Names
letters a−z
array elements: letter[E]
The words 'ibase', 'obase', and 'scale'

Other operands
arbitrarily long numbers with optional sign and decimal point.
(E)
sqrt (E)
<letter> (E , ... , E)

Operators
+ − * / % ^
++ −− (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 are exemplified by

```
define <letter> ( <letter> ,..., <letter> ) {
        auto <letter>, ... , <letter>
        S; ... S
        return ( E )
}
```

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. Assignments to *ibase* or *obase* set the input and output number radix respectively.

The same letter may be used as an array name, a function name, and a simple variable simultaneously. 'Auto' variables are saved and restored during function calls. All other variables are global to the program. 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.

FILES

/usr/lib/lib.b mathematical library

SEE ALSO

dc (I), C Reference Manual, ‘‘BC − An Arbitrary Precision Desk-Calculator Language.’’

BUGS

No &&, | | yet.
for statement must have all three E’s
quit is interpreted when read, not when executed.

NAME

cat – concatenate and print

SYNOPSIS

cat 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 input file is given, or if the argument '–' is encountered, *cat* reads from the standard input file.

SEE ALSO

pr(I), cp(I)

DIAGNOSTICS

none; if a file cannot be found it is ignored.

BUGS

cat x y >x and **cat x y >y** cause strange results.

NAME

cc – C compiler

SYNOPSIS

cc [**−c**] [**−p**] [**−f**] [**−O**] [**−S**] [**−P**] file ...

DESCRIPTION

Cc is the UNIX C compiler. It accepts three 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.

The following flags are interpreted by *cc.* See *ld (I)* for load-time flags.

−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 the *monitor* subroutine (III) 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* (I).

−f In systems without hardware floating-point, use a version of the C compiler which handles floating-point constants and loads the object program with the floating-point interpreter. Do not use if the hardware is present.

−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 on the named C programs, and leave the output on corresponding files suffixed '.i'.

Other arguments are taken to be either loader flag 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/ctm?	temporary
/lib/c[01]	compiler
/lib/fc[01]	floating-point compiler
/lib/c2	optional optimizer
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	runtime startoff of profiling
/lib/fcrt0.o	runtime startoff for floating-point interpretation
/lib/libc.a	C library; see section III.
/lib/liba.a	Assembler library used by some routines in libc.a

SEE ALSO

"Programming in C— a tutorial," C Reference Manual, monitor (III), prof (I), cdb (I), ld (I).

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self-explanatory. Occasional messages may be produced by the assembler or loader. Of these, the most mystifying are from the assembler, in particular "m," which means a multiply-defined external symbol (function or data).

BUGS

NAME

chdir – change working directory

SYNOPSIS

chdir 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, *chdir* would be ineffective if it were written as a normal command. It is therefore recognized and executed by the Shell.

SEE ALSO

sh (I), pwd (I)

BUGS

NAME

chmod – change mode

SYNOPSIS

chmod octal file ...

DESCRIPTION

The octal mode replaces the mode of each of the files. The mode is constructed from the OR of the following modes:

4000	set user ID on execution
2000	set group ID on execution
1000	sticky bit for shared, pure-procedure programs (see below)
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

Only the owner of a file (or the super-user) may change its mode.

If an executable file is set up for sharing (''–n'' option of *ld (I)*), 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.

SEE ALSO

ls (I), chmod (II)

BUGS

NAME

cmp – compare two files

SYNOPSIS

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. Moreover, return code 0 is yielded for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

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 (I), comm (I)

BUGS

-

NAME

comm – print lines common to two files

SYNOPSIS

comm [– [**123**]] file1 file2

DESCRIPTION

Comm reads *file1* and *file2,* which should be in sort, 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 (I), diff (I)

BUGS

NAME

cp – copy

SYNOPSIS

cp file1 file2

DESCRIPTION

The first file is copied onto the second. The mode and owner of the target file are preserved if it already existed; the mode of the source file is used otherwise.

If *file2* is a directory, then the target file is a file in that directory with the file-name of *file1*.

It is forbidden to copy a file onto itself.

SEE ALSO

cat (I), pr (I), mv (I)

BUGS

NAME

cref − make cross reference listing

SYNOPSIS

cref [−**acilostux123**] name ...

DESCRIPTION

Cref makes a cross reference listing of program files in assembler or C format. The files named as arguments in the command line are searched for symbols in the appropriate syntax.

The output report is in four columns:

(1)	(2)	(3)	(4)
symbol	file	see below	text as it appears in file

Cref uses either an *ignore* file or an *only* file. If the −**i** option is given, the next argument is taken to be an *ignore* file; if the −**o** option is given, the next argument is taken to be an *only* file. *Ignore* and *only* files are lists of symbols separated by new lines. All symbols in an *ignore* file are ignored in columns (1) and (3) of the output. If an *only* file is given, only symbols in that file appear in column (1). At most one of −**i** and −**o** may be used. The default setting is −**i.** Assembler predefined symbols or C keywords are ignored.

The −**s** option causes current symbols to be put in column 3. In the assembler, the current symbol is the most recent name symbol; in C, the current function name. The −**l** option causes the line number within the file to be put in column 3.

The −**t** option causes the next available argument to be used as the name of the intermediate temporary file (instead of /tmp/crt??). The file is created and is not removed at the end of the process.

Options:

- **a** assembler format (default)
- **c** C format input
- **i** use *ignore* file (see above)
- **l** put line number in col. 3 (instead of current symbol)
- **o** use *only* file (see above)
- **s** current symbol in col. 3 (default)
- **t** user supplied temporary file
- **u** print only symbols that occur exactly once
- **x** print only C external symbols
- **1** sort output on column 1 (default)
- **2** sort output on column 2
- **3** sort output on column 3

FILES

/tmp/crt??	temporaries
/usr/lib/aign	default assembler *ignore* file
/usr/lib/cign	default C *ignore* file
/usr/bin/crpost	post processor
/usr/bin/upost	post processor for −**u** option

/bin/sort used to sort temporaries

SEE ALSO

as (I), cc (I)

BUGS

NAME

date – print and set the date

SYNOPSIS

date [**s**] [mmddhhmm[yy]]

DESCRIPTION

If no argument is given, the current date and time are printed. If an argument is given, the current date is set. 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; *yy* is the last 2 digits of the year number and is optional. For example:

date 10080045

sets the date to Oct 8, 12:45 AM. The current year is the default if no year is mentioned. The system operates in GMT. *Date* takes care of the conversion to and from local standard and daylight time.

If the argument is ‘‘s,’’ *date* calls the network file store via the TIU interface (if present) and sets the clock to the time thereby obtained.

DIAGNOSTICS

‘‘No permission’’ if you aren’t the super-user and you try to change the date; ‘‘bad conversion’’ if the date set is syntactically incorrect.

FILES

/dev/tiu/d0

BUGS

NAME

db – debug

SYNOPSIS

db [core [namelist]] [–]

DESCRIPTION

Unlike many debugging packages (including DEC's ODT, on which *db* is loosely based), *db* is not loaded as part of the core image which it is used to examine; instead it examines files. Typically, the file will be either a core image produced after a fault or the binary output of the assembler. *Core* is the file being debugged; if omitted **core** is assumed. *Namelist* is a file containing a symbol table. If it is omitted, the symbol table is obtained from the file being debugged, or if not there from **a.out.** If no appropriate name list file can be found, *db* can still be used but some of its symbolic facilities become unavailable.

For the meaning of the optional third argument, see the last paragraph below.

The format for most *db* requests is an address followed by a one character command. Addresses are expressions built up as follows:

1. A name has the value assigned to it when the input file was assembled. It may be relocatable or not depending on the use of the name during the assembly.
2. An octal number is an absolute quantity with the appropriate value.
3. A decimal number immediately followed by '**.**' is an absolute quantity with the appropriate value.
4. An octal number immediately followed by **r** is a relocatable quantity with the appropriate value.
5. The symbol **.** indicates the current pointer of *db*. The current pointer is set by many *db* requests.
6. A * before an expression forms an expression whose value is the number in the word addressed by the first expression. A * alone is equivalent to '***.**'.
7. Expressions separated by + or blank are expressions with value equal to the sum of the components. At most one of the components may be relocatable.
8. Expressions separated by – form an expression with value equal to the difference to the components. If the right component is relocatable, the left component must be relocatable.
9. Expressions are evaluated left to right.

Names for registers are built in:

r0 ... r5
sp
pc
fr0 ... fr5

These may be examined. Their values are deduced from the contents of the stack in a core image file. They are meaningless in a file that is not a core image.

If no address is given for a command, the current address (also specified by "**.**") is assumed. In general, "**.**" points to the last word or byte printed by *db.*

There are *db* commands for examining locations interpreted as numbers, machine instructions, ASCII characters, and addresses. For numbers and characters, either bytes or words may be examined. The following commands are used to examine the specified file.

/ The addressed word is printed in octal.

\ The addressed byte is printed in octal.

" The addressed word is printed as two ASCII characters.

´ The addressed byte is printed as an ASCII character.

` The addressed word is printed in decimal.

? The addressed word is interpreted as a machine instruction and a symbolic form of the instruction, including symbolic addresses, is printed. Often, the result will appear exactly as it was written in the source program.

& The addressed word is interpreted as a symbolic address and is printed as the name of the symbol whose value is closest to the addressed word, possibly followed by a signed offset.

<nl>(i. e., the character ''new line'') This command advances the current location counter ''**.**'' and prints the resulting location in the mode last specified by one of the above requests.

^ This character decrements ''**.**'' and prints the resulting location in the mode last selected one of the above requests. It is a converse to <nl>.

% Exit.

Odd addresses to word-oriented commands are rounded down. The incrementing and decrementing of ''**.**'' done by the **<nl>** and ^ requests is by one or two depending on whether the last command was word or byte oriented.

The address portion of any of the above commands may be followed by a comma and then by an expression. In this case that number of sequential words or bytes specified by the expression is printed. ''**.**'' is advanced so that it points at the last thing printed.

There are two commands to interpret the value of expressions.

= When preceded by an expression, the value of the expression is typed in octal. When not preceded by an expression, the value of ''**.**'' is indicated. This command does not change the value of ''**.**''.

: An attempt is made to print the given expression as a symbolic address. If the expression is relocatable, that symbol is found whose value is nearest that of the expression, and the symbol is typed, followed by a sign and the appropriate offset. If the value of the expression is absolute, a symbol with exactly the indicated value is sought and printed if found; if no matching symbol is discovered, the octal value of the expression is given.

The following command may be used to patch the file being debugged.

! This command must be preceded by an expression. The value of the expression is stored at the location addressed by the current value of ''**.**''. The opcodes do not appear in the symbol table, so the user must assemble them by hand.

The following command is used after a fault has caused a core image file to be produced.

$ causes the fault type and the contents of the general registers and several other registers to be printed both in octal and symbolic format. The values are as they were at the time of the fault.

For some purposes, it is important to know how addresses typed by the user correspond with locations in the file being debugged. The mapping algorithm employed by *db* is non-trivial for two reasons: First, in an **a.out** file, there is a 20(8) byte header which will not appear when the file is loaded into core for execution. Therefore, apparent location 0 should correspond with actual file offset 20. Second, addresses in core images do not correspond with the addresses used by the program because in a core image there is a header containing the system's per-process data for the dumped process, and also because the stack is stored contiguously with the text and data part of the core image rather than at the highest possible locations. *Db* obeys the following rules:

If exactly one argument is given, and if it appears to be an **a.out** file, the 20-byte header is skipped during addressing, i.e., 20 is added to all addresses typed. As a consequence, the header can be examined beginning at location −20.

If exactly one argument is given and if the file does not appear to be an **a.out** file, no mapping is done.

If zero or two arguments are given, the mapping appropriate to a core image file is employed. This means that locations above the program break and below the stack effectively do not exist (and are not, in fact, recorded in the core file). Locations above the user's stack pointer are mapped, in looking at the core file, to the place where they are really stored. The per-process data kept by the system, which is stored in the first part of the core file, cannot currently be examined (except by **$**).

If one wants to examine a file which has an associated name list, but is not a core image file, the last argument ‘‘−’’ can be used (actually the only purpose of the last argument is to make the number of arguments not equal to two). This feature is used most frequently in examining the memory file /dev/mem.

SEE ALSO

as (I), core (V), a.out (V), od (I)

DIAGNOSTICS

‘‘File not found’’ if the first argument cannot be read; otherwise ‘‘**?**’’.

BUGS

There should be some way to examine the registers and other per-process data in a core image; also there should be some way of specifying double-precision addresses. It does not know yet about shared text segments.

NAME

dc – desk calculator

SYNOPSIS

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.

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.

[...] 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.

o The top value on the stack is popped and used as the number radix for further output.

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.

? A line of input is taken from the input source (usually the console) and executed.

An example which prints the first ten values of n! is

```
[la1+dsa*pla10>y]sy
0sa1
lyx
```

SEE ALSO

bc (I), 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) ? for unrecognized character x.
(x) ? for not enough elements on the stack to do what was asked by command x.
'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.

BUGS

NAME

dd – convert and copy a file

SYNOPSIS

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=	input block size (default 512)
obs=	output block size (default 512)
bs=	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
count=*n*	copy only *n* input records
conv=ascii	convert EBCDIC to ASCII
ebcdic	convert 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. Also 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.

SEE ALSO

cp (I)

BUGS

The ASCII/EBCDIC conversion tables are taken from the 256 character standard in the CACM Nov, 1968. It is not clear how this relates to real life.

Newlines are inserted only on conversion to ASCII; padding is done only on conversion to EBCDIC. There should be separate options.

NAME

diff – differential file comparator

SYNOPSIS

diff [–] name1 name2

DESCRIPTION

Diff tells what lines must be changed in two files to bring them into agreement. 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 file *name1* into file *name2.* The numbers after the letters pertain to file *name2.* In fact, by exchanging 'a' for 'd' and reading backward one may ascertain equally how to convert file *name2* into *name1.* 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 '**.**'.

Under the – option, the output of *diff* is a script of *a, c* and *d* commands for the editor *ed,* which will change the contents of the first file into the contents of the second. In this connection, 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.

```
(cat $2 ... $9; echo "1,$p") | ed – $1
```

Except for occasional 'jackpots', *diff* finds a smallest sufficient set of file differences.

SEE ALSO

cmp (I), comm (I), ed (I)

DIAGNOSTICS

'jackpot' – To speed things up, the program uses hashing. You have stumbled on a case where there is a chance that this has resulted in a difference being called where none actually existed. Sometimes reversing the order of files will make a jackpot go away.

BUGS

Editing scripts produced under the – option are naive about creating lines consisting of a single '**.**'.

NAME

dsw − delete interactively

SYNOPSIS

dsw [directory]

DESCRIPTION

For each file in the given directory (‘**.**’ if not specified) *dsw* types its name. If **y** is typed, the file is deleted; if **x,** *dsw* exits; if new-line, the file is not deleted; if anything else, *dsw* asks again.

SEE ALSO

rm (I)

BUGS

The name *dsw* is a carryover from the ancient past. Its etymology is amusing.

NAME

du − summarize disk usage

SYNOPSIS

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.

BUGS

Non-directories given as arguments (not under −**a** option) are not listed.

Removable file systems do not work correctly since i-numbers may be repeated while the corresponding files are distinct. *Du* should maintain an i-number list per root directory encountered.

NAME

echo – echo arguments

SYNOPSIS

echo [arg ...]

DESCRIPTION

Echo writes its arguments in order as a line on the standard output file. It is mainly useful for producing diagnostics in command files.

BUGS

NAME

ed – text editor

SYNOPSIS

ed [–] [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. 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.* There is only one 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. Every command which requires addresses has default addresses, so that the addresses can often be omitted.

In general, only one command may appear on a line. Certain commands allow the input of text. This text is placed in the appropriate place in 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. The regular expressions allowed by *ed* are constructed as follows:

1. An ordinary character (not one of those discussed below) is a regular expression and matches that character.
2. A circumflex '^' at the beginning of a regular expression matches the empty string at the beginning of a line.
3. A currency symbol '$' at the end of a regular expression matches the null character at the end of a line.
4. A period '**.**' matches any character except a new-line character.
5. A regular expression followed by an asterisk '*' matches any number of adjacent occurrences (including zero) of the regular expression it follows.
6. A string of characters enclosed in square brackets '[ ]' matches any character in the string but no others. If, however, the first character of the string is a circumflex '^' the regular expression matches any character except new-line and the characters in the string.
7. The concatenation of regular expressions is a regular expression which matches the concatenation of the strings matched by the components of the regular expression.
8. A regular expression enclosed between the sequences '\(' and '\)'is identical to the unadorned expression; the construction has side effects discussed under the *s* command.
9. The null regular expression standing alone is equivalent to 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 mark name character *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 first line found by searching toward the end of the buffer and stopping at the first line containing a string matching the regular expression. If necessary the search wraps around to the beginning of the buffer.
6. A regular expression enclosed in queries '?' addresses the first line found by searching toward the beginning of the buffer and stopping at the first line containing a string matching 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 version of the editor, the character '^' in addresses is entirely 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, any command 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 first line not deleted.

(**.** , **.**) 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.

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 (global) 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; if there were none, at the addressed line. This command differs from the *a* command only in the placement of the text.

(**.**) k*x*

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 octal, and long lines are folded. An *l* command may follow any other on the same line.

(**.** , **.**) m*a*

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 command.

q

The quit command causes *ed* to exit. No automatic write of a file is done.

($) 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 remembered file name is not changed unless 'filename' is the very first file name mentioned. 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 '\'. As a more general feature, 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 '\'.

(**.** , **.**) t *a*
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.

(1,$) v/regular expression/command list
This command is the same as the global command except that the command list is executed 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 writeable by everyone). The remembered file name is *not* changed unless 'filename' is the very first file name mentioned. 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 typed.

($) =
The line number of the addressed line is typed. '**.**' is unchanged by this command.

!UNIX command
The remainder of the line after the '!' is sent to UNIX 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.

FILES

/tmp/#, temporary; '#' is the process number (in octal).

DIAGNOSTICS

'?' for errors in commands; 'TMP' for temporary file overflow.

SEE ALSO

A Tutorial Introduction to the ED Text Editor (B. W. Kernighan)

BUGS

The *s* command causes all marks to be lost on lines changed.

NAME

eqn − typeset mathematics

SYNOPSIS

eqn [file] ...

DESCRIPTION

Eqn is a troff (I) preprocessor for typesetting mathematics on the Graphics Systems phototypesetter. Usage is almost always

eqn file ... | troff

If no files are specified, *eqn* reads from the standard input. A line beginning with ''.EQ'' marks the start of an equation; the end of an equation is marked by a line beginning with ''.EN''. Neither of these lines is altered or defined by *eqn,* so you can define them yourself to get centering, numbering, etc. All other lines are treated as comments, and passed through untouched.

Spaces, tabs, newlines, braces, double quotes, tilde and circumflex are the only delimiters. Braces ''{}'' are used for grouping. Use tildes ''~'' to get extra spaces in an equation.

Subscripts and superscripts are produced with the keywords **sub** and **sup.** Thus *x sub i* makes x_i, *a sub i sup 2* produces a_i^2, and *e sup {x sup 2 + y sup 2}* gives $e^{x^2+y^2}$. Fractions are made with **over.** *a over b* is $\frac{a}{b}$ and *1 over sqrt {ax sup 2 +bx+c}* is $\frac{1}{\sqrt{ax^2+bx+c}}$. **sqrt** makes square roots.

The keywords **from** and **to** introduce lower and upper limits on arbitrary things: $\lim_{n\to\infty} \sum_0^n x_i$ is made with *lim from {n-> inf} sum from 0 to n x sub i.* Left and right brackets, braces, etc., of the right height are made with **left** and **right:** *left [x sup 2 + y sup 2 over alpha right] ~=~1* produces $\left[x^2 + \frac{y^2}{\alpha} \right] = 1$. The **right** clause is optional.

Vertical piles of things are made with **pile, lpile, cpile,** and **rpile:** *pile {a above b above c}* produces $\begin{matrix} a \\ b \\ c \end{matrix}$. There can be an arbitrary number of elements in a pile. **lpile** left-justifies, **pile** and **cpile** center, with different vertical spacing, and **rpile** right justifies.

Diacritical marks are made with **dot, dotdot, hat, bar:** *x dot = f(t) bar* is $\dot{x} = \overline{f(t)}$. Default sizes and fonts can be changed with **size n** and various of **roman, italic,** and **bold.**

Keywords like *sum* ($\sum$) *int* ($\int$) *inf* (∞) and shorthands like >= (≥) -> (→), != (≠), are recognized. Spell out Greek letters in the desired case, as in *alpha, GAMMA.* Mathematical words like sin, cos, log are made Roman automatically. Troff (I) four-character escapes like \(bs () can be used anywhere. Strings enclosed in double quotes "..." are passed through untouched.

SEE ALSO

A System for Typesetting Mathematics (Computer Science Technical Report #17, Bell Laboratories, 1974.)
TROFF Users' Manual (internal memorandum)
TROFF Made Trivial (internal memorandum)
troff (I), neqn (I)

BUGS

Undoubtedly. Watch out for small or large point sizes − it's tuned too well for size 10. Be cautious if inserting horizontal or vertical motions, and of backslashes in general.

-

NAME

exit − terminate command file

SYNOPSIS

exit

DESCRIPTION

Exit performs a **seek** to the end of its standard input file. Thus, if it is invoked inside a file of commands, upon return from *exit* the shell will discover an end-of-file and terminate.

SEE ALSO

if (I), goto (I), sh (I)

BUGS

NAME

fc – Fortran compiler

SYNOPSIS

fc [**−c**] sfile1.f ... ofile1 ...

DESCRIPTION

Fc is the UNIX Fortran compiler. It accepts three types of arguments:

Arguments whose names end with '.f' are assumed to be Fortran source program units; they are compiled, and the object program is left on the file sfile1.o (i.e. the file whose name is that of the source with '.o' substituted for '.f').

Other arguments (except for **−c**) are assumed to be either loader flags, or object programs, typically produced by an earlier *fc* run, or perhaps libraries of Fortran-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.**

The **−c** argument suppresses the loading phase, as does any syntax error in any of the routines being compiled.

The following is a list of differences between *fc* and ANSI standard Fortran (also see the BUGS section):

1. Arbitrary combination of types is allowed in expressions. Not all combinations are expected to be supported at runtime. All of the normal conversions involving integer, real, double precision and complex are allowed.
2. Two forms of ''implicit'' statements are recognized: **implicit integer /i–n/** or **implicit integer (i−n).**
3. The types doublecomplex, logical*1, integer*1, integer*2, integer*4 (same as integer), real*4 (real), and real*8 (double precision) are supported.
4. **&** as the first character of a line signals a continuation card.
5. **c** as the first character of a line signals a comment.
6. All keywords are recognized in lower case.
7. The notion of 'column 7' is not implemented.
8. G-format input is free form– leading blanks are ignored, the first blank after the start of the number terminates the field.
9. A comma in any numeric or logical input field terminates the field.
10. There is no carriage control on output.
11. A sequence of *n* characters in double quotes '"' is equivalent to *n* **h** followed by those characters.
12. In **data** statements, a hollerith string may initialize an array or a sequence of array elements.
13. The number of storage units requested by a binary **read** must be identical to the number contained in the record being read.
14. If the first character in an input file is ''#'', a preprocessor identical to the C preprocessor is called, which implements ''#define'' and ''#include'' preprocessor statements. (See the C reference manual for details.) The preprocessor does not recognize Hollerith strings written with *n* **h**.

In I/O statements, only unit numbers 0-19 are supported. Unit number *n* refers to file fort*nn;* (e.g. unit 9 is file 'fort09'). For input, the file must exist; for output, it will be created. Unit 5 is permanently associated with the standard input file; unit 6 with the standard output file. Also see *setfil* (III) for a way to associate unit numbers with named files.

FILES

a.out	loaded output
f.tmp[123]	temporary (deleted)
/usr/fort/fc1	compiler proper
/lib/fr0.o	runtime startoff

/lib/filib.a	interpreter library
/lib/libf.a	builtin functions, etc.
/lib/liba.a	system library

SEE ALSO

rc (I), which announces a more pleasant Fortran dialect; the ANSI standard; ld (I) for loader flags. For some subroutines, try ierror, getarg, setfil (III).

DIAGNOSTICS

Compile-time diagnostics are given in English, accompanied if possible with the offending line number and source line with an underscore where the error occurred. Runtime diagnostics are given by number as follows:

1	invalid log argument
2	bad arg count to amod
3	bad arg count to atan2
4	excessive argument to cabs
5	exp too large in cexp
6	bad arg count to cmplx
7	bad arg count to dim
8	excessive argument to exp
9	bad arg count to idim
10	bad arg count to isign
11	bad arg count to mod
12	bad arg count to sign
13	illegal argument to sqrt
14	assigned/computed goto out of range
15	subscript out of range
16	real**real overflow
17	(negative real)**real
100	illegal I/O unit number
101	inconsistent use of I/O unit
102	cannot create output file
103	cannot open input file
104	EOF on input file
105	illegal character in format
106	format does not begin with (
107	no conversion in format but non-empty list
108	excessive parenthesis depth in format
109	illegal format specification
110	illegal character in input field
111	end of format in hollerith specification
112	bad argument to setfil
120	bad argument to ierror
999	unimplemented input conversion

BUGS

The following is a list of those features not yet implemented:
arithmetic statement functions
scale factors on input
Backspace statement.

NAME

file – determine file type

SYNOPSIS

file file ...

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.

BUGS

NAME

find – find files

SYNOPSIS

find pathname expression

DESCRIPTION

Find recursively descends the directory hierarchy from *pathname* 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(I)). If *onum* is prefixed by a minus sign, more flag bits (017777, see stat(II)) 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.*
−**group** gname	As it is for −**user** so shall it be for −**group** (someday).
−**size** *n*	True if the file is *n* blocks long (512 bytes per block).
−**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 exit status zero (most commands do). The end of the command is punctuated by an escaped semicolon. A command argument '{}' is replaced by the current pathname.
−**ok** command	Like −**exec** except that the generated command line is printed with a question mark first, and is executed only if the user responds **y**.
−**print**	Always true; causes the current pathname to be printed.

The primaries may be combined with these operators (ordered by precedence):

!	prefix *not*
−**a**	infix *and,* second operand evaluated only if first is true
−**o**	infix *or,* second operand evaluated only if first is false
(expression **)**	parentheses for grouping. (Must be escaped.)

To remove files named 'a.out' and '*.o' not accessed for a week:

```
find / "(" −name a.out −o −name "*.o" ")" −a −atime +7 −a −exec rm {} ";"
```

FILES

/etc/passwd

SEE ALSO

sh (I), if(I), file system (V)

BUGS

There is no way to check device type.
Syntax should be reconciled with *if.*

NAME

goto − command transfer

SYNOPSIS

goto label

DESCRIPTION

Goto is allowed only when the Shell is taking commands from a file. The file is searched from the beginning for a line beginning with ':' followed by one or more spaces followed by the *label*. If such a line is found, the *goto* command returns. Since the read pointer in the command file points to the line after the label, the effect is to cause the Shell to transfer to the labelled line.

':' is a do-nothing command that is ignored by the Shell and only serves to place a label.

SEE ALSO

sh (I)

BUGS

NAME

grep – search a file for a pattern

SYNOPSIS

grep [**−v**] [**−b**] [**−c**] [**−n**] expression [file] ...

DESCRIPTION

Grep searches the input files (standard input default) for lines matching the regular expression. Normally, each line found is copied to the standard output. If the **−v** flag is used, all lines but those matching are printed. If the **−c** flag is used, only a count of matching lines is printed. If the **−n** flag is used, each line is preceded its relative line number in the file. If the **−b** flag is used, each line is preceded by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.

In all cases the file name is shown if there is more than one input file.

For a complete description of the regular expression, see ed (I). Care should be taken when using the characters $ * [^ | ( ) and \ in the regular expression as they are also meaningful to the Shell. It is generally necessary to enclose the entire *expression* argument in quotes.

SEE ALSO

ed (I), sh (I)

BUGS

Lines are limited to 256 characters; longer lines are truncated.

NAME

if − conditional command

SYNOPSIS

if expr command [arg ...]

DESCRIPTION

If evaluates the expression *expr,* and if its value is true, executes the given *command* with the given arguments.

The following primitives are used to construct the *expr:*

−r file	true if the file exists and is readable.
−w file	true if the file exists and is writable.
s1 **=** s2	true if the strings *s1* and *s2* are equal.
s1 **!=** s2	true if the strings *s1* and *s2* are not equal.
{ command **}**	The bracketed command is executed to obtain the exit status. Status zero is considered *true.* The command must not be another *if.*

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 *if* and hence must be surrounded by spaces. Notice also that parentheses are meaningful to the Shell and must be escaped.

SEE ALSO

sh (I), find (I)

BUGS

NAME

kill – terminate a process

SYNOPSIS

kill [–signo] processid ...

DESCRIPTION

Kills the specified processes. The process number of each asynchronous process started with '&' is reported by the Shell. Process numbers can also be found by using *ps* (I).

If process number 0 is used, then all processes belonging to the current user and associated with the same control typewriter are killed.

The killed process must belong to the current user unless he is the super-user.

If a signal number preceded by ''–'' is given as first argument, that signal is sent instead of *kill* (see *signal (II)*).

SEE ALSO

ps (I), sh (I), signal (II)

BUGS

NAME

ld − link editor

SYNOPSIS

ld [−**sulxrdni**] name ...

DESCRIPTION

Ld combines several object programs into one; resolves external references; and searches libraries. In the simplest case the names of several object programs 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, the referenced routine must appear after the referencing routine in the library. Thus the order of programs within libraries is important.

Ld understands several flag arguments which are written preceded by a '−'. Except for **−l**, they should appear before the file names.

−s 'squash' 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.*

−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 This option is an abbreviation for a library name. **−l** alone stands for '/lib/liba.a', which is the standard system library for assembly language programs. **−l***x* stands for '/lib/lib*x*.a' where *x* is any character. 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* 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 the 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.

FILES

/lib/lib?.a libraries
a.out output file

SEE ALSO

as (I), ar (I)

BUGS

NAME

ln − make a link

SYNOPSIS

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 (I)

BUGS

There is nothing particularly wrong with *ln,* but *tp* doesn't understand about links and makes one copy for each name by which a file is known; thus if the tape is extracted several copies are restored and the information that links were involved is lost.

NAME

login − sign onto UNIX

SYNOPSIS

login [username]

DESCRIPTION

The *login* command is used when a user initially signs onto UNIX, or it may be used at any time to change from one user to another. The latter case is the one summarized above and described here. See 'How to Get Started' for how to dial up initially.

If *login* is invoked without an argument, it asks for a user name, and, if appropriate, a password. Echoing is turned off (if possible) during the typing of the password, so it will not appear on the written record of the session.

After a successful login, accounting files are updated and the user is informed of the existence of and message-of-the-day files. *Login* initializes the user and group IDs and the working directory, then executes a command interpreter (usually *sh* (I)) according to specifications found in a password file.

Login is recognized by the Shell and executed directly (without forking).

FILES

/etc/utmp	accounting
/usr/adm/wtmp	accounting
.mail	mail
/etc/motd	message-of-the-day
/etc/passwd	password file

SEE ALSO

init (VIII), getty (VIII), mail (I), passwd (I), passwd (V)

DIAGNOSTICS

'login incorrect,' if the name or the password is bad. 'No Shell,', 'cannot open password file,' 'no directory': consult a UNIX programming counselor.

BUGS

NAME

ls − list contents of directory

SYNOPSIS

ls [**−ltasdruifg**] 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 those beginning with '**.**' are suppressed

−s give size in 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**)

−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 is normally blank but is printed as ''t'' if the 1000 bit of the mode is on. See *chmod (I)* for the current meaning of this mode.

FILES

/etc/passwd to get user ID's for **ls −l**.

BUGS

NAME

mail − send mail to designated users

SYNOPSIS

mail [**−yn**] [person ...]

DESCRIPTION

Mail with no argument searches for a file called prints it if it is nonempty, then asks if it should be saved. If the answer is **y,** the mail is added to *mbox.* Finally is truncated to zero length. To leave the mailbox untouched, hit 'delete.' The question can be answered on the command line with the argument '−y' or '−n'.

When *persons* are named, *mail* takes the standard input up to an end of file and adds it to each *person's* file. The message is preceded by the sender's name and a postmark.

A *person* is either a user name recognized by *login* (I), in which case the mail is sent to the default working directory of that user; or the path name of a directory, in which case in that directory is used.

When a user logs in he is informed of the presence of mail. No mail will be received from a sender to whom is inaccessible or unwritable.

FILES

/etc/passwd	to identify sender and locate persons
/etc/utmp	to identify sender
.mail	input mail
mbox	saved mail
/tmp/m#	temp file

SEE ALSO

write (I)

BUGS

NAME

man – run off section of UNIX manual

SYNOPSIS

man [section] [title ...]

DESCRIPTION

Man is a shell command file which locates and prints one or more sections of this manual. *Section* is the section number of the manual, as an Arabic not Roman numeral, and is optional. *Title* is one or more section names; these names bear a generally simple relation to the page captions in the manual. If the *section* is missing, **1** is assumed. For example,

man man

would reproduce this page.

FILES

/usr/man/man?/*

BUGS

The manual is supposed to be reproducible either on the phototypesetter or on a typewriter. However, on a typewriter some information is necessarily lost.

NAME

mesg − permit or deny messages

SYNOPSIS

mesg [**n**] [**y**]

DESCRIPTION

Mesg with argument **n** forbids messages via *write* by revoking non-user write permission on the user's typewriter. *Mesg* with argument **y** reinstates permission. All by itself, *mesg* reverses the current permission. In all cases the previous state is reported.

FILES

/dev/tty?

SEE ALSO

write (I)

DIAGNOSTICS

'?' if the standard input file is not a typewriter

BUGS

-

NAME

mkdir − make a directory

SYNOPSIS

mkdir dirname ...

DESCRIPTION

Mkdir creates specified directories in mode 777. The standard entries '**.**' and '**..**' are made automatically.

SEE ALSO

rmdir (I)

BUGS

NAME

mv − move or rename a file

SYNOPSIS

mv name1 name2

DESCRIPTION

Mv changes the name of *name1* to *name2*. If *name2* is a directory, *name1* is moved to that directory with its original file-name. Directories may only be moved within the same parent directory (just renamed).

If *name2* already exists, it is removed before *name1* is renamed. If *name2* has a mode which forbids writing, *mv* prints the mode and reads the standard input to obtain a line; if the line begins with **y,** the move takes place; if not, *mv* exits.

If *name2* would lie on a different file system, so that a simple rename is impossible, *mv* copies the file and deletes the original.

BUGS

It should take a **−f** flag, like *rm,* to suppress the question if the target exists and is not writable.

NAME

neqn – typeset mathematics on terminal

SYNOPSIS

neqn [file] ...

DESCRIPTION

Neqn is an nroff (I) preprocessor. The input language is the same as that of eqn (I). Normal usage is almost always

neqn file ... | nroff

Output is meant for terminals with forward and reverse capabilities, such as the Model 37 teletype or GSI terminal.

If no arguments are specified, *neqn* reads the standard input, so it may be used as a filter.

SEE ALSO

eqn (I), gsi (VI)

BUGS

Because of some interactions with *nroff* there may not always be enough space left before and after lines containing equations.

NAME

newgrp – log in to a new group

SYNOPSIS

newgrp group

DESCRIPTION

Newgrp changes the group identification of its caller, analogously to *login.* 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.'

FILES

/etc/group, /etc/passwd

SEE ALSO

login (I), group (V)

BUGS

NAME

nice – run a command at low priority

SYNOPSIS

nice [–*number*] command [arguments]

DESCRIPTION

Nice executes *command* with low scheduling priority. If a numerical argument is given, that priority (in the range 1-20) is used; if not, priority 4 is used.

The super-user may run commands with priority higher than normal by using a negative priority, e.g. '––10'.

SEE ALSO

nohup (I), nice (II)

BUGS

NAME

nm − print name list

SYNOPSIS

nm [**−cnrupg**] [name]

DESCRIPTION

Nm prints the symbol table from the output file of an assembler or loader run. 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.

If no file is given, the symbols in **a.out** are listed.

Options are:

−c list only C-style external symbols, that is those beginning with underscore '_'.

−g print only global (external) symbols

−n sort by value instead of by name

−p don't sort; print in symbol-table order

−r sort in reverse order

−u print only undefined symbols.

FILES

a.out

BUGS

-

NAME

nohup – run a command immune to hangups

SYNOPSIS

nohup command [arguments]

DESCRIPTION

Nohup executes *command* with hangups, quits and interrupts all ignored.

SEE ALSO

nice (I), signal (II)

BUGS

NAME

nroff − format text

SYNOPSIS

nroff [**+***n*] [−*n*] [**−n***n*] [**−r***an*] [**−m***x*] [−**s**] [−**h**] [−**q**] files

DESCRIPTION

Nroff formats text according to control lines embedded in the text files. *Nroff* reads the standard input if no file arguments are given. An argument of just "−" refers to the standard input. The non-file option arguments are interpreted as follows:

+*n*	Output commences at the first page whose page number is *n* or larger.
−*n*	Printing stops after page *n.*
−n*n*	First generated (not necessarily printed) page is given number *n*; simulates ".pn *n*".
−r*an*	Set number register to the value *n.*
−m*name*	Prepends a standard macro file; simulates ".so /usr/lib/tmac.*name*".
−**s**	Stop prior to each page to permit paper loading. Printing is restarted by typing a 'newline' character.
−**h**	Spaces are replaced where possible with tabs to speed up output (or reduce the size of the output file).
−**q**	Prompt names for insertions are not printed and the bell character is sent instead; the insertion is not echoed.

FILES

/usr/lib/suftab	suffix hyphenation tables
/tmp/rtm?	temporary
/usr/lib/tmac.*	standard macro files

SEE ALSO

NROFF User's Manual (internal memorandum).
neqn (I), col (I)

BUGS

NAME

od − octal dump

SYNOPSIS

od [−**abcdho**] [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:

a interprets words as PDP-11 instructions and dis-assembles the operation code. Unknown operation codes print as ???.

b interprets bytes in octal.

c interprets bytes in ascii. Unknown ascii characters are printed as \?.

d interprets words in decimal.

h interprets words in hex.

o interprets words in octal.

The *file* argument specifies which file is to be dumped. If no file argument is specified, the standard input is used. Thus *od* can be used as a filter.

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. (A block is 512 bytes.) If the file argument is omitted, the offset argument must be preceded by '**+**'.

Dumping continues until end-of-file.

SEE ALSO

db (I)

BUGS

NAME

opr – off line print

SYNOPSIS

opr [–destination] [–**crm**] [name ...]

DESCRIPTION

Opr causes the named files to be printed off line at the specified destination. If no names appear the standard input is assumed.

At the mother system the following destinations are recognized. The default destination is **mh.**

lp Local line printer.

mh GCOS at Murray Hill Comp Center. GCOS identification must be registered in the UNIX password file (see passwd (V)).

sp Spider network printer.

xx The two-character code *xx* is taken to be a Murray Hill GCOS station id. Useful codes are 'r1' for quality print and 'q1' for quality print with special ribbon.

Opr uses spooling daemons that do the job when facilities become available. Flag –**r** causes the named files to be removed when spooled. Flag –**c** causes copies to be made so as to insulate the daemons from any intervening changes to the files.

Flag –**m** causes mail to be sent when UNIX is finished transmitting the file. For GCOS jobs the mail includes the snumb.

FILES

/etc/passwd	personal ident cards
/lib/dpr	dataphone spooler
/etc/dpd	dataphone daemon
/usr/dpd/*	spool area
/lib/lpr	line printer spooler
/etc/lpd	line printer daemon
/usr/lpd/*	spool area
/lib/npr	spider network spooler

SEE ALSO

fsend (I), dpd (VIII), lpd (VIII)

BUGS

Line printer spooler doesn't handle flags.
Spider network spooler doesn't spool.

NAME

passwd – change login password

SYNOPSIS

passwd name password

DESCRIPTION

The *password* becomes associated with the given login name. This can only be done by corresponding user or by the super-user. An explicit null argument ("") for the password argument removes any password.

FILES

/etc/passwd

SEE ALSO

login (I), passwd (V), crypt (III)

BUGS

NAME

pfe – print floating exception

SYNOPSIS

pfe

DESCRIPTION

Pfe examines the floating point exception register and prints a diagnostic for the last floating point exception.

SEE ALSO

signal (II)

BUGS

Since the system does not save the exception register in a core image file, the message refers to the last error encountered by anyone. Floating exceptions are therefore volatile.

NAME

pr − print file

SYNOPSIS

pr [−**h** *header*] [−*n*] [**+***n*] [−**w***n*] [−**l***n*] [−**t**] [−**s***c*] [−**m**] name . . .

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, and is thus usable as a filter.

Options apply to all following files but may be reset between files:

−*n* produce *n*-column output

+*n* begin printing with page *n*

−**h** treat the next argument as a header to be used instead of the file name

−**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

Interconsole messages via write(I) are forbidden during a *pr*.

FILES

/dev/tty? to suspend messages.

SEE ALSO

cat (I), cp (I)

DIAGNOSTICS

none; files not found are ignored

BUGS

NAME

prof – display profile data

SYNOPSIS

prof [**–v**] [**–a**] [**–l**] [file]

DESCRIPTION

Prof interprets the file *mon.out* produced by the *monitor* subroutine. Under default modes, the symbol table in the named object file *(a.out* default) is read and correlated with the *mon.out* profile file. For each external symbol, the percentage of time spent executing between that symbol and the next is printed (in decreasing order), together with the number of times that routine was called and the number of milliseconds per call.

If the **–a** option is used, all symbols are reported rather than just external symbols. If the **–l** option is used, the output is listed by symbol value rather than decreasing percentage. If the **–v** option is used, all printing is suppressed and a profile plot is produced on the 611 display.

In order for the number of calls to a routine to be tallied, the **–p** option of *cc* must have been given when the file containing the routine was compiled. This option also arranges for the *mon.out* file to be produced automatically.

FILES

mon.out for profile
a.out for namelist
/dev/vt0 for plotting

SEE ALSO

monitor (III), profil (II), cc (I)

BUGS

Beware of quantization errors.

NAME

ps – process status

SYNOPSIS

ps [**aklx**] [namelist]

DESCRIPTION

Ps prints certain indicia about active processes. The **a** flag asks for information about all processes with typewriters (ordinarily only one's own processes are displayed); **x** asks even about processes with no typewriter; **l** asks for a long listing. Ordinarily only the typewriter number (if not one's own), the process number, and an approximation to the command line are given. If the **k** flag 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.

The long listing is columnar and contains

The name of the process's control typewriter.

Flags associated with the process. 01: in core; 02: system process; 04: locked in code (e.g. for physical I/O); 10: being swapped; 20: being traced by another process.

The state of the process. 0: nonexistent; S: sleeping; W: waiting; R: running; Z: terminated; T: stopped.

The user ID of the process owner.

The process ID of the process; as in certain cults it is possible to kill a process if you know its true name.

The priority of the process; high numbers mean low priority.

The size in blocks of the core image of the process.

The event for which the process is waiting or sleeping; if blank, the process is running.

The command and its arguments.

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.

FILES

/unix	system namelist
/dev/mem	core memory
/usr/sys/core	alternate core file
/dev	searched to find swap device and typewriter names

SEE ALSO

kill (I)

BUGS

-

NAME

pwd – working directory name

SYNOPSIS

pwd

DESCRIPTION

Pwd prints the pathname of the working (current) directory.

SEE ALSO

chdir (I)

BUGS

NAME

rc – Ratfor compiler

SYNOPSIS

rc [**–c**] [**–r**] [**–f**] [**–v**] file ...

DESCRIPTION

Rc invokes the Ratfor preprocessor on a set of Ratfor source files. It accepts three types of arguments:

Arguments whose names end with '.r' are taken to be Ratfor source programs; they are preprocessed into Fortran and compiled. Each subroutine or function 'name' is placed on a separate file *name.f,* and its object code is left on *name.o.* The main routine is on *MAIN.f* and *MAIN.o;* block data subprograms go on *blockdata?.f* and *blockdata?.o.* The files resulting from a '.r' file are loaded into a single object file *file.o,* and the intermediate object and Fortran files are removed.

The following flags are interpreted by *rc.* See *ld (I)* for load-time flags.

–c Suppresses the loading phase of the compilation, as does any error in anything.

–f Save Fortran intermediate files. This is primarily for debugging.

–r Ratfor only; don't try to compile the Fortran. This implies **–f** and **–c.**

–v Don't list intermediate file names while compiling.

Arguments whose names end with '.f' are taken to be Fortran source programs; they are compiled in the normal manner. (Only one Fortran routine is allowed in a '.f' file.) Other arguments are taken to be either loader flag arguments, or Fortran-compatible object programs, typically produced by an earlier *rc* run, or perhaps libraries of Fortran-compatible routines. These programs, together with the results of any compilations specified, are loaded to produce an executable program with name **a.out.**

FILES

ratjunk	temporary
/usr/bin/ratfor	preprocessor
/usr/fort/fc1	Fortran compiler

SEE ALSO

''RATFOR – A Rational Fortran''.
fc(I) for Fortran error messages.

DIAGNOSTICS

Yes, both from *rc* itself and from Fortran.

BUGS

Limit of about 50 arguments, 10 block data files.

#define and #include lines in ''.f'' files are not processed.

NAME

rev – reverse lines of a file

SYNOPSIS

rev

DESCRIPTION

Rev copies the standard input to the standard output, reversing the order of characters in every line.

BUGS

NAME

rm − remove (unlink) files

SYNOPSIS

rm [**−f**] [**−r**] name ...

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, *rm* prints the file name and its mode, then reads a line from the standard input. If the line begins with **y**, the file is removed, otherwise it is not. The question is not asked if option **−f** was given or if the standard input is not a typewriter.

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. To remove directories *per se* see rmdir(I).

FILES

/etc/glob to implement the **−r** flag

SEE ALSO

rmdir (I)

BUGS

When *rm* removes the contents of a directory under the **−r** flag, full pathnames are not printed in diagnostics.

NAME

rmdir − remove directory

SYNOPSIS

rmdir dir ...

DESCRIPTION

Rmdir removes (deletes) directories. The directory must be empty (except for the standard entries '**.**' and '**..**', which *rmdir* itself removes). Write permission is required in the directory in which the directory to be removed appears.

BUGS

Needs a −**r** flag.
Actually, write permission in the directory's parent is *not* required.
Mildly unpleasant consequences can follow removal of your own or someone else's current directory.

NAME

roff − format text

SYNOPSIS

roff [**+***n*] [−*n*] [−**s**] [−**h**] file ...

DESCRIPTION

Roff formats text according to control lines embedded in the text in the given files. Encountering a nonexistent file terminates printing. Incoming interconsole messages are turned off during printing. The optional flag arguments mean:

+*n* Start printing at the first page with number *n*.

−*n* Stop printing at the first page numbered higher than *n*.

−**s** Stop before each page (including the first) to allow paper manipulation; resume on receipt of an interrupt signal.

−**h** Insert tabs in the output stream to replace spaces whenever appropriate.

Input consists of intermixed *text lines,* which contain information to be formatted, and *request lines,* which contain instructions about how to format it. Request lines begin with a distinguished *control character,* normally a period.

Output lines may be *filled* as nearly as possible with words without regard to input lineation. Line *breaks* may be caused at specified places by certain commands, or by the appearance of an empty input line or an input line beginning with a space.

The capabilities of *roff* are specified in the attached Request Summary. Numerical values are denoted there by n or +n, titles by t, and single characters by c. Numbers denoted +n may be signed + or −, in which case they signify relative changes to a quantity, otherwise they signify an absolute resetting. Missing n fields are ordinarily taken to be 1, missing t fields to be empty, and c fields to shut off the appropriate special interpretation.

Running titles usually appear at top and bottom of every page. They are set by requests like

.he ′part1′part2′part3′

Part1 is left justified, part2 is centered, and part3 is right justified on the page. Any % sign in a title is replaced by the current page number. Any nonblank may serve as a quote.

ASCII tab characters are replaced in the input by a *replacement character,* normally a space, according to the column settings given by a .ta command. (See .tr for how to convert this character on output.)

Automatic hyphenation of filled output is done under control of .hy. When a word contains a designated *hyphenation character,* that character disappears from the output and hyphens can be introduced into the word at the marked places only.

FILES

/usr/lib/suftab	suffix hyphenation tables
/tmp/rtm?	temporary

SEE ALSO

nroff (I), troff (I)

BUGS

Roff is the simplest of the runoff programs, but is utterly frozen.

REQUEST SUMMARY

Request	*Break*	*Initial*	*Meaning*
.ad	yes	yes	Begin adjusting right margins.
.ar	no	arabic	Arabic page numbers.
.br	yes	-	Causes a line break – the filling of the current line is stopped.
.bl n	yes	-	Insert of n blank lines, on new page if necessary.
.bp +n	yes	n=1	Begin new page and number it n; no n means '+1'.
.cc c	no	c=.	Control character becomes 'c'.
.ce n	yes	-	Center the next n input lines, without filling.
.de xx	no	-	Define parameterless macro to be invoked by request '.xx' (definition ends on line beginning '**..**').
.ds	yes	no	Double space; same as '.ls 2'.
.ef t	no	t=´´´´	Even foot title becomes t.
.eh t	no	t=´´´´	Even head title becomes t.
.fi	yes	yes	Begin filling output lines.
.fo	no	t=´´´´	All foot titles are t.
.hc c	no	none	Hyphenation character becomes 'c'.
.he t	no	t=´´´´	All head titles are t.
.hx	no	-	Title lines are suppressed.
.hy n	no	n=1	Hyphenation is done, if n=1; and is not done, if n=0.
.ig	no	-	Ignore input lines through a line beginning with '**..**'.
.in +n	yes	-	Indent n spaces from left margin.
.ix +n	no	-	Same as '.in' but without break.
.li n	no	-	Literal, treat next n lines as text.
.ll +n	no	n=65	Line length including indent is n characters.
.ls +n	yes	n=1	Line spacing set to n lines per output line.
.m1 n	no	n=2	Put n blank lines between the top of page and head title.
.m2 n	no	n=2	n blank lines put between head title and beginning of text on page.
.m3 n	no	n=1	n blank lines put between end of text and foot title.
.m4 n	no	n=3	n blank lines put between the foot title and the bottom of page.
.na	yes	no	Stop adjusting the right margin.
.ne n	no	-	Begin new page, if n output lines cannot fit on present page.
.nn +n	no	-	The next n output lines are not numbered.
.n1	no	no	Add 5 to page offset; number lines in margin from 1 on each page.
.n2 n	no	no	Add 5 to page offset; number lines from n; stop if n=0.
.ni +n	no	n=0	Line numbers are indented n.
.nf	yes	no	Stop filling output lines.
.nx filename	-	Change to input file 'filename'.	
.of t	no	t=´´´´	Odd foot title becomes t.
.oh t	no	t=´´´´	Odd head title becomes t.
.pa +n	yes	n=1	Same as '.bp'.
.pl +n	no	n=66	Total paper length taken to be n lines.
.po +n	no	n=0	Page offset. All lines are preceded by n spaces.
.ro	no	arabic	Roman page numbers.
.sk n	no	-	Produce n blank pages starting next page.
.sp n	yes	-	Insert block of n blank lines, except at top of page.
.ss	yes	yes	Single space output lines, equivalent to '.ls 1'.
.ta n n..		-	Pseudotab settings. Initial tab settings are columns 9 17 25 ...
.tc c	no	space	Tab replacement character becomes 'c'.
.ti +n	yes	-	Temporarily indent next output line n spaces.
.tr cdef..	no	-	Translate c into d, e into f, etc.
.ul n	no	-	Underline the letters and numbers in the next n input lines.

NAME

sh − shell (command interpreter)

SYNOPSIS

sh [**−t**] [**−c**] [name [arg1 ... [arg9]]]

DESCRIPTION

Sh is the standard command interpreter. It is the program which reads and arranges the execution of the command lines typed by most users. It may itself be called as a command to interpret files of commands. Before discussing the arguments to the Shell used as a command, the structure of command lines themselves will be given.

Commands. Each command is a sequence of non-blank command arguments separated by blanks. The first argument specifies the name of a command to be executed. Except for certain types of special arguments discussed below, the arguments other than the command name are passed without interpretation to the invoked command.

If the first argument is the name of an executable file, it is invoked; otherwise the string '/bin/' is prepended to the argument. (In this way most standard commands, which reside in '/bin', are found.) If no such command is found, the string '/usr' is further prepended (to give '/usr/bin/command') and another attempt is made to execute the resulting file. (Certain lesser-used commands live in '/usr/bin'.)

If a non-directory file has executable mode, but not the form of an executable program (does not begin with the proper magic number) then it is assumed to be an ASCII file of commands and a new Shell is created to execute it. See ''Argument passing'' below.

If the file cannot be found, a diagnostic is printed.

Command lines. One or more commands separated by '|' or '^' constitute a chain of *filters*. The standard output of each command but the last is taken as the standard input of the next command. Each command is run as a separate process, connected by pipes (see pipe(II)) to its neighbors. A command line contained in parentheses '( )' may appear in place of a simple command as a filter.

A *command line* consists of one or more pipelines separated, and perhaps terminated by '**;**' or '&'. The semicolon designates sequential execution. The ampersand causes the preceding pipeline to be executed without waiting for it to finish. The process id of such a pipeline is reported, so that it may be used if necessary for a subsequent *wait* or *kill.*

Termination Reporting. If a command (not followed by '&') terminates abnormally, a message is printed. (All terminations other than exit and interrupt are considered abnormal.) Termination reports for commands followed by '&' are given upon receipt of the first command subsequent to the termination of the command, or when a *wait* is executed. The following is a list of the abnormal termination messages:

Bus error
Trace/BPT trap
Illegal instruction
IOT trap
EMT trap
Bad system call
Quit
Floating exception
Memory violation
Killed
Broken Pipe

If a core image is produced, '− Core dumped' is appended to the appropriate message.

Redirection of I/O. There are three character sequences that cause the immediately following string to be interpreted as a special argument to the Shell itself. Such an argument may appear anywhere among the arguments of a simple command, or before or after a parenthesized command list, and is associated with that command or command list.

An argument of the form ‘<arg’ causes the file ‘arg’ to be used as the standard input (file descriptor 0) of the associated command.

An argument of the form ‘>arg’ causes file ‘arg’ to be used as the standard output (file descriptor 1) for the associated command. ‘Arg’ is created if it did not exist, and in any case is truncated at the outset.

An argument of the form ‘>>arg’ causes file ‘arg’ to be used as the standard output for the associated command. If ‘arg’ did not exist, it is created; if it did exist, the command output is appended to the file.

For example, either of the command lines

```
ls >junk; cat tail >>junk
( ls; cat tail ) >junk
```

creates, on file ‘junk’, a listing of the working directory, followed immediately by the contents of file ‘tail’.

Either of the constructs ‘>arg’ or ‘>>arg’ associated with any but the last command of a pipeline is ineffectual, as is ‘<arg’ in any but the first.

In commands called by the Shell, file descriptor 2 refers to the standard output of the Shell before any redirection. Thus filters may write diagnostics to a location where they have a chance to be seen.

Generation of argument lists. If any argument contains any of the characters ‘?’, ‘*’ or ‘[’, it is treated specially as follows. The current directory is searched for files which *match* the given argument.

The character ‘*’ in an argument matches any string of characters in a file name (including the null string).

The character ‘?’ matches any single character in a file name.

Square brackets ‘[...]’ specify a class of characters which matches any single file-name character in the class. Within the brackets, each ordinary character is taken to be a member of the class. A pair of characters separated by ‘–’ places in the class each character lexically greater than or equal to the first and less than or equal to the second member of the pair.

Other characters match only the same character in the file name.

For example, ‘*’ matches all file names; ‘?’ matches all one-character file names; ‘[ab]*.s’ matches all file names beginning with ‘a’ or ‘b’ and ending with ‘.s’; ‘?[zi–m]’ matches all two-character file names ending with ‘z’ or the letters ‘i’ through ‘m’.

If the argument with ‘*’ or ‘?’ also contains a ‘/’, a slightly different procedure is used: instead of the current directory, the directory used is the one obtained by taking the argument up to the last ‘/’ before a ‘*’ or ‘?’. The matching process matches the remainder of the argument after this ‘/’ against the files in the derived directory. For example: ‘/usr/dmr/a*.s’ matches all files in directory ‘/usr/dmr’ which begin with ‘a’ and end with ‘.s’.

In any event, a list of names is obtained which match the argument. This list is sorted into alphabetical order, and the resulting sequence of arguments replaces the single argument containing the ‘*’, ‘[’, or ‘?’. The same process is carried out for each argument (the resulting lists are *not* merged) and finally the command is called with the resulting list of arguments.

Quoting. The character ‘\’ causes the immediately following character to lose any special meaning it may have to the Shell; in this way ‘<’, ‘>’, and other characters meaningful to the Shell may be passed as part of arguments. A special case of this feature allows the continuation of commands onto more than one line: a new-line preceded by ‘\’ is translated into a blank.

Sequences of characters enclosed in double (") or single (´) quotes are also taken literally. For example:

```
ls | pr –h "My directory"
```

causes a directory listing to be produced by *ls,* and passed on to *pr* to be printed with the heading ‘My directory’. Quotes permit the inclusion of blanks in the heading, which is a single argument to *pr.*

Argument passing. When the Shell is invoked as a command, it has additional string processing capabilities. Recall that the form in which the Shell is invoked is

```
sh [ name [ arg1 ... [ arg9 ] ] ]
```

The *name* is the name of a file which is read and interpreted. If not given, this subinstance of the Shell continues to read the standard input file.

In command lines in the file (not in command input), character sequences of the form '$n', where *n* is a digit, are replaced by the *n*th argument to the invocation of the Shell (argn). '$0' is replaced by *name.*

The argument '–t,' used alone, causes *sh* to read the standard input for a single line, execute it as a command, and then exit. This facility replaces the older 'mini-shell.' It is useful for interactive programs which allow users to execute system commands.

The argument '–c' (used with one following argument) causes the next argument to be taken as a command line and executed. No new-line need be present, but new-line characters are treated appropriately. This facility is useful as an alternative to '–t' where the caller has already read some of the characters of the command to be executed.

End of file. An end-of-file in the Shell's input causes it to exit. A side effect of this fact means that the way to log out from UNIX is to type an EOT.

Special commands. The following commands are treated specially by the Shell.

chdir is done without spawning a new process by executing *sys chdir* (II).

login is done by executing /bin/login without creating a new process.

wait is done without spawning a new process by executing *sys wait* (II).

shift is done by manipulating the arguments to the Shell.

'**:**' is simply ignored.

Command file errors; interrupts. Any Shell-detected error, or an interrupt signal, during the execution of a command file causes the Shell to cease execution of that file.

Processes that are created with '&' ignore interrupts. Also if such a process has not redirected its input with a '<', its input is automatically redirected to the zero length file /dev/null.

FILES

/etc/glob, which interprets '*', '?', and '['.
/dev/null as a source of end-of-file.

SEE ALSO

'The UNIX Time-Sharing System', CACM, July, 1974, which gives the theory of operation of the Shell.
chdir (I), login (I), wait (I), shift (I)

BUGS

There is no way to redirect the diagnostic output.

NAME

shift – adjust Shell arguments

SYNOPSIS

shift

DESCRIPTION

Shift is used in Shell command files to shift the argument list left by 1, so that old **$2** can now be referred to by **$1** and so forth. *Shift* is useful to iterate over several arguments to a command file. For example, the command file

```
: loop
if $1x = x exit
pr –3 $1
shift
goto loop
```

prints each of its arguments in 3-column format.

Shift is executed within the Shell.

SEE ALSO

sh (I)

BUGS

-

NAME

size – size of an object file

SYNOPSIS

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.

BUGS

NAME

sleep – suspend execution for an interval

SYNOPSIS

sleep time

DESCRIPTION

Sleep suspends execution for *time* seconds. It is used to execute a command in a certain amount of time as in:

```
(sleep 105; command)&
```

Or to execute a command every so often as in this shell command file:

```
: loop
command
sleep 37
goto loop
```

SEE ALSO

sleep (II)

BUGS

Time must be less than 65536 seconds.

NAME

sort, usort – sort or merge files

SYNOPSIS

sort [–**abdnrt***x*] [**+***pos* [–*pos*]] . . . [–**mo**] [name] . . .
usort [–**umo**] [name] . . .

DESCRIPTION

Sort sorts all the named files together and writes the result on the standard output. The name '–' means the standard input. The standard input is also used if no input file names are given. Thus *sort* may be used as a filter.

The default sort key is an entire line. Default ordering is lexicographic in ASCII collating sequence, except that lower-case letters are considered the same as the corresponding upper-case letters. Non-ASCII bytes are ignored. The ordering is affected by the flags –**abdnrt**, one or more of which may appear:

a Do not map lower case letters.

b Leading blanks (spaces and tabs) are not included in fields.

d 'Dictionary' order: only letters, digits and blanks are significant in ASCII comparisons.

n An initial numeric string, consisting of optional minus sign, digits and optionally included decimal point, is sorted by arithmetic value.

r Reverse the sense of comparisons.

t*x* Tab character between fields is *x*.

Selected parts of the line, specified by **+***pos* and –*pos*, may be used as sort keys. *Pos* has the form *m.n,* where *m* specifies a number of fields to skip, and *n* a number of characters to skip further into the next field. A missing is taken to be 0. **+***pos* denotes the beginning of the key; –*pos* denotes the first position after the key (end of line by default). The ordering rule may be overridden for a particular key by appending one or more of the flags **abdnr** to **+***pos*.

When no tab character has been specified, a field consists of nonblanks and any preceding blanks. Under the –**b** flag, leading blanks are excluded from a field. When a tab character has been specified, a field is a string ending with a tab character.

When keys are specified, later keys are compared only when all earlier ones compare equal. Lines that compare equal are ordered with all bytes significant.

These flag arguments are also understood:

–**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, except under the merge flag –**m**.

Usort is a somewhat specialized version of *sort* which accepts no collating sequence options: order is always plain ASCII. It also strips out the second and following copies of duplicated lines. A *u* flag prevents this stripping. *Usort* also understands the *m* and *o* options in the same way as *sort.*

FILES

/usr/tmp/stm???

BUGS

NAME

spell − find spelling errors

SYNOPSIS

spell [−**v**] file ...

DESCRIPTION

Spell collects the words from the named documents, and looks them up in a dictionary. The words not found are printed on the standard output. Words which are reasonable transformations of dictionary entries (e.g. a dictionary entry plus *s*) are not printed. If no files are given, the input is from the standard input.

If the −**v** flag is given, all words which are not literally in the dictionary are printed; those which can be transformed to lie in the dictionary are so marked, and the others are marked with asterisks.

The process takes several minutes.

FILES

/usr/lib/w2006, /usr/dict/words, /usr/lib/spell[123]

SEE ALSO

typo (I)

BUGS

Because of the mapping into lower case and the stripping of special characters, words may be hard to locate in the original text.

The escape sequences of troff (I) are not correctly recognized.

More suffixes, and perhaps some prefixes, should be added.

The dictionary cannot be distributed because of copyright limitations.

NAME

split – split a file into pieces

SYNOPSIS

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.

BUGS

NAME

strip − remove symbols and relocation bits

SYNOPSIS

strip name ...

DESCRIPTION

Strip removes the symbol table and relocation bits ordinarily attached to the output of the assembler and loader. This is useful to save space after a program has been debugged.

The effect of *strip* is the the same as use of the **−s** option of *ld.*

FILES

/tmp/stm? temporary file

SEE ALSO

ld (I), as (I)

BUGS

NAME

stty − set typewriter options

SYNOPSIS

stty [option ...]

DESCRIPTION

Stty sets certain I/O options on the current output typewriter. 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'
−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*.
kill *c* set kill character to *c*.
cr0 cr1 cr2 cr3
select style of delay for carriage return (see below)
nl0 nl1 nl2 nl3
select style of delay for linefeed (see below)
tab0 tab1 tab2 tab3
select style of delay for tab (see below)
ff0 ff1
select style of delay for form feed (see below)
tty33 set all modes suitable for Teletype model 33
tty37 set all modes suitable for Teletype model 37
vt05 set all modes suitable for DEC VT05 terminal
tn300 set all modes suitable for GE Terminet 300
ti700 set all modes suitable for Texas Instruments 700 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 typewriter baud rate to the number given, if possible. (These are the speeds supported by the DH-11 interface).

The various delay algorithms are tuned to various kinds of terminals. In general the specifications ending in '0' mean no delay for the corresponding character.

SEE ALSO

stty (II)

BUGS

NAME

tee – pipe fitting

SYNOPSIS

tee [name ...]

DESCRIPTION

Tee transcribes the standard input to the standard output and makes copies in the named files.

BUGS

NAME

time – time a command

SYNOPSIS

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.

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.

BUGS

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

tp − manipulate DECtape and magtape

SYNOPSIS

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 zeroed before beginning. Usable only with **r** and **u.** This option is assumed with magtape since it is impossible to selectively overwrite magtape.

f causes new entries on tape to be 'fake' in that no data is present for these entries. Such fake entries cannot be extracted. Usable only with **r** and **u.**

i Errors reading and writing the tape are noted, but no action is taken. Normally, errors cause a return to the command level.

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?

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.

BUGS

A single file with several links to it is treated like several files.

NAME

tr – transliterate

SYNOPSIS

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.* 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 001 through 377 octal. –**d** deletes all input characters in *string1.* –**s** squeezes all strings of repeated output characters that are in *string2* to single characters.

The following abbreviation conventions may be used to introduce ranges of characters or repeated characters into the strings:

[*a –b* **]** stands for the string of characters whose ascii codes run from character *a* to character *b.*

[*a* ***n* **]**, where *n* is an integer or empty, stands for *n*-fold repetition of character *a. n* is taken to be octal or decimal according as its first digit is or is not zero. A zero or missing *n* is taken to be huge; this facility is useful for padding *string2.*

The escape character '\' may be used as in *sh* to remove special meaning from any character in a string. In addition, '\' followed by 1, 2 or 3 octal digits stands for the character whose ascii code is given by those digits.

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 strings are quoted to protect the special characters from interpretation by the Shell; 012 is the ascii code for newline.

```
tr –cs "[A–Z][a–z]" "[\012*]" <file1 >file2
```

SEE ALSO

sh (I), ed (I), ascii (V)

BUGS

Won't handle ascii NUL in *string1* or *string2;* always deletes NUL from input.

NAME

troff − format text

SYNOPSIS

troff [**+***n*] [−*n*] [**−s***n*] [**−n***n*] [**−r***an*] [**−m***name*] [**−t**] [**−f**] [**−w**] [**−a**] [**−p***n*] files

DESCRIPTION

Troff formats text for a Graphic Systems phototypesetter according to control lines embedded in the text files. It reads the standard input if no file arguments are given. An argument of just "−" refers to the standard input. The non-file option arguments are interpreted as follows:

+*n*	Commence typesetting at the first page numbered *n* or larger.
−*n*	Stop after page *n.*
−s*n*	Print output in groups of *n* pages, stopping the typesetter after each group.
−n*n*	First generated (not necessarily printed) page is given the number *n;* simulates ".pn *n*".
−r*an*	Set number register *a* to the value *n.*
−m*name*	Prepends a standard macro file; simulates ".so /usr/lib/tmac.*name*".
−t	Place output on standard output instead of the phototypesetter.
−f	Refrain from feeding out paper and stopping the phototypesetter at the end.
−w	Wait until phototypesetter is available, if currently busy.
−a	Send a printable approximation of the results to the standard output.
−p*n*	Print all characters with point-size *n* while retaining all prescribed spacings and motions.

FILES

/usr/lib/suftab	suffix hyphenation tables
/tmp/rtm?	temporary
/usr/lib/tmac.*	standard macro files

SEE ALSO

TROFF User's Manual (internal memorandum).
TROFF Made Trivial (internal memorandum).
nroff (I), eqn (I), catsim (VI)

BUGS

NAME

tty – get typewriter name

SYNOPSIS

tty

DESCRIPTION

Tty gives the name of the user's typewriter in the form 'tty*n*' for *n* a digit or letter. The actual path name is then '/dev/tty*n*'.

DIAGNOSTICS

'not a tty' if the standard input file is not a typewriter.

BUGS

NAME

typo – find possible typos

SYNOPSIS

typo [**–1**] [**–n**] file ...

DESCRIPTION

Typo hunts through a document for unusual words, typographic errors, and *hapax legomena* and prints them on the standard output.

The words used in the document are printed out in decreasing order of peculiarity along with an index of peculiarity. An index of 10 or more is considered peculiar. Printing of certain very common English words is suppressed.

The statistics for judging words are taken from the document itself, with some help from known statistics of English. The **–n** option suppresses the help from English and should be used if the document is written in, for example, Urdu.

The **–1** option causes the final output to appear in a single column instead of three columns. The normal header and pagination is also suppressed.

Roff (I) and nroff (I) control lines are ignored. Upper case is mapped into lower case. Quote marks, vertical bars, hyphens, and ampersands within words are equivalent to spaces. Words hyphenated across lines are put back together.

FILES

/tmp/ttmp??
/usr/lib/salt
/usr/lib/w2006

BUGS

Because of the mapping into lower case and the stripping of special characters, words may be hard to locate in the original text.

The escape sequences of troff (I) are not correctly recognized.

NAME

uniq – report repeated lines in a file

SYNOPSIS

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(I). 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 (I), comm (I)

BUGS

NAME

wait – await completion of process

SYNOPSIS

wait

DESCRIPTION

Wait until all processes started with **&** have completed, and report on abnormal terminations.

Because *sys wait* must be executed in the parent process, the Shell itself executes *wait,* without creating a new process.

SEE ALSO

sh (I)

BUGS

After executing *wait* you are committed to waiting until termination, because interrupts and quits are ignored by all processes concerned. The only out, if the process does not terminate, is to *kill* it from another terminal or to hang up.

-

NAME

wc – word count

SYNOPSIS

wc [name ...]

DESCRIPTION

Wc counts lines and words in the named files, or in the standard input if no name appears. A word is a maximal string of printing characters delimited by spaces, tabs or newlines. All other characters are simply ignored.

BUGS

NAME

who − who is on the system

SYNOPSIS

who [who-file] [**am I**]

DESCRIPTION

Who, without an argument, lists the name, typewriter channel, 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, typewriter 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, *who* behaves as if it had no arguments except for restricting the printout to the line for the current typewriter. Thus 'who am I' (and also 'who are you') tells you who you are logged in as.

FILES

/etc/utmp

SEE ALSO

login (I), init (VIII)

BUGS

NAME

write − write to another user

SYNOPSIS

write user [ttyno]

DESCRIPTION

Write copies lines from your typewriter to that of another user. When first called, it sends the message

message from yourname...

The recipient of the message should write back at this point. Communication continues until an end of file is read from the typewriter 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 *ttyno* argument may be used to indicate the last character of the appropriate typewriter 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 *roff* and *pr,* 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 (I), who (I), mail (I)

BUGS

NAME

yacc − yet another compiler-compiler

SYNOPSIS

yacc [−**vor**] [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 is *y.tab.c,* which must be compiled by the C compiler and loaded with any other routines required (perhaps a lexical analyzer) and the Yacc library:

```
cc y.tab.c other.o −ly
```

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.

The −**o** flag calls an optimizer for the tables; the optimized tables, with parser included, appear on file *y.tab.c*

The −**r** flag causes Yacc to accept grammars with Ratfor actions, and produce Ratfor output on *y.tab.r;* −**r** implies the −**o** flag. Typical usage is then

```
rc y.tab.r other.o
```

SEE ALSO

''LR Parsing'', by A. V. Aho and S. C. Johnson, Computing Surveys, June, 1974. ''The YACC Compiler-compiler'', internal memorandum.

AUTHOR

S. C. Johnson

FILES

y.output	
y.tab.c	
y.tab.r	when ratfor output is obtained
yacc.tmp	when optimizer is called
/lib/liby.a	runtime library for compiler
/usr/yacc/fpar.r	ratfor parser
/usr/yacc/opar.c	parser for optimized tables
/usr/yacc/yopti	optimizer postpass

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.

BUGS

Because file names are fixed, at most one Yacc process can be active in a given directory at a time.

NAME

break, brk, sbrk – change core allocation

SYNOPSIS

(break = 17.)
sys break; addr

char *brk(addr)

char *sbrk(incr)

DESCRIPTION

Break 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). 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.

From C, *brk* will set the break to *addr.* The old break is returned.

In the alternate entry *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 (II), alloc (III), end (III)

DIAGNOSTICS

The c-bit is set if the program requests more memory than the system limit or if more than 8 segmentation registers would be required to implement the break. From C, −1 is returned for these errors.

BUGS

Setting the break in the range 0177700 to 0177777 is the same as setting it to zero.

NAME

chdir – change working directory

SYNOPSIS

(chdir = 12.)
sys chdir; dirname

chdir(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.

SEE ALSO

chdir (I)

DIAGNOSTICS

The error bit (c-bit) is set if the given name is not that of a directory or is not readable. From C, a –1 returned value indicates an error, 0 indicates success.

NAME

chmod – change mode of file

SYNOPSIS

(chmod = 15.)
sys chmod; name; mode

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:

4000 set user ID on execution
2000 set group ID on execution
1000 save text image after execution
0400 read by owner
0200 write by owner
0100 execute (search on directory) by owner
0070 read, write, execute (search) by group
0007 read, write, execute (search) by others

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 (I)

DIAGNOSTIC

Error bit (c-bit) set if *name* cannot be found or if current user is neither the owner of the file nor the super-user. From C, a –1 returned value indicates an error, 0 indicates success.

NAME

chown − change owner and group of a file

SYNOPSIS

(chmod = 16.)
sys chown; name; owner

chown(name, owner)
char *name;

DESCRIPTION

The file whose name is given by the null-terminated string pointed to by *name* has its owner and group changed to the low and high bytes of *owner* respectively. 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 (VIII), chgrp (VIII), passwd (V)

DIAGNOSTICS

The error bit (c-bit) is set on illegal owner changes. From C a −1 returned value indicates error, 0 indicates success.

NAME

close − close a file

SYNOPSIS

(close = 6.)
(file descriptor in r0)
sys close

close(fildes)

DESCRIPTION

Given a file descriptor such as returned from an *open, creat,* or *pipe* call, *close* closes the associated file. A close of all files is automatic on *exit,* but since processes are limited to 15 simultaneously open files, *close* is necessary for programs which deal with many files.

SEE ALSO

creat (II), open (II), pipe (II)

DIAGNOSTICS

The error bit (c-bit) is set for an unknown file descriptor. From C a −1 indicates an error, 0 indicates success.

NAME

creat − create a new file

SYNOPSIS

(creat = 8.)
sys creat; name; mode
(file descriptor in r0)

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.* See *chmod* (II) 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 (in r0).

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 (II), close (II), stat (II)

DIAGNOSTICS

The error bit (c-bit) may be set 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.

From C, a −1 return indicates an error.

-

NAME

csw – read console switches

SYNOPSIS

(csw = 38.; not in assembler)

sys csw

getcsw( )

DESCRIPTION

The setting of the console switches is returned (in r0).

-

NAME

dup – duplicate an open file descriptor

SYNOPSIS

(dup = 41.; not in assembler)
(file descriptor in r0)
sys dup

dup(fildes)
int fildes;

DESCRIPTION

Given a file descriptor returned from an *open, pipe,* or *creat* call, *dup* will allocate another file descriptor synonymous with the original. The new file descriptor is returned in r0.

Dup is used more to reassign the value of file descriptors than to genuinely duplicate a file descriptor. Since the algorithm to allocate file descriptors returns the lowest available value, combinations of *dup* and *close* can be used to manipulate file descriptors in a general way. This is handy for manipulating standard input and/or standard output.

SEE ALSO

creat (II), open (II), close (II), pipe (II)

DIAGNOSTICS

The error bit (c-bit) is set if: the given file descriptor is invalid; there are already too many open files. From C, a −1 returned value indicates an error.

NAME

exec, execl, execv − execute a file

SYNOPSIS

(exec = 11.)
sys exec; name; args
...
name: <...\0>
...
args: arg0; arg1; ...; 0
arg0: <...\0>
arg1: <...\0>
...

execl(name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ..., *argn;

execv(name, argv)
char *name;
char *argv[ ];

DESCRIPTION

Exec overlays the calling process with the named file, then transfers to the beginning of the core image of the file. There can be no return from the file; the calling core image is lost.

Files remain open across *exec* calls. Ignored signals remain ignored across *exec,* but signals that are caught 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 form of this call differs somewhat depending on whether it is called from assembly language or C; see below for the C version.

The first argument to *exec* is a pointer to the name of the file to be executed. The second is the address of a null-terminated list of pointers to arguments to be passed to the file. Conventionally, the first argument is the name of the file. Each pointer addresses a string terminated by a null byte.

Once the called file starts execution, the arguments are available as follows. 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. The arguments are placed as high as possible in core.

```
 sp→    nargs
        arg0
        ...
        argn
        −1

 arg0:  <arg0\0>
        ...
 argn:  <argn\0>
```

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; as in the basic call, 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 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)
int argc;
char **argv;
```

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 not directly usable in another *execv,* since *argv[argc]* is −1 and not 0.

SEE ALSO

fork (II)

DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not have a valid header (407, 410, or 411 octal as first word), if maximum memory is exceeded, or if the arguments require more than 512 bytes a return from *exec* constitutes the diagnostic; the error bit (c-bit) is set. Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed. From C the returned value is −1.

BUGS

Only 512 characters of arguments are allowed.

NAME

exit – terminate process

SYNOPSIS

(exit = 1.)
(status in r0)
sys exit

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 byte of r0 (resp. the argument to *exit*) is available as status to the parent process.

This call can never return.

SEE ALSO

wait (II)

DIAGNOSTICS

None.

NAME

fork − spawn new process

SYNOPSIS

(fork = 2.)
sys fork
(new process return)
(old process return)

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 return location and the fact that r0 in the old (parent) process contains the process ID of the new (child) process. This process ID is used by *wait*.

The two returning processes share all open files that existed before the call. In particular, this is the way that standard input and output files are passed and also how pipes are set up.

From C, the child process receives a 0 return, and the parent receives a non-zero number which is the process ID of the child; a return of −1 indicates inability to create a new process.

SEE ALSO

wait (II), exec (II)

DIAGNOSTICS

The error bit (c-bit) is set in the old process if a new process could not be created because of lack of process space. From C, a return of −1 (not just negative) indicates an error.

NAME

fstat − get status of open file

SYNOPSIS

(fstat = 28.)
(file descriptor in r0)
sys fstat; buf

fstat(fildes, buf)
struct inode *buf;

DESCRIPTION

This call is identical to *stat,* except that it operates on open files instead of files given by name. It is most often used to get the status of the standard input and output files, whose names are unknown.

SEE ALSO

stat (II)

DIAGNOSTICS

The error bit (c-bit) is set if the file descriptor is unknown; from C, a −1 return indicates an error, 0 indicates success.

NAME

getgid – get group identifications

SYNOPSIS

(getgid = 47.; not in assembler)
sys getgid

getgid( )

DESCRIPTION

Getgid returns a word (in r0), the low byte of which contains the real group ID of the current process. The high byte contains the effective group ID of the current process. The real group ID identifies the group of the person who is logged in, in contradistinction to the effective group ID, which determines his access permission at the moment. It is thus useful to programs which operate using the ‘‘set group ID’’ mode, to find out who invoked them.

SEE ALSO

setgid (II)

DIAGNOSTICS

–

-

NAME

getpid – get process identification

SYNOPSIS

(getpid = 20.; not in assembler)
sys getpid
(pid in r0)

getpid( )

DESCRIPTION

Getpid returns the process ID of the current process. Most often it is used to generate uniquely-named temporary files.

SEE ALSO

–

DIAGNOSTICS

–

NAME

getuid – get user identifications

SYNOPSIS

(getuid = 24.)
sys getuid

getuid( )

DESCRIPTION

Getuid returns a word (in r0), the low byte of which contains the real user ID of the current process. The high byte contains the effective user ID of the current process. 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.

SEE ALSO

setuid (II)

DIAGNOSTICS

–

NAME

gtty − get typewriter status

SYNOPSIS

(gtty = 32.)
(file descriptor in r0)
sys gtty; arg
...
arg: .=.+6

gtty(fildes, arg)
int arg[3];

DESCRIPTION

Gtty stores in the three words addressed by *arg* the status of the typewriter whose file descriptor is given in r0 (resp. given as the first argument). The format is the same as that passed by *stty.*

SEE ALSO

stty (II)

DIAGNOSTICS

Error bit (c-bit) is set if the file descriptor does not refer to a typewriter. From C, a −1 value is returned for an error, 0, for a successful call.

NAME

indir – indirect system call

SYNOPSIS

(indir = 0.; not in assembler)
sys indir; syscall

DESCRIPTION

The system call at the location *syscall* 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, the executing process will get a fault.

SEE ALSO

–

DIAGNOSTICS

–

INTRODUCTION TO SYSTEM CALLS

Section II of this manual lists all the entries into the system. In most cases two calling sequences are specified, one of which is usable from assembly language, and the other from C. Most of these calls have an error return. From assembly language an erroneous call is always indicated by turning on the c-bit of the condition codes. 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.

From C, an error condition is indicated by an otherwise impossible returned value. Almost always this is −1; the individual sections specify the details.

In both cases an error number is also available. In assembly language, this number is returned in r0 on erroneous calls. From C, the external variable *errno* is set to the error number. *Errno* is not cleared on successful calls, so it should be tested only after an error has occurred. There is a table of messages associated with each error, and a routine for printing the message. See *perror (III)*.

The possible error numbers are not recited with each writeup in section II, since many errors are possible for most of the calls. Here is a list of the error numbers, their names inside the system (for the benefit of system-readers), and the messages available using *perror*. A short explanation is also provided.

0 – (unused)

1 EPERM Not owner and not super-user
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner. 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* 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 which 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 512 bytes (counting the null at the end of each argument) 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 one of the magic numbers 407 or 410.

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 which is open on-

ly 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 and no more processes can for the moment be created.

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 is such as to require more than the existing 8 segmentation registers.

13 EACCES Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

14 – (unused)

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 open file or some process's current directory.

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: currently, dismounting a non-mounted device, mentioning an unknown signal in *signal,* and giving an unknown request in *stty* to the TIU special file.

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
Only 15 files can be open per process.

25 ENOTTY Not a typewriter
The file mentioned in *stty* or *gtty* is not a typewriter or one of the other devices to which these calls apply.

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program which 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
An attempt to make a file larger than the maximum of 32768 blocks.

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 Seek on pipe
A *seek* 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 127 links to a file.

32 EPIPE Write on 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.

NAME

kill − send signal to a process

SYNOPSIS

(kill = 37.; not in assembler)
(process number in r0)
sys kill; sig

kill(pid, sig);

DESCRIPTION

Kill sends the signal *sig* to the process specified by the process number in r0. See signal (II) 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 which have the same controlling typewriter and user ID.

In no case is it possible for a process to kill itself.

SEE ALSO

signal (II), kill (I)

DIAGNOSTICS

The error bit (c-bit) is set 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. From C, −1 is returned.

NAME

link – link to a file

SYNOPSIS

(link = 9.)
sys link; name1; name2

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

link (I), unlink (II)

DIAGNOSTICS

The error bit (c-bit) is set 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 more than 127 links are made. From C, a –1 return indicates an error, a 0 return indicates success.

NAME

mknod − make a directory or a special file

SYNOPSIS

(mknod = 14.; not in assembler)
sys mknod; name; mode; addr

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 first physical address of the file is initialized from *addr.* Note that in the case of a directory, *addr* should be 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 (I), mknod (VIII), fs (V)

DIAGNOSTICS

Error bit (c-bit) is set if the file already exists or if the user is not the super-user. From C, a −1 value indicates an error.

NAME

mount – mount file system

SYNOPSIS

(mount = 21.)
sys mount; special; name; rwflag

mount(special, name, rwflag)
char *special, *name;

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. 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.

SEE ALSO

mount (VIII), umount (II)

DIAGNOSTICS

Error bit (c-bit) set if: *special* is inaccessible or not an appropriate file; *name* does not exist; *special* is already mounted; *name* is in use; there are already too many file systems mounted.

BUGS

–

NAME

nice – set program priority

SYNOPSIS

(nice = 34.)
(priority in r0)
sys nice

nice(priority)

DESCRIPTION

The scheduling *priority* of the process is changed to the argument. Positive priorities get less service than normal; 0 is default. Only the super-user may specify a negative priority. The valid range of *priority* is 20 to −220. The value of 4 is recommended to users who wish to execute long-running programs without flak from the administration.

The effect of this call is passed to a child process by the *fork* system call. The effect can be cancelled by another call to *nice* with a *priority* of 0.

The actual running priority of a process is the *priority* argument plus a number that ranges from 100 to 119 depending on the cpu usage of the process.

SEE ALSO

nice (I)

DIAGNOSTICS

The error bit (c-bit) is set if the user requests a *priority* outside the range of 0 to 20 and is not the super-user.

NAME

open − open for reading or writing

SYNOPSIS

(open = 5.)
sys open; name; mode
(file descriptor in r0)

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 returned file descriptor should be saved for subsequent calls to *read, write,* and *close.*

SEE ALSO

creat (II), read (II), write (II), close (II)

DIAGNOSTICS

The error bit (c-bit) is set 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. From C, a −1 value is returned on an error.

NAME

pipe – create an interprocess channel

SYNOPSIS

(pipe = 42.)
sys pipe
(read file descriptor in r0)
(write file descriptor in r1)

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 returned in r1 (resp. fildes[1]), up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor returned in r0 (resp. fildes[0]) will pick up the 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) return an end-of-file. Write calls under similar conditions generate a fatal signal (signal (II)); if the signal is ignored, an error is returned on the write.

SEE ALSO

sh (I), read (II), write (II), fork (II)

DIAGNOSTICS

The error bit (c-bit) is set if too many files are already open. From C, a −1 returned value indicates an error. A signal is generated if a write on a pipe with only one end is attempted.

BUGS

NAME

profil – execution time profile

SYNOPSIS

(profil = 44.; not in assembler)
sys profil; buff; bufsiz; offset; scale

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: 177777(8) gives a 1-1 mapping of pc's to words in *buff;* 77777(8) maps each pair of instruction words together. 2(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 also turned off when an *exec* is executed but remains on in child and parent both after a *fork.*

SEE ALSO

monitor (III), prof (I)

DIAGNOSTICS

–

NAME

ptrace − process trace

SYNOPSIS

(ptrace = 26.; not in assembler)
(data in r0)
sys ptrace; pid; addr; request
(value in r0)

ptrace(request, pid, addr, data);

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, but it should be adaptable for simulation of non-UNIX environments. 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 (II) for the list. Then the traced process enters a stopped state and its parent is notified via *wait* (II). 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 (in r0). Request 1 indicates the data space (normally used); 2 indicates the instruction space (when I and D space are separated). *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. Request 4 specifies data space (normally used), 5 specifies instruction space. Attempts to write in pure procedure result in termination of the child, instead of going through or causing an error for the parent.

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 as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal which caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop.

8 The traced process terminates.

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* (II) calls.

SEE ALSO

wait (II), signal (II), cdb (I)

DIAGNOSTICS

From assembler, the c-bit (error bit) is set on errors; from C, −1 is returned and *errno* has the error code.

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.

Also, it should be possible to stop a process on occurrence of a system call; in this way a completely controlled environment could be provided.

NAME

read − read from file

SYNOPSIS

(read = 3.)
(file descriptor in r0)
sys read; buffer; nbytes

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 (in r0).

If the returned value is 0, then end-of-file has been reached.

SEE ALSO

open (II), creat (II), dup (II), pipe (II)

DIAGNOSTICS

As mentioned, 0 is returned when the end of the file has been reached. If the read was otherwise unsuccessful the error bit (c-bit) is set. Many conditions can generate an error: physical I/O errors, bad buffer address, preposterous *nbytes,* file descriptor not that of an input file. From C, a −1 return indicates the error.

NAME

seek – move read/write pointer

SYNOPSIS

(seek = 19.)
(file descriptor in r0)
sys seek; offset; ptrname

seek(fildes, offset, ptrname)

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 *ptrname* is 0, the pointer is set to *offset.*

if *ptrname* is 1, the pointer is set to its current location plus *offset.*

if *ptrname* is 2, the pointer is set to the size of the file plus *offset.*

if *ptrname* is 3, 4 or 5, the meaning is as above for 0, 1 and 2 except that the offset is multiplied by 512.

If *ptrname* is 0 or 3, *offset* is unsigned, otherwise it is signed.

SEE ALSO

open (II), creat (II)

DIAGNOSTICS

The error bit (c-bit) is set for an undefined file descriptor. From C, a −1 return indicates an error.

NAME

setgid – set process group ID

SYNOPSIS

(setgid = 46.; not in assembler)
(group ID in r0)
sys setgid

setgid(gid)

DESCRIPTION

The group ID of the current process is set to the argument. Both the effective and the real group ID are set. This call is only permitted to the super-user or if the argument is the real group ID.

SEE ALSO

getgid (II)

DIAGNOSTICS

Error bit (c-bit) is set as indicated; from C, a −1 value is returned.

NAME

setuid – set process user ID

SYNOPSIS

(setuid = 23.)
(user ID in r0)
sys setuid

setuid(uid)

DESCRIPTION

The user ID of the current process is set to the argument. Both the effective and the real user ID are set. This call is only permitted to the super-user or if the argument is the real user ID.

SEE ALSO

getuid (II)

DIAGNOSTICS

Error bit (c-bit) is set as indicated; from C, a –1 value is returned.

NAME

signal – catch or ignore signals

SYNOPSIS

(signal = 48.)
sys signal; sig; label
(old value in r0)

signal(sig, func)
int (*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 this call allows them either to be ignored or to cause an interrupt to a specified location. Here is the list of signals:

1	hangup
2	interrupt
3*	quit
4*	illegal instruction (not reset when caught)
5*	trace trap (not reset when caught)
6*	IOT instruction
7*	EMT instruction
8*	floating point exception
9	kill (cannot be caught or ignored)
10*	bus error
11*	segmentation violation
12*	bad argument to system call
13	write on a pipe with no one to read it

In the assembler call, if *label* is 0, the process is terminated when the signal occurs; this is the default action. 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. Except as indicated, a signal is reset to 0 after being caught. Thus if it is desired to catch every such signal, the catching routine must issue another *signal* call.

In C, if *func* is 0, the default action for signal *sig* (termination) is reinstated. If *func* is 1, the signal is ignored. If *func* is non-zero and even, it is assumed to be the address of a function entry point. When the signal occurs, the function will be called. A return from the function will continue the process at the point it was interrupted. As in the assembler call, *signal* must in general be called again to catch subsequent signals.

When a caught signal occurs during certain system calls, the call terminates prematurely. In particular this can occur during a *read* or *write* on a slow device (like a typewriter; but not a file); and during or *wait*. 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 a characteristic error status. The user's program may then, if it wishes, re-execute the call.

The starred signals in the list above cause a core image if not caught or ignored.

The value of the call is the old action defined for the signal.

After a *fork* (II) the child inherits all signals. *Exec* (II) resets all caught signals to default action.

SEE ALSO

kill (I), kill (II), ptrace (II), reset (III)

DIAGNOSTICS

The error bit (c-bit) is set if the given signal is out of range. In C, a −1 indicates an error; 0 indicates success.

BUGS

NAME

sleep − stop execution for interval

SYNOPSIS

(sleep = 35.; not in assembler)
(seconds in r0)
sys sleep

sleep(seconds)

DESCRIPTION

The current process is suspended from execution for the number of seconds specified by the argument.

SEE ALSO

sleep (I)

DIAGNOSTICS

−

NAME

stat – get file status

SYNOPSIS

(stat = 18.)
sys stat; name; buf

stat(name, buf)
char *name;
struct inode *buf;

DESCRIPTION

Name points to a null-terminated string naming a file; *buf* is the address of a 36(10) byte 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 readable. After *stat, buf* has the following structure (starting offset given in bytes):

```
struct inode {
        char    minor;          /* +0: minor device of i-node */
        char    major;          /* +1: major device */
        int     inumber;        /* +2 */
        int     flags;          /* +4: see below */
        char    nlinks;         /* +6: number of links to file */
        char    uid;            /* +7: user ID of owner */
        char    gid;            /* +8: group ID of owner */
        char    size0;          /* +9: high byte of 24-bit size */
        int     size1;          /* +10: low word of 24-bit size */
        int     addr[8];        /* +12: block numbers or device number */
        int     actime[2];      /* +28: time of last access */
        int     modtime[2];     /* +32: time of last modification */
};
```

The flags are as follows:

```
100000  i-node is allocated
060000  2-bit file type:
        000000  plain file
        040000  directory
        020000  character-type special file
        060000  block-type special file.
010000  large file
004000  set user-ID on execution
002000  set group-ID on execution
001000  save text image after execution
000400  read (owner)
000200  write (owner)
000100  execute (owner)
000070  read, write, execute (group)
000007  read, write, execute (others)
```

SEE ALSO

ls (I), fstat (II), fs (V)

DIAGNOSTICS

Error bit (c-bit) is set if the file cannot be found. From C, a –1 return indicates an error.

NAME

stime – set time

SYNOPSIS

(stime = 25.)
(time in r0-r1)
sys stime

stime(tbuf)
int tbuf[2];

DESCRIPTION

Stime sets the system's idea of the time and date. Time is measured in seconds from 0000 GMT Jan 1 1970. Only the super-user may use this call.

SEE ALSO

date (I), time (II), ctime (III)

DIAGNOSTICS

Error bit (c-bit) set if user is not the super-user.

NAME

stty – set mode of typewriter

SYNOPSIS

(stty = 31.)
(file descriptor in r0)
sys stty; arg
...
arg: .byte ispeed, ospeed; .byte erase, kill; mode

```
stty(fildes, arg)
struct {
        char    ispeed, ospeed;
        char    erase, kill;
        int     mode;
} *arg;
```

DESCRIPTION

Stty sets mode bits and character speeds for the typewriter whose file descriptor is passed in r0 (resp. is the first argument to the call). First, the system delays until the typewriter is quiescent. The input and output speeds are set from the first two bytes of the argument structure as indicated by the following table, which corresponds to the speeds supported by the DH-11 interface. If DC-11, DL-11 or KL-11 interfaces are used, impossible speed changes are ignored.

0	(hang up dataphone)
1	50 baud
2	75 baud
3	110 baud
4	134.5 baud
5	150 baud
6	200 baud
7	300 baud
8	600 baud
9	1200 baud
10	1800 baud
11	2400 baud
12	4800 baud
13	9600 baud
14	External A
15	External B

In the current configuration, only 110, 150 and 300 baud are really supported on dial-up lines, in that the code conversion and line control required for IBM 2741's (134.5 baud) must be implemented by the user's program, and the half-duplex line discipline required for the 202 dataset (1200 baud) is not supplied.

The next two characters of the argument structure specify the erase and kill characters respectively. (Defaults are # and @.)

The *mode* contains several bits which determine the system's treatment of the typewriter:

100000	Select one of two algorithms for backspace delays
040000	Select one of two algorithms for form-feed and vertical-tab delays
030000	Select one of four algorithms for carriage-return delays
006000	Select one of four algorithms for tab delays
001400	Select one of four algorithms for new-line delays
000200	even parity allowed on input (e. g. for M37s)
000100	odd parity allowed on input
000040	raw mode: wake up on all characters
000020	map CR into LF; echo LF or CR as CR-LF
000010	echo (full duplex)

000004 map upper case to lower on input (e. g. M33)
000002 echo and print tabs as spaces
000001 hang up (remove 'data terminal ready,' lead CD) after last close

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 will 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. Other types are unimplemented and are 0.

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 character (EOT), the interrupt character (DEL) and the quit character (FS) are not treated specially.

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 (used for GE TermiNet 300's and other terminals without the newline function).

The hangup mode 01 causes the line to be disconnected when the last process with the line open closes it or terminates. It is useful when a port is to be used for some special purpose; for example, if it is associated with an ACU used to place outgoing calls.

This system call is also used with certain special files other than typewriters, but since none of them are part of the standard system the specifications will not be given.

SEE ALSO

stty (I), gtty (II)

DIAGNOSTICS

The error bit (c-bit) is set if the file descriptor does not refer to a typewriter. From C, a negative value indicates an error.

NAME

sync – update super-block

SYNOPSIS

(sync = 36.; not in assembler)

sys 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 (VIII), update (VIII)

DIAGNOSTICS

–

NAME

time – get date and time

SYNOPSIS

(time = 13.)
sys time

time(tvec)
int tvec[2];

DESCRIPTION

Time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds. From *as,* the high order word is in the r0 register and the low order is in r1. From C, the user-supplied vector is filled in.

SEE ALSO

date (I), stime (II), ctime (III)

DIAGNOSTICS

–

NAME

times – get process times

SYNOPSIS

(times = 43.; not in assembler)
sys times; buffer

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 1/60 seconds.

After the call, the buffer will appear as follows:

```
struct tbuffer {
        int     proc_user_time;
        int     proc_system_time;
        int     child_user_time[2];
        int     child_system_time[2];
};
```

The children times are the sum of the children's process times and their children's times.

SEE ALSO

time (I)

DIAGNOSTICS

–

BUGS

The process times should be 32 bits as well.

NAME

umount – dismount file system

SYNOPSIS

(umount = 22.)
sys umount; special

DESCRIPTION

Umount announces to the system that special file *special* is no longer to contain a removable file system. The file associated with the special file reverts to its ordinary interpretation; see *mount* (II).

SEE ALSO

umount (VIII), mount (II)

DIAGNOSTICS

Error bit (c-bit) set if no file system was mounted on the special file or if there are still active files on the mounted file system.

NAME

unlink – remove directory entry

SYNOPSIS

(unlink = 10.)
sys unlink; name

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 (I), rmdir (I), link (II)

DIAGNOSTICS

The error bit (c-bit) is set to indicate that the file does not exist or that its directory cannot be written. Write permission is not required on the file itself. It is also illegal to unlink a directory (except for the super-user). From C, a –1 return indicates an error.

NAME

wait − wait for process to terminate

SYNOPSIS

(wait = 7.)
sys wait
(process ID in r0)
(status in r1)

wait(status)
int *status;

DESCRIPTION

Wait causes its caller to delay until 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 r0). In the case of several children several *wait* calls are needed to learn of all the deaths.

If no error is indicated on return, the r1 high byte (resp. the high byte stored into *status*) contains the low byte of the child process r0 (resp. the argument of *exit*) when it terminated. The r1 (resp. *status*) low byte contains the termination status of the process. See signal (II) 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 (II). 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 (II), fork (II), signal (II)

DIAGNOSTICS

The error bit (c-bit) is set if there are no children not previously waited for. From C, a returned value of −1 indicates an error.

NAME

write − write on a file

SYNOPSIS

(write = 4.)
(file descriptor in r0)
sys write; buffer; nbytes

write(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 address of *nbytes* contiguous bytes which are written on the output file. The number of characters actually written is returned (in r0). 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 (II), open (II), pipe (II)

DIAGNOSTICS

The error bit (c-bit) is set on an error: bad descriptor, buffer address, or count; physical I/O errors. From C, a returned value of −1 indicates an error.

NAME

abort − generate an IOT fault

SYNOPSIS

abort()

DESCRIPTION

Abort executes the IOT instruction. This is usually considered a program fault by the system and results in termination with a core dump. It is used to generate a core image for debugging.

SEE ALSO

db (I), cdb (I), signal (II)

DIAGNOSTICS

usually ‘‘IOT trap -- core dumped’’ from the Shell.

BUGS

NAME

abs, fabs – absolute value

SYNOPSIS

abs(i)
int i;

double fabs(x)
double x;

DESCRIPTION

Abs returns the absolute value of its integer operand; *fabs* is the *double* version.

NAME

alloc, free – core allocator

SYNOPSIS

char *alloc(size)

free(ptr)
char *ptr;

DESCRIPTION

Alloc and *free* provide a simple general-purpose core management package. *Alloc* is given a size in bytes; it returns a pointer to an area at least that size which is even and hence can hold an object of any type. The argument to *free* is a pointer to an area previously allocated by *alloc;* this space is made available for further allocation.

Needless to say, grave disorder will result if the space assigned by *alloc* is overrun or if some random number is handed to *free.*

The routine uses a first-fit algorithm which coalesces blocks being freed with other blocks already free. It calls *sbrk* (see *break (II))* to get more core from the system when there is no suitable space already free.

DIAGNOSTICS

Returns −1 if there is no available core.

BUGS

Allocated memory contains garbage instead of being cleared.

NAME

atan, atan2 – arc tangent function

SYNOPSIS

jsr pc,atan[**2**]

double atan(x)
double x;

double atan2(x, y)
double x, y;

DESCRIPTION

The *atan* entry returns the arc tangent of fr0 in fr0; from C, the arc tangent of *x* is returned. The range is $-\pi/2$ to $\pi/2$. The *atan2* entry returns the arc tangent of fr0/fr1 in fr0; from C, the arc tangent of *x/y* is returned. The range is $-\pi$ to π.

DIAGNOSTIC

There is no error return.

BUGS

NAME

atof – convert ASCII to floating

SYNOPSIS

double atof(nptr)
char *nptr;

DESCRIPTION

Atof converts a string to a floating number. *Nptr* should point to a string containing the number; the first unrecognized character ends the number.

The only numbers recognized are: an optional minus sign followed by a string of digits optionally containing one decimal point, then followed optionally by the letter **e** followed by a signed integer.

DIAGNOSTICS

There are none; overflow results in a very large number and garbage characters terminate the scan.

BUGS

The routine should accept initial **+**, initial blanks, and **E** for **e**. Overflow should be signalled.

-

NAME

atoi – convert ASCII to integer

SYNOPSIS

atoi(nptr)
char *nptr;

DESCRIPTION

Atoi converts the string pointed to by *nptr* to an integer. The string can contain leading blanks or tabs, an optional '−', and then an unbroken string of digits. Conversion stops at the first non-digit.

SEE ALSO

atof (III)

BUGS

There is no provision for overflow.

NAME

crypt – password encoding

SYNOPSIS

```
mov     $key,r0
jsr     pc,crypt

char *crypt(key)
char *key;
```

DESCRIPTION

On entry, r0 points to a string of characters terminated by an ASCII NUL. The routine performs an operation on the key which is difficult to invert (i.e. encrypts it) and leaves the resulting eight bytes of ASCII alphanumerics in a global cell called ‘‘word’’.

From C, the *key* argument is a string and the value returned is a pointer to the eight-character result.

This routine is used to encrypt all passwords.

SEE ALSO

passwd(I), passwd(V), login(I)

BUGS

Short or otherwise simple passwords can be decrypted easily by exhaustive search. Six characters of gibberish is reasonably safe.

NAME

ctime, localtime, gmtime – convert date and time to ASCII

SYNOPSIS

char *ctime(tvec)
int tvec[2];

[from Fortran]
double precision ctime
... = ctime(dummy)

int *localtime(tvec)
int tvec[2];

int *gmtime(tvec)
int tvec[2];

DESCRIPTION

Ctime converts a time in the vector *tvec* such as returned by time (II) into ASCII and returns a pointer to a character string in the form

Sun Sep 16 01:03:52 1973\n\0

All the fields have constant width.

The *localtime* and *gmtime* entries return pointers to integer vectors 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. The value is a pointer to an array whose components are

0 seconds
1 minutes
2 hours
3 day of the month (1-31)
4 month (0-11)
5 year – 1900
6 day of the week (Sunday = 0)
7 day of the year (0-365)
8 Daylight Saving Time flag if non-zero

The external variable *timezone* contains the difference, in seconds, between GMT and local standard time (in EST, is 5*60*60); the external variable *daylight* is non-zero iff the standard U.S.A. Daylight Savings Time conversion should be applied. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

A routine named *ctime* is also available from Fortran. Actually it more resembles the *time* (II) system entry in that it returns the number of seconds since the epoch 0000 GMT Jan. 1, 1970 (as a floating-point number).

SEE ALSO

time(II)

BUGS

NAME

ecvt, fcvt – output conversion

SYNOPSIS

jsr pc,ecvt

jsr pc,fcvt

char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
...

DESCRIPTION

Ecvt is called with a floating point number in fr0.

On exit, the number has been converted into a string of ascii digits in a buffer pointed to by r0. The number of digits produced is controlled by a global variable *_ndigits*.

Moreover, the position of the decimal point is contained in r2: r2=0 means the d.p. is at the left hand end of the string of digits; r2>0 means the d.p. is within or to the right of the string.

The sign of the number is indicated by r1 (0 for +; 1 for –).

The low order digit has suffered decimal rounding (i. e. may have been carried into).

From C, the *value* is converted and a pointer to a null-terminated string of *ndigit* digits is returned. The position of the decimal point 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.

Fcvt is identical to *ecvt*, except that the correct digit been rounded for F-style output of the number of digits specified by *_ndigits*.

SEE ALSO

printf (III)

BUGS

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. Instead, their addresses coincide with the first address above the program text region *(etext),* above the initialized data region *(edata),* or uninitialized data region *(end).* The last is the same as the program break. Values are given to these symbols by the link editor *ld* (I) when, and only when, they are referred to but not defined in the set of programs loaded.

The usage of these symbols is rather specialized, but one plausible possibility is

```
extern end;
...
... = brk(&end+...);
```

(see *break* (II)). The problem with this is that it ignores any other subroutines which may want to extend core for their purposes; these include *sbrk* (see *break* (II)), *alloc* (III), and also secret subroutines invoked by the profile (−p) option of *cc.* Of course it was for the benefit of such systems that the symbols were invented, and user programs, unless they are in firm control of their environment, are wise not to refer to the absolute symbols directly.

One technique sometimes useful is to call *sbrk(0),* which returns the value of the current program break, instead of referring to *&end,* which yields the program break at the instant execution started.

These symbols are accessible from assembly language if it is remembered that they should be prefixed by ‘ _ ’

SEE ALSO

break (II), alloc (III)

BUGS

NAME

exp – exponential function

SYNOPSIS

jsr pc,exp

double exp(x)
double x;

DESCRIPTION

The exponential of fr0 is returned in fr0. From C, the exponential of *x* is returned.

DIAGNOSTICS

If the result is not representable, the c-bit is set and the largest positive number is returned. From C, no diagnostic is available.

Zero is returned if the result would underflow.

BUGS

NAME

floor, ceil – floor and ceiling functions

SYNOPSIS

double floor(x)
double x;

double ceil(x)
double x;

DESCRIPTION

The floor function returns the largest integer (as a double precision number) not greater than **x**.

The ceil function returns the smallest integer not less than **x**.

BUGS

-

NAME

fmod – floating modulo function

SYNOPSIS

double fmod(x, y)
double x, y;

DESCRIPTION

Fmod returns the number *f* such that $x = iy + f$, i is an integer, and $0 \leq f < y$.

BUGS

NAME

fptrap – floating point interpreter

SYNOPSIS

sys signal; 4; fptrap

DESCRIPTION

Fptrap is a simulator of the 11/45 FP11-B floating point unit. It works by intercepting illegal instruction traps and decoding and executing the floating point operation codes.

FILES

In systems with real floating point, there is a fake routine in /lib/liba.a with this name; when simulation is desired, the real version should be put in liba.a

DIAGNOSTICS

A break point trap is given when a real illegal instruction trap occurs.

SEE ALSO

signal (II), cc (I) ('−f' option)

BUGS

Rounding mode is not interpreted. It's slow.

NAME

gamma – log gamma function

SYNOPSIS

jsr pc,gamma

double gamma(x)
double x;

DESCRIPTION

If *x* is passed (in fr0) *gamma* returns $\ln|\Gamma(x)|$ (in fr0). The sign of $\Gamma(x)$ is returned in the external integer *signgam.* The following C program might be used to calculate Γ:

```
y = gamma(x);
if (y > 88.)
        error( );
y = exp(y);
if(signgam)
        y = –y;
```

DIAGNOSTICS

The c-bit is set on negative integral arguments and the maximum value is returned. There is no error return for C programs.

BUGS

No error return from C.

NAME

getarg, iargc – get command arguments from Fortran

SYNOPSIS

call getarg (i, iarray [, isize])

... = iargc(dummy)

DESCRIPTION

The *getarg* entry fills in *iarray* (which is considered to be *integer)* with the Hollerith string representing the *i* th argument to the command in which it it is called. If no *isize* argument is specified, at least one blank is placed after the argument, and the last word affected is blank padded. The user should make sure that the array is big enough.

If the *isize* argument is given, the argument will be followed by blanks to fill up *isize* words, but even if the argument is long no more than that many words will be filled in.

The blank-padded array is suitable for use as an argument to setfil (III).

The *iargc* entry returns the number of arguments to the command, counting the first (file-name) argument.

SEE ALSO

exec (II), setfil (III)

BUGS

NAME

getc, getw, fopen − buffered input

SYNOPSIS

```
mov     $filename,r0
jsr     r5,fopen; iobuf

fopen(filename, iobuf)
char *filename;
struct buf *iobuf;

jsr     r5,getc; iobuf
(character in r0)

getc(iobuf)
struct buf *iobuf;

jsr     r5,getw; iobuf
(word in r0)

getw(iobuf)
struct buf *iobuf;
```

DESCRIPTION

These routines provide a buffered input facility. *Iobuf* is the address of a 518(10) byte buffer area whose contents are maintained by these routines. Its structure is

```
struct buf {
        int fildes;         /* File descriptor */
        int nleft;          /* Chars left in buffer */
        char *nextp;        /* Ptr to next character */
        char buff[512];     /* The buffer */
};
```

Fopen may be called initially to open the file. On return, the error bit (c-bit) is set if the open failed. If *fopen* is never called, *get* will read from the standard input file. From C, the value is negative if the open failed.

Getc returns the next byte from the file in r0. The error bit is set on end of file or a read error. From C, the character is returned as an integer, without sign extension; it is −1 on end-of-file or error.

Getw returns the next word in r0. *Getc* and *getw* may be used alternately; there are no odd/even problems. *Getw* is may be called from C; −1 is returned on end-of-file or error, but of course is also a legitimate value.

Iobuf must be provided by the user; it must be on a word boundary.

To reuse the same buffer for another file, it is sufficient to close the original file and call *fopen* again.

SEE ALSO

open (II), read (II), getchar (III), putc (III)

DIAGNOSTICS

c-bit set on EOF or error; from C, negative return indicates error or EOF. Moreover, *errno* is set by this routine just as it is for a system call (see introduction (II)).

BUGS

NAME

getchar – read character

SYNOPSIS

getchar( )

DESCRIPTION

Getchar provides the simplest means of reading characters from the standard input for C programs. It returns successive characters until end-of-file, when it returns ‘‘\0’’.

Associated with this routine is an external variable called *fin*, which is a structure containing a buffer such as described under *getc* (III).

Generally speaking, *getchar* should be used only for the simplest applications; *getc* is better when there are multiple input files.

SEE ALSO

getc (III)

DIAGNOSTICS

Null character returned on EOF or error.

BUGS

−1 should be returned on EOF; null is a legitimate character.

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

passwd (V)

DIAGNOSTICS

non-zero return on error.

BUGS

NAME

hmul – high-order product

SYNOPSIS

hmul(x, y)

DESCRIPTION

Hmul returns the high-order 16 bits of the product of **x** and **y.** (The binary multiplication operator generates the low-order 16 bits of a product.)

BUGS

NAME

ierror – catch Fortran errors

SYNOPSIS

if (ierror (*errno* **) .ne. 0) goto** *label*

DESCRIPTION

Ierror provides a way of detecting errors during the running of a Fortran program. Its argument is a run-time error number such as enumerated in *fc* (I).

When *ierror* is called, it returns a 0 value; thus the **goto** statement in the synopsis is not executed. However, the routine stores inside itself the call point and invocation level. If and when the indicated error occurs, a **return** is simulated from *ierror* with a non-zero value; thus the **goto** (or other statement) is executed. It is a ghastly error to call *ierror* from a subroutine which has already returned when the error occurs.

This routine is essentially tailored to catching end-of-file situations. Typically it is called just before the start of the loop which reads the input file, and the **goto** jumps to a graceful termination of the program.

There is a limit of 5 on the number of different error numbers which can be caught.

SEE ALSO

fc (I)

BUGS

There is no way to ignore errors.

NAME

ldiv, lrem – long division

SYNOPSIS

ldiv(hidividend, lodividend, divisor)

lrem(hidividend, lodividend, divisor)

DESCRIPTION

The concatenation of the signed 16-bit *hidividend* and the unsigned 16-bit *lodividend* is divided by *divisor*. The 16-bit signed quotient is returned by *ldiv* and the 16-bit signed remainder is returned by *lrem.* Divide check and erroneous results will occur unless the magnitude of the divisor is greater than that of the high-order dividend.

An integer division of an unsigned dividend by a signed divisor may be accomplished by

```
quo = ldiv(0, dividend, divisor);
```

and similarly for the remainder operation.

Often both the quotient and the remainder are wanted. Therefore *ldiv* leaves a remainder in the external cell *ldivr.*

BUGS

No divide check check.

NAME

locv – long output conversion

SYNOPSIS

char *locv(hi, lo)
int hi, lo;

DESCRIPTION

Locv converts a signed double-precision integer, whose parts are passed as arguments, to the equivalent ASCII character string and returns a pointer to that string.

BUGS

Since *locv* returns a pointer to a static buffer containing the converted result, it cannot be used twice in the same expression; the second result overwrites the first.

NAME

log – natural logarithm

SYNOPSIS

jsr pc,log

double log(x)
double x;

DESCRIPTION

The natural logarithm of fr0 is returned in fr0. From C, the natural logarithm of **x** is returned.

DIAGNOSTICS

The error bit (c-bit) is set if the input argument is less than or equal to zero and the result is a negative number very large in magnitude. From C, there is no error indication.

BUGS

NAME

monitor – prepare execution profile

SYNOPSIS

monitor(lowpc, highpc, buffer, bufsize)
int lowpc( ), highpc( ), buffer[ ], bufsize;

DESCRIPTION

Monitor is an interface to the system's profile entry (II). *Lowpc* and *highpc* are the names of two functions; *buffer* is the address of a (user supplied) array of *bufsize* integers. *Monitor* arranges for the system to sample the user's program counter periodically and record the execution histogram in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc.* 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(2, &etext, buf, bufsize);
```

Etext is a loader-defined symbol which lies just above all the program text.

To stop execution monitoring and write the results on the file *mon.out,* use

```
monitor(0);
```

Then, when the program exits, prof (I) can be used to examine the results.

It is seldom necessary to call this routine directly; the **−p** option of *cc* is simpler if one is satisfied with its default profile range and resolution.

FILES

mon.out

SEE ALSO

prof (I), profil (II), cc (I)

NAME

nargs – argument count

SYNOPSIS

nargs( )

DESCRIPTION

Nargs returns the number of actual parameters supplied by the caller of the routine which calls *nargs*.

The argument count is accurate only when none of the actual parameters is *float* or *double*. Such parameters count as four arguments instead of one.

BUGS

As indicated. Also, this routine does not work (and cannot be made to work) in programs with separated I and D space. Altogether it is best to avoid using this routine and depend, for example, on passing an explicit argument count.

NAME

nlist – get entries from name list

SYNOPSIS

```
nlist(filename, nl)
char *filename;
struct {
        char    name[8];
        int     type;
        int     value;
} 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 a list of 8-character names (null padded) each followed by two words. 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 placed in the two words following the name. If the name is not found, the type entry is set to −1.

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 (V)

DIAGNOSTICS

All type entries are set to −1 if the file cannot be found or if it is not a valid namelist.

BUGS

NAME

perror, sys_errlist, sys_nerr, errno – system error messages

SYNOPSIS

perror(s)
char *s;

int sys_nerr;
char *sys_errlist[];

int errno;

DESCRIPTION

Perror produces a short error message 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,* 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 largest message number 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

Introduction to System Calls

BUGS

NAME

pow – floating exponentiation

SYNOPSIS

```
movf    x,fr0
movf    y,fr1
jsr     pc,pow

double pow(x,y)
double x, y;
```

DESCRIPTION

Pow returns the value of x^y (in fr0). *Pow(0.0, y)* is 0 for any *y*. *Pow(−x, y)* returns a result only if *y* is an integer.

SEE ALSO

exp (III), log (III)

DIAGNOSTICS

The carry bit is set on return in case of overflow, *pow(0.0, 0.0),* or *pow(−x, y)* for non-integral *y*. From C there is no diagnostic.

BUGS

NAME

printf – formatted print

SYNOPSIS

printf(format, arg$_1$, ...);
char *format;

DESCRIPTION

Printf converts, formats, and prints its arguments after the first under control of the first argument. The first argument is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to *printf.*

Each conversion specification is introduced by the character **%**. Following the **%**, there may be

- an optional minus sign "–" which specifies *left adjustment* of the converted argument in the indicated field;
- an optional digit string specifying a *field width;* if the converted argument has fewer characters than the field width it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width;
- an optional period "**.**" which serves to separate the field width from the next digit string;
- an optional digit string *(precision)* which specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string;
- a character which indicates the type of conversion to be applied.

The conversion characters and their meanings are

d
o
x The integer argument is converted to decimal, octal, or hexadecimal notation respectively.

f The argument is converted to decimal notation in the style "[–]ddd.ddd" where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed. The argument should be *float* or *double.*

e The argument is converted in the style "[–]d**.**ddd**e**±dd" where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced. The argument should be a *float* or *double* quantity.

c The argument character is printed.

s The argument is taken to be a string (character pointer) and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however if the precision is 0 or missing all characters up to a null are printed.

l The argument is taken to be an unsigned integer which is converted to decimal and printed (the result will be in the range 0 to 65535).

If no recognizable character appears after the **%**, that character is printed; thus **%** may be printed by use of the string **%%**. In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by *printf* are printed by calling *putchar.*

SEE ALSO

putchar (III)

BUGS

Very wide fields (>128 characters) fail.

NAME

putc, putw, fcreat, fflush – buffered output

SYNOPSIS

```
mov     $filename,r0
jsr     r5,fcreat; iobuf

fcreat(file, iobuf)
char *file;
struct buf *iobuf;

(get byte in r0)
jsr     r5,putc; iobuf

putc(c, iobuf)
int c;
struct buf *iobuf;

(get word in r0)
jsr     r5,putw; iobuf

putw(w, iobuf);
int w;
struct buf *iobuf;

jsr     r5,flush; iobuf

fflush(iobuf)
struct buf *iobuf;
```

DESCRIPTION

Fcreat creates the given file (mode 666) and sets up the buffer *iobuf* (size 518 bytes); *putc* and *putw* write a byte or word respectively onto the file; *flush* forces the contents of the buffer to be written, but does not close the file. The structure of the buffer is:

```
struct buf {
        int fildes;         /* File descriptor */
        int nunused;        /* Remaining slots */
        char *xfree;        /* Ptr to next free slot */
        char buff[512];     /* The buffer */
};
```

Before terminating, a program should call *flush* to force out the last of the output (*fflush* from C).

The user must supply *iobuf,* which should begin on a word boundary.

To write a new file using the same buffer, it suffices to call *[f]flush,* close the file, and call *fcreat* again.

SEE ALSO

creat (II), write (II), getc (III)

DIAGNOSTICS

Fcreat sets the error bit (c-bit) if the file creation failed (from C, returns −1). *Putc* and *putw* return their character (word) argument. In all calls *errno* is set appropriately to 0 or to a system error number. See introduction (II).

BUGS

NAME

putchar, flush – write character

SYNOPSIS

putchar(ch)

flush( )

DESCRIPTION

Putchar writes out its argument and returns it unchanged. Only the low-order byte is written, and only if it is non-null. Unless other arrangements have been made, *putchar* writes in unbuffered fashion on the standard output file.

Associated with this routine is an external variable *fout* which has the structure of a buffer discussed under putc (III). If the file descriptor part of this structure (first word) is greater than 2, output via *putchar* is buffered. To achieve buffered output one may say, for example,

```
fout = dup(1);          or
fout = creat(...);
```

In such a case *flush* must be called before the program terminates in order to flush out the buffered output. *Flush* may be called at any time.

SEE ALSO

putc (III)

BUGS

The *fout* notion is kludgy.

-

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. It is 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 (I)

BUGS

NAME

rand, srand – random number generator

SYNOPSIS

(seed in r0)

```
jsr     pc,srand        /to initialize
jsr     pc,rand /to get a random number
srand(seed)
int seed;
rand( )
```

DESCRIPTION

Rand uses a multiplicative congruential random number generator to return successive pseudo-random numbers (in r0) in the range from 0 to $2^{15}-1$.

The generator is reinitialized by calling *srand* with 1 as argument (in r0). It can be set to a random starting point by calling *srand* with whatever you like as argument, for example the low-order word of the time.

BUGS

The low-order bits are not very random.

NAME

reset, setexit – execute non-local goto

SYNOPSIS

setexit( )

reset( )

DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setexit saves its stack environment in a static place for later use by *reset.*

Reset restores the environment saved by the last call of *setexit.* It then returns in such a way that execution continues as if the call of *setexit* had just returned. All accessible data have values as of the time *reset* was called.

The routine that called *setexit* must still be active when *reset* is called.

SEE ALSO

signal (II)

BUGS

NAME

setfil – specify Fortran file name

SYNOPSIS

call setfil (unit**,** hollerith-string **)**

DESCRIPTION

Setfil provides a primitive way to associate an integer I/O *unit* number with a file named by the *hollerith-string*. The end of the file name is indicated by a blank. Subsequent I/O on this unit number will refer to the file whose name is specified by the string.

Setfil should be called only before any I/O has been done on the *unit,* or just after doing a **rewind** or **end-file.** It is ineffective for unit numbers 5 and 6.

SEE ALSO

fc (I)

BUGS

The exclusion of units 5 and 6 is unwarranted.

-

NAME

sin, cos – trigonometric functions

SYNOPSIS

jsr pc,sin (cos)

double sin(x)
double x;

double cos(x)
double x;

DESCRIPTION

The sine (cosine) of fr0 (resp. **x**), measured in radians, is returned (in fr0).

The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

BUGS

NAME

sqrt – square root function

SYNOPSIS

jsr pc,sqrt

double sqrt(x)
double x;

DESCRIPTION

The square root of fr0 (resp. **x**) is returned (in fr0).

DIAGNOSTICS

The c-bit is set on negative arguments and 0 is returned. There is no error return for C programs.

BUGS

No error return from C.

NAME

ttyn – return name of current typewriter

SYNOPSIS

jsr pc,ttyn

ttyn(file)

DESCRIPTION

Ttyn hunts up the last character of the name of the typewriter which is the standard input (from *as*) or is specified by the argument *file* descriptor (from C). If *n* is returned, the typewriter name is then ''/dev/tty*n*''.

x is returned if the indicated file does not correspond to a typewriter.

BUGS

NAME

cat – phototypesetter interface

DESCRIPTION

Cat provides the interface to a Graphic Systems C/A/T phototypesetter. Bytes written on the file specify font, size, and other control information as well as the characters to be flashed. The coding will not be described here.

Only one process may have this file open at a time. It is write-only.

FILES

/dev/cat

SEE ALSO

troff (I), Graphic Systems specification (available on request)

BUGS

NAME

dc – DC-11 communications interface

DESCRIPTION

The discussion of typewriter I/O given in tty (IV) applies to these devices.

The DC-11 typewriter interface operates at any of four speeds, independently settable for input and output. The speed is selected by the same encoding used by the DH (IV) device (enumerated in stty (II)); impossible speed changes are ignored.

FILES

/dev/tty[01234567abcd] 113B Dataphones (not currently connected– see dh (IV))

SEE ALSO

tty (IV), stty (II), dh (IV)

BUGS

NAME

dh – DH-11 communications multiplexer

DESCRIPTION

Each line attached to the DH-11 communications multiplexer behaves as described in tty (IV). Input and output for each line may independently be set to run at any of 16 speeds; see stty (II) for the encoding.

FILES

/dev/tty[f-u]

SEE ALSO

tty (IV), stty (II)

BUGS

NAME

dn – DN-11 ACU interface

DESCRIPTION

The *dn?* files are write-only. The permissible codes are:

0-9	dial 0-9
:	dial *
;	dial #
–	4 second delay for second dial tone
=	end-of-number

The entire telephone number must be presented in a single *write* system call.

It is recommended that an end-of-number code be given even though not all ACU's actually require it.

FILES

/dev/dn0	connected to 801 with dp0
/dev/dn1	not currently connected
/dev/dn2	not currently connected

SEE ALSO

dp (IV)

BUGS

NAME

dp – DP-11 201 data-phone interface

DESCRIPTION

The *dp0* file is a 201 data-phone interface. *Read* and *write* calls to dp0 are limited to a maximum of 512 bytes. Each write call is sent as a single record. Seven bits from each byte are written along with an eighth odd parity bit. The sync must be user supplied. Each read call returns characters received from a single record. Seven bits are returned unaltered; the eighth bit is set if the byte was not received in odd parity. A 10 second time out is set and a zero-byte record is returned if nothing is received in that time.

FILES

/dev/dp0

SEE ALSO

dn (IV), gerts (III)

BUGS

NAME

hp – RH-11/RP04 moving-head disk

DESCRIPTION

The files *hp0 ... hp7* refer to sections of RP disk drive 0. The files *hp8 ... hp15* refer to drive 1 etc. This is done since the size of a full RP drive is 170,544 blocks and internally the system is only capable of addressing 65536 blocks. Also since the disk is so large, this allows it to be broken up into more manageable pieces.

The origin and size of the pseudo-disks on each drive are as follows:

disk	start	length
0	0	9614
1	18392	65535
2	48018	65535
3	149644	20900
4	0	40600
5	41800	40600
6	83600	40600
7	125400	40600

It is unwise for all of these files to be present in one installation, since there is overlap in addresses and protection becomes a sticky matter.

The *hp* files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RP files begin with *rhp* and end with a number which selects the same disk section as the corresponding *hp* file.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/hp?, /dev/rhp?

BUGS

NAME

hs – RH11/RS03-RS04 fixed-head disk file

DESCRIPTION

The files *hs0 ... hs7* refer to RJS03 disk drives 0 through 7. The files *hs8 ... hs15* refer to RJS04 disk drives 0 through 7. The RJS03 drives are each 1024 blocks long and the RJS04 drives are 2048 blocks long.

The *hs* files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a ''raw'' inteface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw HS files begin with *rhs*. The same minor device considerations hold for the raw interface as for the normal interface.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/hs?, /dev/rhs?

BUGS

NAME

ht – RH-11/TU-16 magtape interface

DESCRIPTION

The files *mt0, ..., mt7* refer to the DEC RH/TM/TU16 magtape. When opened for reading or writing, the tape is rewound. When closed, it is rewound; if it was open for writing, a double end-of-file is written first.

A standard tape consists of a series of 512 byte records terminated by a double end-of-file. To the extent possible, the system makes it possible, if inefficient, to treat the tape like any other file. Seeks have their usual meaning and it is possible to read or write a byte at a time. Writing in very small units is inadvisable, however, because it tends to create monstrous record gaps.

The *mt* files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the ''raw'' interface is appropriate. The associated files are named *rmt0, ..., rmt7.* Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, provided it is no greater than the buffer size; if the record is long, an error is indicated. In raw tape I/O, the buffer must begin on a word boundary and the count must be even. Seeks are ignored. An error is returned when a tape mark is read, but another read will fetch the first record of the new tape file.

FILES

/dev/mt?, /dev/rmt?

SEE ALSO

tp (I)

BUGS

Raw I/O doesnt work yet. The magtape system is supposed to be able to take 64 drives. Such addressing has never been tried. These bugs will be fixed when we get more experience with this device.

If any non-data error is encountered, it refuses to do anything more until closed. In raw I/O, there should be a way to perform forward and backward record and file spacing and to write an EOF mark.

NAME

kl – KL-11 or DL-11 asynchronous interface

DESCRIPTION

The discussion of typewriter I/O given in tty (IV) applies to these devices.

Since they run at a constant speed, attempts to change the speed via stty (II) are ignored.

The on-line console typewriter is interfaced using a KL-11 or DL-11. By appropriate switch settings during a reboot, UNIX will come up as a single-user system with I/O on the console typewriter.

FILES

/dev/tty8console

SEE ALSO

tty (IV), init (VIII)

BUGS

Modem control for the DL-11E is not implemented.

NAME

lp – line printer

DESCRIPTION

Lp provides the interface to any of the standard DEC line printers. When it is opened or closed, a suitable number of page ejects is generated. Bytes written are printed.

An internal parameter within the driver determines whether or not the device is treated as having a 96- or 64-character set. In half-ASCII mode, lower case letters are turned into upper case and certain characters are escaped according to the following table:

{	(
}	)
`	'
|	!
~	^

The driver correctly interprets carriage returns, backspaces, tabs, and form feeds. A sequence of newlines which extends over the end of a page is turned into a form feed. All lines are indented 8 characters. Lines longer than 80 characters are truncated. These numbers are parameters in the driver; another parameter allows indenting all printout if it is unpleasantly near the left margin.

FILES

/dev/lp

SEE ALSO

lpr (I)

BUGS

Half-ASCII mode, the indent and the maximum line length should be settable by a call analogous to stty (II).

NAME

mem, kmem, null − 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 using the debugger.

A memory address is an 18-bit quantity which is used directly as a UNIBUS address. References to non-existent locations cause errors to be returned.

Examining and patching device registers is likely to lead to unexpected results when read-only or write-only bits are present.

The file *kmem* is the same as *mem* except that kernel virtual memory rather than physical memory is accessed. In particular, the I/O area of *kmem* is located beginning at 160000 (octal) rather than at 760000. The 1K region beginning at 140000 (octal) is the system's data for the current process.

The file *null* returns end-of-file on *read* and ignores *write.*

FILES

/dev/mem, /dev/kmem, /dev/null

NAME

pc − PC-11 paper tape reader/punch

DESCRIPTION

Ppt refers to the PC-11 paper tape reader or punch, depending on whether it is read or written.

When *ppt* is opened for writing, a 100-character leader is punched. Thereafter each byte written is punched on the tape. No editing of the characters is performed. When the file is closed, a 100-character trailer is punched.

When *ppt* is opened for reading, the process waits until tape is placed in the reader and the reader is on-line. Then requests to read cause the characters read to be passed back to the program, again without any editing. This means that several null leader characters will usually appear at the beginning of the file. Likewise several nulls are likely to appear at the end. End-of-file is generated when the tape runs out.

Seek calls for this file are meaningless.

FILES

/dev/ppt

BUGS

If both the reader and the punch are open simultaneously, the trailer is sometimes not punched. Sometimes the reader goes into a dead state in which it cannot be opened.

NAME

rf – RF11/RS11 fixed-head disk file

DESCRIPTION

This file refers to the concatenation of all RS-11 disks.

Each disk contains 1024 256-word blocks. The length of the combined RF file is 1024×(minor+1) blocks. That is minor device zero is taken to be 1024 blocks long; minor device one is 2048, etc.

The *rf0* file accesses the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a "raw" interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The name of the raw RF file is *rrf0*. The same minor device considerations hold for the raw interface as for the normal interface.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/rf0, /dev/rrf0

BUGS

The 512-byte restrictions on the raw device are not physically necessary, but are still imposed.

NAME

rk − RK-11/RK03 (or RK05) disk

DESCRIPTION

Rk? refers to an entire RK03 disk as a single sequentially-addressed file. Its 256-word blocks are numbered 0 to 4871.

Drive numbers (minor devices) of eight and larger are treated specially. Drive 8+*x* is the *x*+1 way interleaving of devices rk0 to rk*x*. Thus blocks on rk10 are distributed alternately among rk0, rk1, and rk2.

The *rk* files discussed above access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a ''raw'' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RK files begin with *rrk* and end with a number which selects the same disk as the corresponding *rk* file.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/rk?, /dev/rrk?

BUGS

Care should be taken in using the interleaved files. First, the same drive should not be accessed simultaneously using the ordinary name and as part of an interleaved file, because the same physical blocks have in effect two different names; this fools the system's buffering strategy. Second, the combined files cannot be used for swapping or raw I/O.

NAME

rp – RP-11/RP03 moving-head disk

DESCRIPTION

The files *rp0 ... rp7* refer to sections of RP disk drive 0. The files *rp8 ... rp15* refer to drive 1 etc. This is done since the size of a full RP drive is 81200 blocks and internally the system is only capable of addressing 65536 blocks. Also since the disk is so large, this allows it to be broken up into more manageable pieces.

The origin and size of the pseudo-disks on each drive are as follows:

disk	start	length
0	0	40600
1	40600	40600
2	0	9200
3	72000	9200
4	0	65535
5	15600	65535
6-7	unassigned	

It is unwise for all of these files to be present in one installation, since there is overlap in addresses and protection becomes a sticky matter. Here is a suggestion for two useful configurations: If the root of the file system is on some other device and the RP used as a mounted device, then *rp0* and *rp1,* which divide the disk into two equal size portions, is a good idea. Other things being equal, it is advantageous to have two equal-sized portions since one can always be copied onto the other, which is occasionally useful.

If the RP is the only disk and has to contain the root and the swap area, the root can be put on *rp2* and a mountable file system on *rp5.* Then the swap space can be put in the unused blocks 9200 to 15600 of *rp0* (or, equivalently, *rp4).* This arrangement puts the root file system, the swap area, and the i-list of the mounted file system relatively near each other and thus tends to minimize head movement.

The *rp* files access the disk via the system's normal buffering mechanism and may be read and written without regard to physical disk records. There is also a ''raw'' interface which provides for direct transmission between the disk and the user's read or write buffer. A single read or write call results in exactly one I/O operation and therefore raw I/O is considerably more efficient when many words are transmitted. The names of the raw RP files begin with *rrp* and end with a number which selects the same disk section as the corresponding *rp* file.

In raw I/O the buffer must begin on a word boundary, and counts should be a multiple of 512 bytes (a disk block). Likewise *seek* calls should specify a multiple of 512 bytes.

FILES

/dev/rp?, /dev/rrp?

BUGS

NAME

tc – TC-11/TU56 DECtape

DESCRIPTION

The files *tap0 ... tap7* refer to the TC-11/TU56 DECtape drives 0 to 7.

The 256-word blocks on a standard DECtape are numbered 0 to 577.

FILES

/dev/tap?

SEE ALSO

tp (I)

BUGS

NAME

tm – TM-11/TU-10 magtape interface

DESCRIPTION

The files *mt0, ..., mt7* refer to the DEC TU10/TM11 magtape. When opened for reading or writing, the tape is rewound. When closed, it is rewound; if it was open for writing, an end-of-file is written first.

A standard tape consists of a series of 512 byte records terminated by an end-of-file. To the extent possible, the system makes it possible, if inefficient, to treat the tape like any other file. Seeks have their usual meaning and it is possible to read or write a byte at a time. Writing in very small units is inadvisable, however, because it tends to create monstrous record gaps.

The *mt* files discussed above are useful when it is desired to access the tape in a way compatible with ordinary files. When foreign tapes are to be dealt with, and especially when long records are to be read or written, the ''raw'' interface is appropriate. The associated files are named *rmt0, ..., rmt7.* Each *read* or *write* call reads or writes the next record on the tape. In the write case the record has the same length as the buffer given. During a read, the record size is passed back as the number of bytes read, provided it is no greater than the buffer size; if the record is long, an error is indicated. In raw tape I/O, the buffer must begin on a word boundary and the count must be even. Seeks are ignored. An error is returned when a tape mark is read, but another read will fetch the first record of the new tape file.

FILES

/dev/mt?, /dev/rmt?

SEE ALSO

tp (I)

BUGS

If any non-data error is encountered, it refuses to do anything more until closed. In raw I/O, there should be a way to perform forward and backward record and file spacing and to write an EOF mark.

NAME

tty – general typewriter interface

DESCRIPTION

This section describes both a particular special file, and the general nature of the typewriter interface.

The file */dev/tty* is, in each process, a synonym for the control typewriter associated with that process. It is useful for programs or Shell sequences which wish to be sure of writing messages on the typewriter no matter how output has been redirected. It can also be used for programs which demand a file name for output, when typed output is desired and it is tiresome to find out which typewriter is currently in use.

As for typewriters 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; the KL, DC, and DH writeups (IV) describe peculiarities of the individual devices.

When a typewriter 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 typewriter file open in a process becomes the *control typewriter* for that process. The control typewriter plays a special role in handling quit or interrupt signals, as discussed below. The control typewriter is inherited by a child process during a *fork*.

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 which 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.

These special files have a number of modes which can be changed by use of the *stty* system call (II). When first opened, the interface mode is 300 baud; either parity accepted; 10 bits/character (one stop bit); and newline action character. Modes that can be changed by *stty* include the interface speed (if the hardware permits); acceptance of even parity, odd parity, or both; a raw mode in which all characters may be read one at a time; a carriage return (CR) mode in which CR is mapped into newline on input and either CR or line feed (LF) cause echoing of the sequence LF-CR; mapping of upper case letters into lower case; suppression of echoing; a variety of delays after function characters; and the printing of tabs as spaces. See *getty* (VIII) for the way that terminal speed and type are detected.

Normally, typewriter 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.

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 are generated on output and accepted on input:

for	use
`	\´
|	\!
~	\^
{	\(
}	\)

In raw mode, the program reading is awakened on each character. No erase or kill processing is done; and the EOT, quit and interrupt characters are not treated specially. The input parity bit is passed back to the reader, but parity is still generated for output characters.

The ASCII EOT (control-D) character may be used to generate an end of file from a typewriter. 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. The EOT is passed back unchanged in raw mode.

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 typewriter as control typewriter. 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 which read a typewriter and test for end-of-file on their input can terminate appropriately when hung up on.

Two characters have a special meaning when typed. The ASCII DEL character (sometimes called 'rubout') is not passed to a program but generates an *interrupt* signal which is sent to all processes with the associated control typewriter. 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* (II).

The ASCII character FS 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. If you find it hard to type this character, try control-\ or control-shift-L.

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 which respond to it from hanging up.

FILES

/dev/tty

SEE ALSO

dc (IV), kl (IV), dh (IV), getty (VIII), stty (I, II), gtty (I, II), signal (II)

BUGS

Half-duplex terminals are not supported. On raw-mode output, parity should be transmitted as specified in the characters written.

NAME

a.out – assembler and link editor output

DESCRIPTION

A.out is the output file of the assembler *as* and the link editor *ld.* Both programs make *a.out* executable if there were no errors and no unresolved external references.

This file has four sections: a header, the program and data text, a symbol table, and relocation bits (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.*

The header always contains 8 words:

1 A magic number (407, 410, or 411(8))
2 The size of the program text segment
3 The size of the initialized portion of the data segment
4 The size of the uninitialized (bss) portion of the data segment
5 The size of the symbol table
6 The entry location (always 0 at present)
7 Unused
8 A flag indicating relocation bits have been suppressed

The sizes of each segment are in bytes but are even. The size of the header is not included in any of the other sizes.

When a file produced by the assembler or loader 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 (word 0) is 407, 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 410, 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 again pure, write-protected, and shared, and moreover instruction and data space are separated; the text and data segment both begin at location 0. See the 11/45 handbook for restrictions which apply to this situation.

The stack will occupy the highest possible locations in the core image: from 177776(8) and growing downwards. The stack is automatically extended as required. The data segment is only extended as requested by the *break* system call.

The start of the text segment in the file is 20(8); the start of the data segment is $20+S_t$ (the size of the text) the start of the relocation information is $20+S_t+S_d$; the start of the symbol table is $20+2(S_t+S_d)$ if the relocation information is present, $20+S_t+S_d$ if not.

The symbol table consists of 6-word entries. The first four words contain the ASCII name of the symbol, null-padded. The next word is a flag indicating the type of symbol. The following values are possible:

00 undefined symbol
01 absolute symbol
02 text segment symbol
03 data segment symbol
37 file name symbol (produced by ld)
04 bss segment symbol
40 undefined external (.globl) symbol
41 absolute external symbol
42 text segment external symbol
43 data segment external symbol
44 bss segment external symbol

Values other than those given above may occur if the user has defined some of his own instructions.

The last word of a symbol table entry contains the value of the symbol.

If the 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 bits 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 ''suppress relocation'' 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:

00 indicates the reference is absolute
02 indicates the reference is to the text segment
04 indicates the reference is to initialized data
06 indicates the reference is to bss (uninitialized data)
10 indicates the reference is to an undefined external symbol.

Bit 0 of the relocation word indicates if *on* that the reference is relative to the pc (e.g. ''clr x''); if *off,* 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 (I), ld (I), strip (I), nm (I)

NAME

ar – archive (library) file format

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 is 177555(8) (it was chosen to be unlikely to occur anywhere else). The header of each file is 16 bytes long:

0-7	file name, null padded on the right
8-11	modification time of the file
12	user ID of file owner
13	file mode
14-15	file size

Each file begins on a word boundary; a null byte is inserted between files if necessary. Nevertheless the size give 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 (I), ld (I)

BUGS

Names are only 8 characters, not 14. More important, there isn't enough room to store the proper mode, so *ar* always extracts in mode 666.

NAME

ascii – map of ASCII character set

SYNOPSIS

cat /usr/pub/ascii

DESCRIPTION

Ascii is a map of the ASCII character set, to be printed as needed. It contains:

```
|000 nul|001 soh|002 stx|003 etx|004 eot|005 enq|006 ack|007 bel|
|010 bs |011 ht |012 nl |013 vt |014 np |015 cr |016 so |017 si |
|020 dle|021 dc1|022 dc2|023 dc3|024 dc4|025 nak|026 syn|027 etb|
|030 can|031 em |032 sub|033 esc|034 fs |035 gs |036 rs |037 us |
|040 sp |041  ! |042  " |043  # |044  $ |045  % |046  & |047  ´ |
|050  ( |051  ) |052  * |053  + |054  , |055  – |056  . |057  / |
|060  0 |061  1 |062  2 |063  3 |064  4 |065  5 |066  6 |067  7 |
|070  8 |071  9 |072  : |073  ; |074  < |075  = |076  > |077  ? |
|100  @ |101  A |102  B |103  C |104  D |105  E |106  F |107  G |
|110  H |111  I |112  J |113  K |114  L |115  M |116  N |117  O |
|120  P |121  Q |122  R |123  S |124  T |125  U |126  V |127  W |
|130  X |131  Y |132  Z |133  [ |134  \ |135  ] |136  ^ |137  _ |
|140  ` |141  a |142  b |143  c |144  d |145  e |146  f |147  g |
|150  h |151  i |152  j |153  k |154  l |155  m |156  n |157  o |
|160  p |161  q |162  r |163  s |164  t |165  u |166  v |167  w |
|170  x |171  y |172  z |173  { |174  | |175  } |176  ~ |177 del|
```

FILES

found in /usr/pub

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 (II)* 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. 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.

The format of the information in the first 1024 bytes is described by the *user* structure of the system. The important stuff not detailed therein is the locations of the registers. Here are their offsets. The parenthesized numbers for the floating registers are used if the floating-point hardware is in single precision mode, as indicated in the status register.

fpsr	0004	
fr0	0006	(0006)
fr1	0036	(0022)
fr2	0046	(0026)
fr3	0056	(0032)
fr4	0016	(0012)
fr5	0026	(0016)
r0	1772	
r1	1766	
r2	1750	
r3	1752	
r4	1754	
r5	1756	
sp	1764	
pc	1774	
ps	1776	

In general the debuggers *db (I)* and *cdb (I)* are sufficient to deal with core images.

SEE ALSO

cdb (I), db (I), signal (II)

NAME

dir – format of directories

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. Directory entries are 16 bytes long. The first word is the i-number of the file represented by the entry, if non-zero; if zero, the entry is empty.

Bytes 2-15 represent the (14-character) file name, null padded on the right. These bytes are not cleared for empty slots.

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

file system (V)

NAME

dump – incremental dump tape format

DESCRIPTION

The *dump* and *restor* commands are used to write and read incremental dump magnetic tapes.

The dump tape consists of blocks of 512-bytes each. The first block has the following structure.

```
struct {
        int     isize;
        int     fsize;
        int     date[2];
        int     ddate[2];
        int     tsize;
};
```

Isize, and *fsize* are the corresponding values from the super block of the dumped file system. (See file system (V).) *Date* is the date of the dump. *Ddate* is the incremental dump date. The incremental dump contains all files modified between *ddate* and *date. Tsize* is the number of blocks per reel. This block checksums to the octal value 031415.

Next there are enough whole tape blocks to contain one word per file of the dumped file system. This is *isize* divided by 16 rounded to the next higher integer. The first word corresponds to i-node 1, the second to i-node 2, and so forth. If a word is zero, then the corresponding file exists, but was not dumped. (Was not modified after *ddate)* If the word is −1, the file does not exist. Other values for the word indicate that the file was dumped and the value is one more than the number of blocks it contains.

The rest of the tape contains for each dumped file a header block and the data blocks from the file. The header contains an exact copy of the i-node (see file system (V)) and also checksums to 031415. The next-to-last word of the block contains the tape block number, to aid in (unimplemented) recovery after tape errors. The number of data blocks per file is directly specified by the control word for the file and indirectly specified by the size in the i-node. If these numbers differ, the file was dumped with a ‘phase error’.

SEE ALSO

dump (VIII), restor (VIII), file system(V)

NAME

fs – format of file system volume

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 256 word (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.* Starting from its first word, the format of a super-block is

```
struct {
        int     isize;
        int     fsize;
        int     nfree;
        int     free[100];
        int     ninode;
        int     inode[100];
        char    flock;
        char    ilock;
        char    fmod;
        int     time[2];
};
```

Isize is the number of blocks devoted to the i-list, which starts just after the super-block, in block 2. *Fsize* is the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block numbers; if an ''impossible'' block number 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 *free* array contains, in *free[1], ... , free[nfree−1],* up to 99 numbers of free blocks. *Free[0]* is the block number of the head of a chain of blocks constituting the free list. The first word in each free-chain block is the number (up to 100) of free-block numbers listed in the next 100 words of this chain member. The first of these 100 blocks is the link to the next member of the chain. To allocate a block: decrement *nfree,* and the new block is *free[nfree].* If the new block number is 0, there are no blocks left, so give an error. If *nfree* became 0, read in the block named by the new block number, replace *nfree* by its first word, and copy the block numbers in the next 100 words into the *free* array. To free a block, check if *nfree* is 100; if so, copy *nfree* and the *free* array into it, write it out, and set *nfree* to 0. In any event set *free[nfree]* to the freed block's number and increment *nfree.*

Ninode is the number of free i-numbers in the *inode* array. To allocate an i-node: if *ninode* is greater than 0, decrement it and return *inode[ninode].* If it was 0, read the i-list and place the numbers of all free inodes (up to 100) into the *inode* array, then try again. To free an i-node, provided *ninode* is less than 100, place its number into *inode[ninode]* and increment *ninode.* If *ninode* is already 100, 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.

Flock and *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 *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.

Time is the last time the super-block of the file system was changed, and is a double-precision representation of the number of seconds that have elapsed since 0000 Jan. 1 1970 (GMT). During a reboot, the *time* of the super-block for the root file system is used to set the system's idea of the time.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. Also, i-nodes are 32 bytes long, so 16 of them fit into a block. Therefore, i-node i is located in block $(i + 31) / 16$, and begins $32 \cdot ((i + 31) \pmod{16})$ bytes from its start. I-node 1 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 is as follows.

```
struct {
        int     flags;          /* +0: see below */
        char    nlinks;         /* +2: number of links to file */
        char    uid;            /* +3: user ID of owner */
        char    gid;            /* +4: group ID of owner */
        char    size0;          /* +5: high byte of 24-bit size */
        int     size1;          /* +6: low word of 24-bit size */
        int     addr[8];        /* +8: block numbers or device number */
        int     actime[2];      /* +24: time of last access */
        int     modtime[2];     /* +28: time of last modification */
};
```

The flags are as follows:

```
100000  i-node is allocated
060000  2-bit file type:
        000000  plain file
        040000  directory
        020000  character-type special file
        060000  block-type special file.
010000  large file
004000  set user-ID on execution
002000  set group-ID on execution
000400  read (owner)
000200  write (owner)
000100  execute (owner)
000070  read, write, execute (group)
000007  read, write, execute (others)
```

Special files are recognized by their flags and not by i-number. A block-type special file is basically 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 high byte of the first address word specifies the type of device; the low byte specifies one of several devices of that type. The device type numbers of block and character special files overlap.

The address words of ordinary files and directories contain the numbers of the blocks in the file (if it is small) or the numbers of indirect blocks (if the file is large). Byte number *n* of a file is accessed as follows. *N* is divided by 512 to find its logical block number (say *b*) in the file. If the file is small (flag 010000 is 0), then *b* must be less than 8, and the physical block number is *addr[b]*.

If the file is large, *b* is divided by 256 to yield *i*. If *i* is less than 7, then *addr[i]* is the physical block number of the indirect block. The remainder from the division yields the word in the indirect block which contains the number of the block for the sought-for byte.

If *i* is equal to 7, then the file has become extra-large (huge), and *addr[7]* is the address of a first indirect block. Each word in this block is the number of a second-level indirect block; each word in the second-level indirect blocks points to a data block. Notice that extra-large files are not marked by any mode bit, but only by having *addr[7]* non-zero; and that although this scheme allows for more than 256×256×512 = 33,554,432 bytes per file, the length of files is stored in 24 bits so in practice a file can be at most 16,777,216 bytes long.

For block *b* in 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, dcheck (VIII)

NAME

greek – graphics for extended TTY-37 type-box

SYNOPSIS

cat /usr/pub/greek

DESCRIPTION

Greek gives the mapping from ascii to the "shift out" graphics in effect between SO and SI on model 37 Teletypes with a 128-character type-box. It contains:

alpha	α	A	beta	β	B	gamma	γ	\
GAMMA	Γ	G	delta	δ	D	DELTA	Δ	W
epsilon	ε	S	zeta	ζ	Q	eta	η	N
THETA	Θ	T	theta	θ	O	lambda	λ	L
LAMBDA	Λ	E	mu	μ	M	nu	ν	@
xi	ξ	X	pi	π	J	PI	Π	P
rho	ρ	K	sigma	σ	Y	SIGMA	Σ	R
tau	τ	I	phi	ϕ	U	PHI	Φ	F
psi	ψ	V	PSI	Ψ	H	omega	ω	C
OMEGA	Ω	Z	nabla	∇	[	not	$\neg$	_
partial	∂	]	integral	$\int$	^			

SEE ALSO

ascii (VII)

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 newline. 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 (I), login (I), crypt (III), passwd (I)

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 (VIII), umount (VIII)

BUGS

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 (for now, always 1)
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.

FILES

/etc/passwd

SEE ALSO

login (I), crypt (III), passwd (I), group (V)

NAME

tabs – set tab stops

SYNOPSIS

cat /usr/pub/tabs

DESCRIPTION

Printing this file on a suitable terminal sets tab stops every 8 columns. Suitable terminals include the Teletype model 37 and the GE TermiNet 300.

These tab stop settings are desirable because UNIX assumes them in calculating delays.

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 boot procedures (VIII).

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:

path name	32 bytes
mode	2 bytes
uid	1 byte
gid	1 byte
unused	1 byte
size	3 bytes
time modified	4 bytes
tape address	2 bytes
unused	16 bytes
check sum	2 bytes

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 (file system (V)). 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 25 (resp. 63) on are available for file storage.

A fake entry (see tp (I)) has a size of zero.

SEE ALSO

file system (V), tp (I)

NAME

ttys – typewriter initialization data

DESCRIPTION

The *ttys* file is read by the *init* program and specifies which typewriter special files are to have a process created for them which will allow people to log in. It consists of lines of 3 characters each.

The first character is either '0' or '1'; the former causes the line to be ignored, the latter causes it to be effective. The second character is the last character in the name of a typewriter; e.g. *x* refers to the file '/dev/tty*x*'. The third character is used as an argument to the *getty* program, 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.)

FILES

/etc/ttys

SEE ALSO

init (VIII), getty (VIII), login (I)

NAME

utmp – user information

DESCRIPTION

This file allows one to discover information about who is currently using UNIX. The file is binary; each entry is 16(10) bytes long. The first eight bytes contain a user's login name or are null if the table slot is unused. The low order byte of the next word contains the last character of a typewriter name. The next two words contain the user's login time. The last word is unused.

FILES

/etc/utmp

SEE ALSO

init (VIII) and login (I), which maintain the file; who (I), which interprets it.

NAME

wtmp – user login history

DESCRIPTION

This file records all logins and logouts. Its format is exactly like utmp (V) except that a null user name indicates a logout on the associated typewriter. Furthermore, the typewriter name '~' indicates that the system was rebooted at the indicated time; the adjacent pair of entries with typewriter 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 (I) and init (VIII). Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by ac (VIII).

FILES

/usr/adm/wtmp

SEE ALSO

utmp (V), login (I), init (VIII), ac (VIII), who (I)

NAME

azel – satellite predictions

SYNOPSIS

azel [**−d**] [**−l**] satellite1 [**−d**] [**−l**] satellite2 ...

DESCRIPTION

Azel predicts, in convenient form, the apparent trajectories of Earth satellites whose orbital elements are given in the argument files. If a given satellite name cannot be read, an attempt is made to find it in a directory of satellites maintained by the programs's author. The **−d** option causes *azel* to ask for a date and read line 1 data (see below) from the standard input. The **−l** option causes *azel* to ask for the observer's latitude, west-longitude, and height above sea level.

For each satellite given the program types its full name, the date, and a sequence of lines each containing a time, an azimuth, an elevation, a distance, and a visual magnitude. Each such line indicates that: at the indicated time, the satellite may be seen from Murray Hill (or provided location) at the indicated azimuth and elevation, and that its distance and apparent magnitude are as given. Predictions are printed only when the sky is dark (sun more than 5 degrees below the horizon) and when the satellite is not eclipsed by the earth's shadow. Satellites which have not been seen and verified will not have had their visual magnitude level set correctly.

All times input and output by *azel* are GMT (Universal Time).

The satellites for which elements are maintained are:

sla,b,e,f,k — Skylab A through Skylab K. Skylab A is the laboratory; B was the rocket but it has crashed. A and probably K have been verified.

cop — Copernicus I. Never verified.

oao — Orbiting Astronomical Observatory. Seen and verified.

pag — Pageos I. Seen and verified; fairly dim (typically 2nd-3rd magnitude), but elements are extremely accurate.

exp19 — Explorer 19; seen and verified, but quite dim (4th-5th magnitude) and fast-moving.

c103b, c156b, c184b, c206b, c220b, c461b, c500b — Various of the USSR Cosmos series; none seen.

7276a — Unnamed (satellite # 72-76A); not seen.

The element files used by *azel* contain five lines. The first line gives a year, month number, day, hour, and minute at which the program begins its consideration of the satellite, followed by a number of minutes and an interval in minutes. If the year, month, and day are 0, they are taken to be the current date (taken to change at 6 A.M. local time). The output report starts at the indicated epoch and prints the position of the satellite for the indicated number of minutes at times separated by the indicated interval. This line is ended by two numbers which specify options to the program governing the completeness of the report; they are ordinarily both ''1''. The first option flag suppresses output when the sky is not dark; the second supresses output when the satellite is eclipsed by the earth's shadow. The next line of an element file is the full name of the satellite. The next three are the elements themselves (including certain derivatives of the elements).

FILES

/usr/jfo/el/* – orbital element files

SEE ALSO

sky (VI)

AUTHOR

J. F. Ossanna

BUGS

NAME

bj – the game of black jack

SYNOPSIS

/usr/games/bj

DESCRIPTION

Bj is a serious attempt at simulating the dealer in the game of black jack (or twenty-one) as might be found in Reno. The following rules apply:

The bet is $2 every hand.

A player 'natural' (black jack) pays $3. A dealer natural loses $2. Both dealer and player naturals is a 'push' (no money exchange).

If the dealer has an ace up, the player is allowed to make an 'insurance' bet against the chance of a dealer natural. If this bet is not taken, play resumes as normal. If the bet is taken, it is a side bet where the player wins $2 if the dealer has a natural and loses $1 if the dealer does not.

If the player is dealt two cards of the same value, he is allowed to 'double'. He is allowed to play two hands, each with one of these cards. (The bet is doubled also; $2 on each hand.)

If a dealt hand has a total of ten or eleven, the player may 'double down'. He may double the bet ($2 to $4) and receive exactly one more card on that hand.

Under normal play, the player may 'hit' (draw a card) as long as his total is not over twenty-one. If the player 'busts' (goes over twenty-one), the dealer wins the bet.

When the player 'stands' (decides not to hit), the dealer hits until he attains a total of seventeen or more. If the dealer busts, the player wins the bet.

If both player and dealer stand, the one with the largest total wins. A tie is a push.

The machine deals and keeps score. The following questions will be asked at appropriate times. Each question is answered by **y** followed by a new line for 'yes', or just new line for 'no'.

? (means, "do you want a hit?")
Insurance?
Double down?

Every time the deck is shuffled, the dealer so states and the 'action' (total bet) and 'standing' (total won or lost) is printed. To exit, hit the interrupt key (DEL) and the action and standing will be printed.

BUGS

NAME

cal – print calendar

SYNOPSIS

cal [month] year

DESCRIPTION

Cal prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. *Year* can be between 1 and 9999. The *month* is a number between 1 and 12. The calendar produced is that for England and her colonies.

Try September 1752.

BUGS

The year is always considered to start in January even though this is historically naive.

NAME

chess – the game of chess

SYNOPSIS

/usr/games/chess

DESCRIPTION

Chess is a computer program that plays class D chess. Moves may be given either in standard (descriptive) notation or in algebraic notation. The symbol '+' is used to specify check; 'o-o' and 'o-o-o' specify castling. To play black, type 'first'; to print the board, type an empty line.

Each move is echoed in the appropriate notation followed by the program's reply.

FILES

/usr/lib/book opening 'book'

DIAGNOSTICS

The most cryptic diagnostic is 'eh?' which means that the input was syntactically incorrect.

WARNING

Over-use of this program will cause it to go away.

BUGS

Pawns may be promoted only to queens.

NAME

col – filter reverse line feeds

SYNOPSIS

col

DESCRIPTION

Col reads the standard input and writes the standard output. It performs the line overlays implied by reverse line feeds (ascii code ESC-7). *Col* is particularly useful for filtering multicolumn output made with the '.rt' command of *nroff*.

SEE ALSO

nroff (I)

BUGS

Can't back up more than 102 lines.

-

NAME

cubic – three dimensional tic-tac-toe

SYNOPSIS

/usr/games/cubic

DESCRIPTION

Cubic plays the game of three dimensional 4×4×4 tic-tac-toe. Moves are given by the three digits (each 1-4) specifying the coordinate of the square to be played.

WARNING

Too much playing of the game will cause it to disappear.

BUGS

NAME

factor – discover prime factors of a number

SYNOPSIS

factor

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 7.2×10^{16}) 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^{13}.

DIAGNOSTICS

'Ouch.' for input out of range or for garbage input.

BUGS

NAME

fed – edit form letter memory

SYNOPSIS

fed

DESCRIPTION

Fed is used to edit a form letter associative memory file, **form.m,** which consists of named strings. Commands consist of single letters followed by a list of string names separated by a single space and ending with a new line. The conventions of the Shell with respect to '*' and '?' hold for all commands but **m**. The commands are:

e name ...
Fed writes the string whose name is *name* onto a temporary file and executes *ed.* On exit from the *ed* the temporary file is copied back into the associative memory. Each argument is operated on separately. Be sure to give an editor *w* command (without a filename) to rewrite *fed's* temporary file before quitting out of *ed.*

d [name ...]
deletes a string and its name from the memory. When called with no arguments **d** operates in a verbose mode typing each string name and deleting only if a **y** is typed. A **q** response returns to *fed*'s command level. Any other response does nothing.

m name1 name2 ...
(move) changes the name of name1 to name2 and removes previous string name2 if one exists. Several pairs of arguments may be given. Literal strings are expected for the names.

n [name ...]
(names) lists the string names in the memory. If called with the optional arguments, it just lists those requested.

p name ...
prints the contents of the strings with names given by the arguments.

q
returns to the system.

c [**p**] [**f**]
checks the associative memory file for consistency and reports the number of free headers and blocks. The optional arguments do the following:

p causes any unaccounted-for string to be printed.

f fixes broken memories by adding unaccounted-for headers to free storage and removing references to released headers from associative memory.

FILES

/tmp/ftmp?	temporary
form.m	associative memory

SEE ALSO

form (VI), ed (I), sh (I)

WARNING

It is legal but unwise to have string names with blanks, '*' or '?' in them.

BUGS

NAME

form – form letter generator

SYNOPSIS

form proto arg ...

DESCRIPTION

Form generates a form letter from a prototype letter, an associative memory, arguments and in a special case, the current date.

If *form* is invoked with the *proto* argument *x*, the associative memory is searched for an entry with name *x* and the contents filed under that name are used as the prototype. If the search fails, the message '[*x*]:' is typed on the console and whatever text is typed in from the console, terminated by two new lines, is used as the prototype. If the prototype argument is missing, '{letter}' is assumed.

Basically, *form* is a copy process from the prototype to the output file. If an element of the form [*n*] (where *n* is a digit from 1 to 9) is encountered, the *n*-th *arg* is inserted in its place, and that argument is then rescanned. If [0] is encountered, the current date is inserted. If the desired argument has not been given, a message of the form '[*n*]:' is typed. The response typed in then is used for that argument.

If an element of the form [*name*] or {*name*} is encountered, the *name* is looked up in the associative memory. If it is found, the contents of the memory under this *name* replaces the original element (again rescanned). If the *name* is not found, a message of the form '[*name*]:' is typed. The response typed in is used for that element. The response is entered in the memory under the name if the name is enclosed in [ ]. The response is not entered in the memory but is remembered for the duration of the letter if the name is enclosed in { }. Brackets and braces may be nested.

In both of the above cases, the response is typed in by entering arbitrary text terminated by two new lines. Only the first of the two new lines is passed with the text.

If one of the special characters [{]}\ is preceded by a \, it loses its special character.

If a file named 'forma' already exists in the user's directory, 'formb' is used as the output file and so forth to 'formz'.

The file 'form.m' is created if none exists. Because form.m is operated on by the disc allocator, it should only be changed by using *fed,* the form letter editor, or *form.*

FILES

form.m associative memory
form? output file (read only)

SEE ALSO

fed (VI), roff (I)

BUGS

An unbalanced] or } acts as an end of file but may add a few strange entries to the associative memory.

NAME

graph – draw a graph

SYNOPSIS

graph [option] ... | plotter

DESCRIPTION

Graph with no options takes pairs of numbers from the standard input as abscissas and ordinates of a graph. The graph is written on the standard output to be piped to the plotter program for a particular device; see *plot* (VI). These plotters exist: *gsip,* for the GSI and other Diablo terminals; *tek,* for the Tektronix 4014 terminal; and *vt0* for the on-line storage scope.

The following options are recognized, each as a separate argument.

a Supply abscissas automatically (they are missing from the input); spacing is given by the next argument, or is assumed to be 1 if next argument is not a number. A second optional argument is the starting point for the automatic abscissa.

c Place character string given by next argument at each point.

d Omit connections between points. (Disconnect.)

g*n* Grid style:
n=0, no grid
n=1, axes only
n=2, complete grid (default).

s Save screen, don't erase before plotting.

x Next 1 (or 2) arguments are lower (and upper) *x* limits.

y Next 1 (or 2) arguments are lower (and upper) *y* limits.

h Next argument is fraction of space for height

w Next argument is fraction of space for width.

r Next argument is fraction of space to move right before plotting.

u Next argument is fraction of space to move up before plotting.

Points are connected by straight line segments in the order they appear in input. If a specified lower limit exceeds the upper limit, or if the automatic increment is negative, the graph is plotted upside down. Automatic abscissas begin with the lower *x* limit, or with 0 if no limit is specified. Grid lines and automatically determined limits fall on round values, however roundness may be subverted by giving an inappropriately rounded lower limit. Plotting symbols specified by **c** are placed so that a small initial letter, such as + o x, will fall approximately on the plotting point.

SEE ALSO

spline (VI), plot (VI)

BUGS

A limit of 1000 points is enforced silently.

NAME

gsi – interpret extended character set on GSI terminal

SYNOPSIS

gsi

DESCRIPTION

Gsi interprets special characters understood by the Model 37 Teletype terminal and turns them into the escape sequences understood by the GSI and other Diablo-based terminals. The things interpreted include vertical motions and extended graphic characters. It is most often used in a pipeline like

neqn file ... | nroff | gsi

SEE ALSO

greek (V)

BUGS

Some funny characters can't be correctly printed in column 1 because you can't move to the left from there.

NAME

m6 – general purpose macroprocessor

SYNOPSIS

m6 [name]

DESCRIPTION

M6 copies the standard input to the standard output, with substitutions for any macro calls that appear. When a file name argument is given, that file is read before the standard input.

The processor is as described in the reference with these exceptions:

#def,arg1,arg2,arg3: causes *arg1* to become a macro with defining text *arg2* and (optional) built-in serial number *arg3*.

#del,arg1: deletes the definition of macro *arg1*.

#end: is not implemented.

#list,arg1: sends the name of the macro designated by *arg1* to the current destination without recognition of any warning characters; *arg1* is 1 for the most recently defined macro, 2 for the next most recent, and so on. The name is taken to be empty when *arg1* doesn't make sense.

#warn,arg1,arg2: replaces the old warning character *arg1* by the new warning character *arg2*.

#quote,arg1: sends the definition text of macro *arg1* to the current destination without recognition of any warning characters.

#serial,arg1: delivers the built-in serial number associated with macro *arg1*.

#source,arg1: is not implemented.

#trace,arg1: with *arg1* = '1' causes a reconstruction of each later call to be placed on the standard output with a call level number; other values of *arg1* turn tracing off.

The built-in 'warn' may be used to replace inconvenient warning characters. The example below replaces '#' ':' '<' '>' by '[' ']' '{' '}'.

```
#warn,<#>,[:
[warn,<:>,]:
[warn,[substr,<<>>,1,1;,{]
[warn,[substr,{{>>,2,1;,}]
[now,{calls look like this}]
```

Every built-in function has a serial number, which specifies the action to be performed before the defining text is expanded. The serial numbers are: 1 gt, 2 eq, 3 ge, 4 lt, 5 ne, 6 le, 7 seq, 8 sne, 9 add, 10 sub, 11 mpy, 12 div, 13 exp, 20 if, 21 def, 22 copy, 23 warn, 24 size, 25 substr, 26 go, 27 gobk, 28 del, 29 dnl, 32 quote, 33 serial, 34 list, 35 trace. Serial number 0 specifies no built-in action.

SEE ALSO

A. D. Hall, M6 Reference Manual. Computer Science Technical Report #2, Bell Laboratories, 1969.

DIAGNOSTICS

Various table overflows and ''impossible'' conditions result in comment and dump. There are no diagnostics for poorly formed input.

AUTHOR

M. D. McIlroy

BUGS

Provision should be made to extend tables as needed, instead of wasting a big fixed core allocation. You get what the PDP11 gives you for arithmetic.

-

NAME

moo – guessing game

SYNOPSIS

/usr/games/moo

DESCRIPTION

Moo is a guessing game imported from England. The computer picks a number consisting of four distinct decimal digits. The player guesses four distinct digits being scored on each guess. A ‘cow’ is a correct digit in an incorrect position. A ‘bull’ is a correct digit in a correct position. The game continues until the player guesses the number (a score of four bulls).

BUGS

NAME

plot: tek, gsip, vt0 – graphics filters

SYNOPSIS

source | **tek**
source | **gsip**
source | **vt0**

DESCRIPTION

These commands produce graphical output on the Tektronix 4014 terminal, the GSI or other Diablo-mechanism terminals, and the on-line storage scope respectively. They read the standard input to obtain plotting instructions, which are usually generated by a program calling the graphics subroutines described in *plot* (VII). Each instruction consists of an ASCII letter usually followed by binary information. A plotting coordinate is transmitted as four bytes representing the *x* and *y* values; each value is a signed number transmitted low-order byte first. The assumed plotting space is set by request. The instructions are taken from

m move: the next four bytes specify the coordinates of a point to move to. This is used before writing a label.

p point: the next four bytes specify the coordinates at which a point is drawn.

l line: the next eight bytes are taken as two pairs of coordinates specifying the endpoints of a line to be drawn.

t label: the bytes up to a new-line are written as ASCII starting at the last point drawn or moved to.

a arc: the first four bytes specify the center, the next four specify the starting point, and the last four specify 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. This command is not necessarily implemented on all (or even any) of the output devices.

c circle: The first four bytes specify the center of the circle, the next two the radius.

e erases the screen

f linemod: takes the following string as the type for all future lines. The types are ‘dotted,’ ‘solid,’ ‘longdashed,’ ‘shortdashed,’ and ‘dotdashed.’ This instruction is effective only with the Tektronix terminal.

d dotline: takes the first four bytes as the coordinates of the beginning of a dotted line. The next two are a signed x-increment, and the next two are a word count. Following are the indicated number of bytepairs representing words. For each bit in this list of words a point is plotted which is visible if the bit is ‘1,’ invisible if not. Each point is offset rightward by the x-increment. The instruction is effective only on the vt0 scope.

SEE ALSO

plot (VII), graph (VI)

BUGS

NAME

primes − print all primes larger than somewhat

SYNOPSIS

primes

DESCRIPTION

When *primes* is invoked, it waits for a number to be typed in. If you type in a positive number less than 2^{56} (about 7.2×10^{16}) it will print all primes greater than or equal to this number.

DIAGNOSTICS

'Ouch.' for input out of range or for garbage input.

BUGS

NAME

quiz – test your knowledge

SYNOPSIS

quiz [–**i** file] [–**t**] [category1 category2]

DESCRIPTION

Quiz gives associative knowledge tests on various subjects. It asks items chosen from *category1* and expects answers from *category2*. If no categories are specified, *quiz* gives instructions and lists the available categories.

Quiz tells a correct answer whenever you type a bare newline. At the end of input, upon interrupt, or when questions run out, *quiz* reports a score and terminates.

The –**t** flag specifies 'tutorial' mode, where missed questions are repeated later, and material is gradually introduced as you learn.

The –**i** flag causes the named file to be substituted for the default index file. The lines of these files have the syntax:

line	= category newline | category ':' line
category	= alternate | category '|' alternate
alternate	= empty | alternate primary
primary	= character | '[' category ']' | option
option	= '{' category '}'

The first category on each line of an index file names an information file. The remaining categories specify the order and contents of the data in each line of the information file. Information files have the same syntax. Backslash '\' is used as with *sh* (I) to quote syntactically significant characters or to insert transparent newlines into a line. When either a question or its answer is empty, *quiz* will refrain from asking it.

FILES

/usr/lib/quiz/index
/usr/lib/quiz/*

BUGS

NAME

sky – obtain ephemerides

SYNOPSIS

sky [**−l**]

DESCRIPTION

Sky predicts the apparent locations of the Sun, the Moon, the planets out to Saturn, stars of magnitude at least 2.5, and certain other celestial objects. *Sky* reads the standard input to obtain a GMT time typed on one line with blanks separating year, month number, day, hour, and minute; if the year is missing the current year is used. If a blank line is typed the current time is used. The program prints the azimuth, elevation, and magnitude of objects which are above the horizon at the ephemeris location of Murray Hill at the indicated time. The '−l' flag causes it to ask for another location.

Placing a ''1'' input after the minute entry causes the program to print out the Greenwich Sidereal Time at the indicated moment and to print for each body its topographic right ascension and declination as well as its azimuth and elevation. Also, instead of the magnitude, the semidiameter of the body, in seconds of arc, is reported.

A ''2'' after the minute entry makes the coordinate system geocentric.

The effects of atmospheric extinction on magnitudes are not included; the brightest magnitudes of variable stars are marked with ''*''.

For all bodies, the program takes into account precession and nutation of the equinox, annual (but not diurnal) aberration, diurnal parallax, and the proper motion of stars. In no case is refraction included.

The program takes into account perturbations of the Earth due to the Moon, Venus, Mars, and Jupiter. The expected accuracies are: for the Sun and other stellar bodies a few tenths of seconds of arc; for the Moon (on which particular care is lavished) likewise a few tenths of seconds. For the Sun, Moon and stars the accuracy is sufficient to predict the circumstances of eclipses and occultations to within a few seconds of time. The planets may be off by several minutes of arc.

There are lots of special options not described here, which do things like substituting named star catalogs, smoothing nutation and aberration to aid generation of mean places of stars, and making conventional adjustments to the Moon to improve eclipse predictions.

For the most accurate use of the program it is necessary to know that it actually runs in Ephemeris time.

FILES

/usr/lib/startab, /usr/lib/moontab

SEE ALSO

azel (VI)
American Ephemeris and Nautical Almanac, for the appropriate years; also, the *Explanatory Supplement to the American Ephemeris and Nautical Almanac.*

AUTHOR

R. Morris

BUGS

NAME

sno – Snobol interpreter

SYNOPSIS

sno [file]

DESCRIPTION

Sno is a Snobol III (with slight differences) compiler and interpreter. *Sno* obtains input from the concatenation of *file* and the standard input. All input through a statement containing the label 'end' is considered program and is compiled. The rest is available to 'syspit'.

Sno differs from Snobol III in the following ways.

There are no unanchored searches. To get the same effect:

```
a ** b          unanchored search for b
a *x* b = x c   unanchored assignment
```

There is no back referencing.

```
x = "abc"
a *x* x         is an unanchored search for 'abc'
```

Function declaration is different. The function declaration is done at compile time by the use of the label 'define'. Thus there is no ability to define functions at run time and the use of the name 'define' is preempted. There is also no provision for automatic variables other than the parameters. For example:

define f( )

or

define f(a,b,c)

All labels except 'define' (even 'end') must have a non-empty statement.

If 'start' is a label in the program, program execution will start there. If not, execution begins with the first executable statement. 'define' is not an executable statement.

There are no builtin functions.

Parentheses for arithmetic are not needed. Normal precedence applies. Because of this, the arithmetic operators '/' and '*' must be set off by space.

The right side of assignments must be non-empty.

Either ´ or " may be used for literal quotes.

The pseudo-variable 'sysppt' is not available.

SEE ALSO

Snobol III manual. (JACM; Vol. 11 No. 1; Jan 1964; pp 21)

BUGS

NAME

speak – word to voice translator

SYNOPSIS

speak [–**efpsv**] [vocabulary [output]]

DESCRIPTION

Speak turns a stream of words into utterances and outputs them to a voice synthesizer, or to the specified *output.* It has facilities for maintaining a vocabulary. It receives, from the standard input

- working lines: text of words separated by blanks
- phonetic lines: strings of phonemes for one word preceded and separated by commas. The phonemes may be followed by comma-percent then a 'replacement part' – an ASCII string with no spaces. The phonetic code is given in *vs* (V).
- empty lines
- command lines: beginning with **!.** The following command lines are recognized:

!r file	replace coded vocabulary from file
!w file	write coded vocabulary on file
!p	print phonetics for working word
!l	list vocabulary on standard output with phonetics
!c word	copy phonetics from working word to specified word
!d	print decomposition of working word into substrings
!f *n*	turn off (or on) English preprocessing rule number *n* (see listing for meaning of *n)*

Each working line replaces its predecessor. Its first word is the 'working word'. Each phonetic line replaces the phonetics stored for the working word. In particular, a phonetic line of comma only deletes the entry for the working word. Each working line, phonetic line or empty line causes the working line to be uttered. The process terminates at the end of input.

Unknown words are pronounced by rules, and failing that, are spelled. For the builtin part of the rules, see the reference. Spelling is done by taking each character of the word, prefixing it with '*', and looking it up. Unspellable words burp.

Words not found verbatim in the vocabulary are pronounced piecewise. First the word is bracketed by sharps: '#...#'. The vocabulary is then searched for the longest fragment that matches the beginning of the word. The phonetic part of the phonetic string is uttered, and the matched fragment is replaced by the replacement part of the phonetic string, if any. The process is repeated until the word is exhausted. A fragment is entered into the vocabulary as a working word prefixed by '%'.

Speak is initialized with a coded vocabulary stored in file */usr/lib/speak.m.* The vocabulary option substitutes a different file for */usr/lib/speak.m.* Other vocabularies, to be used with option –**e**, exist in /usr/vs/latin.m and /usr/vs/polish.m.

A set of single letter options may appear in any order preceded by –**.** Their meanings are:

e	suppress English preprocessing
f	equivalent to 'f1, f2,...'
p	suppress pronunciation by rule
s	suppress spelling
v	suppress voice output

The following input will reconstitute a coded vocabulary, 'speak.m', from an ascii listing, 'speak.v', that was created using **!l**.

```
(cat speak.v; echo !w speak.m) | speak –v /dev/null
```

FILES

/usr/lib/speak.m

SEE ALSO

M. D. McIlroy, ''Synthetic English Speech by Rule,'' Computing Science Technical Report #14, Bell Laboratories, 1973

vs (V), vs (IV)

BUGS

Excessively long words cause dumps.
Space is not reclaimed from changed entries; use **!w** and **!r** to effect reclamation.
!p doesn't always work as advertised.

NAME

spline – interpolate smooth curve

SYNOPSIS

spline [option] ...

DESCRIPTION

Spline takes pairs of numbers from the standard input as abcissas and ordinates of a function. It produces a similar set, which is approximately equally spaced and includes the input set, on the standard output. The cubic spline output (R. W. Hamming, *Numerical Methods for Scientists and Engineers,* 2nd ed., 349ff) has two continuous derivatives, and sufficiently many points to look smooth when plotted, for example by *plot* (I).

The following options are recognized, each as a separate argument.

a Supply abscissas automatically (they are missing from the input); spacing is given by the next argument, or is assumed to be 1 if next argument is not a number.

k The constant k used in the boundary value computation

$$y_0'' = ky_1'', \quad y_n'' = ky_{n-1}''$$

is set by the next argument. By default $k = 0$.

n Space output points so that approximately n points occur between the lower and upper x limits. (Default $n = 100$.)

p Make output periodic, i.e. match derivatives at ends. First and last input values should normally agree.

x Next 1 (or 2) arguments are lower (and upper) x limits. Normally these limits are calculated from the data. Automatic abcissas start at lower limit (default 0).

SEE ALSO

plot (I)

AUTHOR

M. D. McIlroy

BUGS

A limit of 1000 input points is enforced silently.

NAME

tbl – format tables for nroff or troff

SYNOPSIS

tbl [files] ...

DESCRIPTION

Tbl is an nroff (I) or troff(I) preprocessor for formatting tables. 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. The first line after .TS specifies the various columns: it consists of a list of column describers separated by blanks or tabs. Each column describer is a character string made up of the letters 'n', 'r', 'c', 'l' and 's', which mean:

c center within the column

r right-adjust

l left-adjust

n numerical adjustment: the units digits of numbers are aligned.

s span the previous entry over this column.

The column describer may be followed by an integer giving the number of spaces between this column and the next; 3 is default. The describer 'ccr5' indicates that the first two lines in this column are centered; the third and remaining lines are right-adjusted; and the column should be separated from the column to the right by 5 spaces. Letting \t represent a tab (which must be typed as a genuine tab) the input

```
.TS
cccl sccn sscn
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.

BUGS

No column describer may end with 's'.

NAME

tmg – compiler-compiler

SYNOPSIS

tmg name

DESCRIPTION

Tmg produces a translator for the language whose parsing and translation rules are described in file *name***.t**. The new translator appears in a.out and may be used thus:

a.out input [output]

Except in rare cases input must be a randomly addressable file. If no output file is specified, the standard output file is assumed.

FILES

*name***.s**: assembly language version of *name***.t**
/usr/lib/tmg: the compiler-compiler
/usr/lib/tmg[abc], /lib/libs.a: libraries
alloc.d: scratch file for table storage

SEE ALSO

A Manual for the Tmg Compiler-writing Language, internal memorandum.

DIAGNOSTICS

Syntactic errors result in "???" followed by the offending line.
Situations such as space overflow with which the Tmg processor or a Tmg-produced processor can not cope result in a descriptive comment and a dump.

AUTHOR

M. D. McIlroy

BUGS

Footnote 1 of Section 9.2 of Tmg Manual is not enforced, causing trouble.
Restrictions (7.) against mixing bundling primitives should be lifted.
Certain hidden reserved words exist: gpar, classtab, trans, goto, alt, salt.
Octal digits include 8=10 and 9=11.

-

NAME

ttt – the game of tic-tac-toe

SYNOPSIS

/usr/games/ttt

DESCRIPTION

Ttt is the X and O game popular in the first grade. This is a learning program that never makes the same mistake twice.

Although it learns, it learns slowly. It must lose nearly 80 games to completely know the game.

FILES

/usr/games/ttt.k learning file

BUGS

NAME

units – conversion program

SYNOPSIS

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'. For a complete list of units, 'cat /usr/lib/units'.

FILES

/usr/lib/units

BUGS

NAME

wump – the game of hunt-the-wumpus

SYNOPSIS

/usr/games/wump

DESCRIPTION

Wump plays the game of ‘‘Hunt the Wumpus.’’ A Wumpus is a creature that lives in a cave with several rooms connected by tunnels. You wander among the rooms, trying to shoot the Wumpus with an arrow, meanwhile avoiding being eaten by the Wumpus and falling into Bottomless Pits. There are also Super Bats which are likely to pick you up and drop you in some random room.

The program asks various questions which you answer one per line; it will give a more detailed description if you want.

This program is based on one described in *People's Computer Company, 2,* 2 (November 1973).

BUGS

It will never replace Space War.

NAME

crfork, crexit, crread, crwrite, crexch, crprior – coroutine scheme

SYNOPSIS

int crfork([stack, nwords])
int stack[];
int nwords;

crexit()

int crread(connector, buffer, nbytes)
int *connector[2];
char *buffer;
int nbytes;

crwrite(connector, buffer, nbytes)
int *connector[2];
char *buffer;
int nbytes;

crexch(conn1, conn2, i)
int *conn1[2], *conn2[2];
int i;

#define logical char *
crprior(p)
logical p;

DESCRIPTION

These functions are named by analogy to *fork, exit, read, write* (II). They establish and synchronize 'coroutines', which behave in many respects like a set of processes working in the same address space. The functions live in */usr/lib/cr.a.*

Coroutines are placed on queues to indicate their state of readiness. One coroutine is always distinguished as 'running'. Coroutines that are runnable but not running are registered on a 'ready queue'. The head member of the ready queue is started whenever no other coroutine is specifically caused to be running.

Each connector heads two queues: *Connector[0]* is the queue of unsatisfied *crreads* outstanding on the connector. *Connector[1]* is the queue of *crwrites*. All queues must start empty, *i.e.* with heads set to zero.

Crfork is normally called with no arguments. It places the running coroutine at the head of the ready queue, creates a new coroutine, and starts the new one running. *Crfork* returns immediately in the new coroutine with value 0, and upon restarting of the old coroutine with value 1.

Crexit stops the running coroutine and does not place it in any queue.

Crread copies characters from the *buffer* of the *crwrite* at the head of the *connector's* write queue to the *buffer* of *crread.* If the write queue is empty, copying is delayed and the running coroutine is placed on the read queue. The number of characters copied is the minimum of *nbytes* and the number of characters remaining in the write *buffer,* and is returned as the value of *crread.* After copying, the location of the write *buffer* and the corresponding *nbytes* are updated appropriately. If zero characters remain, the coroutine of the *crwrite* is moved to the head of the ready queue. If the write queue remains nonempty, the head member of the read queue is moved to the head of the ready queue.

Crwrite queues the running coroutine on the *connector's* write queue, and records the fact that *nbytes* (zero or more) characters in the string *buffer* are available to *crreads*. If the read queue is not empty, its head member is started running.

Crexch exchanges the read queues of connectors *conn1* and *conn2* if *i*=0; and it exchanges the write queues if *i*=1. If a nonempty read queue that had been paired with an empty write queue becomes paired with a nonempty write queue, *crexch* moves the head member of that read queue to the head of the ready queue.

Crprior sets a priority on the running coroutine to control the queuing of *crreads* and *crwrites*. When queued, the running coroutine will take its place before coroutines whose priorities exceed its own priority and after others. Priorities are compared as logical, *i.e.* unsigned, quantities. Initially each coroutine's priority is set as large as possible, so default queuing is FIFO.

Storage allocation. The old and new coroutine share the same activation record in the function that invoked *crfork,* so only one may return from the invoking function, and then only when the other has completed execution in that function.

The activation record for each function execution is dynamically allocated rather than stacked; a factor of 3 in running time overhead can result if function calls are very frequent. The overhead may be overcome by providing a separate stack for each coroutine and dispensing with dynamic allocation. The base (lowest) address and size of the new coroutine's stack are supplied to *crfork* as optional arguments *stack* and *nwords.* Stacked allocation and dynamic allocation cannot be mixed in one run. For stacked operation, obtain the coroutine functions from */usr/lib/scr.a* instead of */usr/lib/cr.a.*

FILES

/usr/lib/cr.a
/usr/lib/scr.a

DIAGNOSTICS

'rsave doesn't work' − an old C compilation has called 'rsave'. It must be recompiled to work with the coroutine scheme.

BUGS

Under /usr/lib/cr.a each function has just 12 words of anonymous stack for hard expressions and arguments of further calls, regardless of actual need. There is no checking for stack overflow.
Under /usr/lib/scr.a stack overflow checking is not rigorous.

NAME

ms – macros for formatting manuscripts

SYNOPSIS

nroff –ms [options] file ...
troff –ms [options] file ...

DESCRIPTION

This package of *nroff* and *troff* macro definitions provides a canned formatting facility for technical papers. When producing 2-column output on a terminal, its output should be filtered through *col* (I).

The package supports three different formats: BTL technical memorandum with cover sheet, released paper with cover sheet, and an abbreviated 'debugging' form without cover sheet.

The macro requests are defined in the attached Request Reference. Many *nroff* and *troff* requests are unsafe in conjunction with this package, however the requests listed below may be used with impunity after the first .PP.

.bp	begin new page
.br	break output line here
.sp n	insert n spacing lines
.ls n	(line spacing) n=1 single, n=2 double space
.na	no alignment of right margin

Output of the *eqn, neqn* and *tbl* (I) preprocessors for equations and tables is acceptable as input.

FILES

/usr/lib/tmac.s

SEE ALSO

eqn (I), nroff (I), troff (I), tbl (VI)

BUGS

REQUEST REFERENCE

Request Form	Initial Value	Cause Break	Explanation
.1C	yes	yes	One column format on a new page.
.2C	no	yes	Two column format.
.AB	no	yes	Begin abstract.
.AE	-	yes	End abstract.
.AI	no	yes	Author's institution follows. Suppressed in TM.
.AU *x y*	no	yes	Author's name follows. *x* is location and *y* is extension, ignored except in TM.
.B	no	no	Boldface text follows.
.CS *x...*	-	yes	Cover sheet info if TM format, suppressed otherwise. Arguments are number of text pages, other pages, total pages, figures, tables, references.
.DA *x*	nroff	no	'Date line' at bottom of page is *x*. Default is today.
.DE	-	yes	End displayed text. Implies .KE.
.DS *x*	no	yes	Start of displayed text, to appear verbatim line-by-line. *x*=I for indented display (default), *x*=L for left-justified on the page, *x*=C for centered. Implies .KS.
.EN	-	yes	Space after equation produced by *eqn* or *neqn.*
.EQ *x*	-	yes	Space before equation. Equation number is *x*.
.FE	-	yes	End footnote.
.FS	no	no	Start footnote. The note will be moved to the bottom of the page.
.HO	-	no	'Bell Laboratories, Holmdel, New Jersey 07733'.
.I	no	no	Italic text follows.
.IP *x y*	no	yes	Start indented paragraph, with hanging tag *x*. Indentation is *y* ens (default 5).
.KE	-	yes	End keep. Put kept text on next page if not enough room.
.KF	no	yes	Start floating keep. If the kept text must be moved to the next page, float later text back to this page.
.KS	no	yes	Start keeping following text.
.LG	no	no	Make letters larger.
.LP	yes	yes	Start left-blocked paragraph.
.MH	-	no	'Bell Laboratories, Murray Hill, New Jersey 07974'.
.ND	troff	no	No date line at bottom of page.
.NH *n*	-	yes	Same as .SH, with section number supplied automatically. Numbers are multilevel, like 1.2.3, where *n* tells what level is wanted (default is 1).
.NL	yes	no	Make letters normal size.
.OK	-	yes	'Other keywords' for TM cover sheet follow.
.PP	no	yes	Begin paragraph. First line indented.
.R	yes	no	Roman text follows.
.RE	-	yes	End relative indent level.
.RP	no	-	Cover sheet and first page for released paper. Must precede other requests.
.RS	-	yes	Start level of relative indentation. Following .IP's measured from current indentation.
.SG *x*	no	yes	Insert signature(s) of author(s), ignored except in TM. *x* is the reference line (initials of author and typist).
.SH	-	yes	Section head follows, font automatically bold.
.SM	no	no	Make letters smaller.
.TL	no	yes	Title follows.
.TM *x y z*	no	-	BTL TM cover sheet and first page, *x*=TM number, *y*=(quoted list of) case number(s), *z*=file number. Must precede other requests.
.WH	-	no	'Bell Laboratories, Whippany, New Jersey 07981'.

NAME

plot: openpl et al. – graphics interface

SYNOPSIS

openpl( )

erase( )

label(s)
char s[ ];

line(x1, y1, x2, y2)

circle(x, y, r)

arc(x, y, x0, y0, x1, y1)

dot(x, y, dx, n, pattern)
int pattern[ ];

move(x, y)

point(x, y)

linemod(s)
char s[ ];

space(x0, y0, x1, y1)

closepl( )

DESCRIPTION

These subroutines generate graphic output in a relatively device-independent manner. See *plot* (VI) for a description of the meaning of the subroutines.

There are four libraries containing these routines, one that produces general graphics commands on the standard output, and one each for the vt0 storage scope, the Diablo plotting terminal and the Tektronix 4014 terminal. *Openpl* must be used before any of the others to open the device for writing. *Closepl* flushes the output.

FILES

/usr/lib/plot.a	produces output for plotting filters
/usr/lib/vt0.a	produces output on vt0 storage scope
/usr/lib/gsip.a	produces output on Diablo terminal
/usr/lib/tek.a	produces output for the Tektronix 4014 terminal

SEE ALSO

plot (VI), graph (VI)

BUGS

NAME

salloc – string allocation and manipulation

SYNOPSIS

(get size in r0)
jsr pc,allocate
(header address in r1)

(get source header address in r0,
destination header address in r1)
jsr pc,copy

jsr pc,wc

(all following routines assume r1 contains header address)

jsr pc,release

(get character in r0)
jsr pc,putchar

jsr pc,lookchar
(character in r0)

jsr pc,getchar
(character in r0)

(get character in r0)
jsr pc,alterchar

(get position in r0)
jsr pc,seekchar

jsr pc,backspace
(character in r0)

(get word in r0)
jsr pc,putword

jsr pc,lookword
(word in r0)

jsr pc,getword
(word in r0)

(get word in r0)
jsr pc,alterword

jsr pc,backword
(word in r0)

jsr pc,length
(length in r0)

jsr pc,position
(position in r0)

jsr pc,rewind

jsr pc,create

jsr pc,fsfile

jsr pc,zero

DESCRIPTION

This package is a complete set of routines for dealing with almost arbitrary length strings of words and bytes. It lives in */lib/libs.a*. The strings are stored on a disk file, so the sum of their lengths can be considerably larger than the available core. A small buffer cache makes for reasonable speed.

For each string there is a header of four words, namely a write pointer, a read pointer and pointers to the beginning and end of the block containing the string. Initially the read and write pointers point to the beginning of the string. All routines that refer to a string require the header address in r1. Unless the string is destroyed by the call, upon return r1 will point to the same string, although the string may have grown to the extent that it had to be be moved.

Allocate obtains a string of the requested size and returns a pointer to its header in r1.

Release releases a string back to free storage.

Putchar and *putword* write a byte or word respectively into the string and advance the write pointer.

Lookchar and *lookword* read a byte or word respectively from the string but do not advance the read pointer.

Getchar and *getword* read a byte or word respectively from the string and advance the read pointer.

Alterchar and *alterword* write a byte or word respectively into the string where the read pointer is pointing and advance the read pointer.

Backspace and *backword* read the last byte or word written and decrement the write pointer.

All write operations will automatically get a larger block if the current block is exceeded. All read operations return with the error bit set if attempting to read beyond the write pointer.

Seekchar moves the read pointer to the offset specified in r0.

Length returns the current length of the string (beginning pointer to write pointer) in r0.

Position returns the current offset of the read pointer in r0.

Rewind moves the read pointer to the beginning of the string.

Create returns the read and write pointers to the beginning of the string.

Fsfile moves the read pointer to the current position of the write pointer.

Zero zeros the whole string and sets the write pointer to the beginning of the string.

Copy copies the string whose header pointer is in r0 to the string whose header pointer is in r1. Care should be taken in using the copy instruction since r1 will be changed if the contents of the source string is bigger than the destination string.

Wc forces the contents of the internal buffers and the header blocks to be written on disc.

An in-core version of this allocator exists in *dc* (I), and a permanent-file version exists in *form* and *fed* (VI).

FILES

/lib/libs.a	library, accessed by *ld ... -ls*
alloc.d	temporary file for string storage

SEE ALSO

alloc (III)

DIAGNOSTICS

'error in copy' – disk write error encountered in *copy.*
'error in allocator' – routine called with bad header pointer.
'cannot open output file' – temp file *alloc.d* cannot be created or opened.
'out of space' – no sufficiently large block or no header is available for a new or growing block.

BUGS

-

NAME

ac – login accounting

SYNOPSIS

ac [−**w** wtmp] [−**p**] [−**d**] people

DESCRIPTION

Ac produces a printout giving connect time for each user who has logged in during the life of the current *wtmp* file. A total is also produced. −**w** is used to specify an alternate *wtmp* file. −**p** prints individual totals; without this option, only totals are printed. −**d** causes a printout for each midnight to midnight period. Any *people* will limit the printout to only the specified login names. If no *wtmp* file is given, */usr/adm/wtmp* is used.

The accounting file */usr/adm/wtmp* is maintained by *init* and *login.* Neither of these programs creates the file, so if it does not exist no connect-time accounting is done. To start accounting, it should be created with length 0. On the other hand if the file is left undisturbed it will grow without bound, so periodically any information desired should be collected and the file truncated.

FILES

/usr/adm/wtmp

SEE ALSO

init (VIII), login (I), wtmp (V).

BUGS

NAME

boot procedures – UNIX startup

DESCRIPTION

How to start UNIX. UNIX is started by placing it in core at location zero and transferring to zero. Since the system is not reenterable, it is necessary to read it in from disk or tape.

The *tp* command places a bootstrap program on the otherwise unused block zero of the tape. The DECtape version of this program is called *tboot,* the magtape version *mboot.* If *tboot* or *mboot* is read into location zero and executed there, it will type '=' on the console, read in a *tp* entry name, load that entry into core, and transfer to zero. Thus one way to run UNIX is to maintain the system code on a tape using *tp.* Caution: the file /usr/mdec/tboot (DECtape) or /usr/mdec/mboot (magtape) must be present when the tape is made! When a boot is required, execute (somehow) a program which reads in and jumps to the first block of the tape. In response to the '=' prompt, type the entry name of the system on the tape (we use plain 'unix'). It is strongly recommended that a current version of the system be maintained in this way, even if it is usually booted from disk.

The standard DEC ROM which loads DECtape is sufficient to read in *tboot,* but the magtape ROM loads block one, not zero. If no suitable ROM is available, magtape and DECtape programs are presented below which may be manually placed in core and executed.

The system can also be booted from a disk file with the aid of the *uboot* program. When read into location 0 and executed, *uboot* reads a single character (either **p** or **k** for RP or RK, both drive 0) to specify which device is to be searched. Then it reads a UNIX pathname from the console, finds the corresponding file on the given device, loads that file into core location zero, and transfers to it. *Uboot* operates under very severe space constraints. It supplies no prompts, except that it echoes a carriage return and line feed after the **p** or **k.** No diagnostic is provided if the indicated file cannot be found, nor is there any means of correcting typographical errors in the file name except to start the program over. If it fails to find the file, however, it jumps back to its start, so another try can be attempted, starting again with the **p** or **k.** Notice that *uboot* will only load a file from drive 0, and the file system it searches must start at the beginning of the disk. *Uboot* itself usually resides in the otherwise unused block 0 of the disk, so it can be loaded by ROM program; *mkfs* can be used to put it there when the file system is created. It can also be loaded from a *tp* tape as described above.

The switches. The console switches play an important role in the use and especially the booting of UNIX. During operation, the console switches are examined 60 times per second, and the contents of the address specified by the switches are displayed in the display register. (This is not true on the 11/40 since there is no display register on that machine.) If the switch address is even, the address is interpreted in kernel (system) space; if odd, the rounded-down address is interpreted in the current user space.

If any diagnostics are produced by the system, they are printed on the console only if the switches are non-zero. Thus it is wise to have a non-zero value in the switches at all times.

During the startup of the system, the *init* program (VIII) reads the switches and will come up single-user if the switches are set to 173030.

It is unwise to have a non-existent address in the switches. This causes a bus error in the system (displayed as 177777) at the rate of 60 times per second. If there is a transfer of more than 16ms duration on a device with a data rate faster than the bus error timeout (about 10μs) then a permanent disk non-existent-memory error will occur.

ROM programs. Here are some programs which are suitable for installing in read-only memories, or for manual keying into core if no ROM is present. Each program is position-independent but should be placed well above location 0 so it will not be overwritten. Each reads a block from the beginning of a device into core location zero. The octal words constituting the program are listed on the left.

DECtape (drive 0) from endzone:

```
012700          mov     $tcba,r0
177346
010040          mov     r0,–(r0)        / use tc addr for wc
012710          mov     $3,(r0)         / read bn forward
000003
```

```
105710    1:    tstb    (r0)              / wait for ready
002376          bge     1b
112710          movb    $5,(r0)           / read (forward)
000005
000777          br      .                 / loop; now halt and start at 0
```

DECtape (drive 0) with search:

```
012700    1:    mov     $tcba,r0
177346
010040          mov     r0,–(r0)          / use tc addr for wc
012740          mov     $4003,–(r0)       / read bn reverse
004003
005710    2:    tst     (r0)
002376          bge     2b                / wait for error
005760          tst     –2(r0)            / loop if not end zone
177776
002365          bge     1b
012710          mov     $3,(r0)           / read bn forward
000003
105710    2:    tstb    (r0)              / wait for ready
002376          bge     2b
112710          movb    $5,(r0)           / read (forward)
000005
105710    2:    tstb    (r0)              / wait for ready
002376          bge     2b
005007          clr     pc                / transfer to zero
```

Caution: both of these DECtape programs will (literally) blow a fuse if 2 drives are dialed to zero.

Magtape from load point:

```
012700          mov     $mtcma,r0
172526
010040          mov     r0,–(r0)          / usr mt addr for wc
012740          mov     $60003,–(r0)      / read 9–track
060003
000777          br      .                 / loop; now halt and start at 0
```

RK (drive 0):

```
012700          mov     $rkda,r0
177412
005040          clr     –(r0)             / rkda cleared by start
010040          mov     r0,–(r0)
012740          mov     $5,–(r0)
000005
105710    1:    tstb    (r0)
002376          bge     1b
005007          clr     pc
```

RP (drive 0)

```
012700          mov     $rpmr,r0
176726
005040          clr     –(r0)
005040          clr     –(r0)
005040          clr     –(r0)
010040          mov     r0,–(r0)
012740          mov     $5,–(r0)
000005
105710    1:    tstb    (r0)
002376          bge     1b
005007          clr     pc
```

FILES

/unix – UNIX code
/usr/mdec/mboot – *tp* magtape bootstrap
/usr/mdec/tboot – *tp* DECtape bootstrap
/usr/mdec/uboot – file system bootstrap

SEE ALSO

tp (I), init (VIII)

NAME

chgrp – change group

SYNOPSIS

chgrp group file ...

DESCRIPTION

The group-ID of the files is changed to *group*. The group may be either a decimal GID or a group name found in the group-ID file.

Only the super-user is allowed to change the group of a file, in order to simplify as yet unimplemented accounting procedures.

SEE ALSO

chown (VIII)

FILES

/etc/group

BUGS

NAME

chown – change owner

SYNOPSIS

chown owner file ...

DESCRIPTION

The user-ID of the files is changed to *owner*. The owner may be either a decimal UID or a login name found in the password file.

Only the super-user is allowed to change the owner of a file, in order to simplify as yet unimplemented accounting procedures.

FILES

/etc/passwd

SEE ALSO

chgrp (VIII)

BUGS

NAME

clri – clear i-node

SYNOPSIS

clri i-number [filesystem]

DESCRIPTION

Clri writes zeros on the 32 bytes occupied by the i-node numbered *i-number.* If the *file system* argument is given, the i-node resides on the given device, otherwise on a default file system. The file system argument must be a special file name referring to a device containing a file system. After *clri,* any blocks in the affected file will show up as "missing" in an *icheck* of of the file system.

Read and write permission is required on the specified file system device. The i-node becomes allocatable.

The primary purpose of this routine is to remove a file which for some reason appears in no directory. If it is used to zap an i-node which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the i-node is reallocated to some new file, the old entry will still point to that file. At that point removing the old entry will destroy the new file. The new entry will again point to an unallocated i-node, so the whole cycle is likely to be repeated again and again.

BUGS

Whatever the default file system is, it is likely to be wrong. Specify the file system explicitly.

If the file is open, *clri* is likely to be ineffective.

NAME

crash – what to do when the system crashes

DESCRIPTION

This section gives at least a few clues about how to proceed if the system crashes. It can't pretend to be complete.

How to bring it back up. If the reason for the crash is not evident (see below for guidance on 'evident') you may want to try to dump the system if you feel up to debugging. At the moment a dump can be taken only on magtape. With a tape mounted and ready, stop the machine, load address 44, and start. This should write a copy of all of core on the tape with an EOF mark. Caution: Any error is taken to mean the end of core has been reached. This means that you must be sure the ring is in, the tape is ready, and the tape is clean and new. If the dump fails, you can try again, but some of the registers will be lost. See below for what to do with the tape.

In restarting after a crash, always bring up the system single-user. This is accomplished by following the directions in *boot procedures* (VIII) as modified for your particular installation; a single-user system is indicated by having a particular value in the switches (173030 unless you've changed *init)* as the system starts executing. When it is running, perform a *dcheck* and *icheck* (VIII) on all file systems which could have been in use at the time of the crash. If any serious file system problems are found, they should be repaired. When you are satisfied with the health of your disks, check and set the date if necessary, then come up multi-user. This is most easily accomplished by changing the single-user value in the switches to something else, then logging out by typing an EOT.

To even boot UNIX at all, three files (and the directories leading to them) must be intact. First, the initialization program */etc/init* must be present and executable. If it is not, the CPU will loop in user mode at location 6. For *init* to work correctly, */dev/tty8* and */bin/sh* must be present. If either does not exist, the symptom is best described as thrashing. *Init* will go into a *fork/exec* loop trying to create a Shell with proper standard input and output.

If you cannot get the system to boot, a runnable system must be obtained from a backup medium. The root file system may then be doctored as a mounted file system as described below. If there are any problems with the root file system, it is probably prudent to go to a backup system to avoid working on a mounted file system.

Repairing disks. The first rule to keep in mind is that an addled disk should be treated gently; it shouldn't be mounted unless necessary, and if it is very valuable yet in quite bad shape, perhaps it should be dumped before trying surgery on it. This is an area where experience and informed courage count for much.

The problems reported by *icheck* typically fall into two kinds. There can be problems with the free list: duplicates in the free list, or free blocks also in files. These can be cured easily with an *icheck –s*. If the same block appears in more than one file or if a file contains bad blocks, the files should be deleted, and the free list reconstructed. The best way to delete such a file is to use *clri* (VIII), then remove its directory entries. If any of the affected files is really precious, you can try to copy it to another device first.

Dcheck may report files which have more directory entries than links. Such situations are potentially dangerous; *clri* discusses a special case of the problem. All the directory entries for the file should be removed. If on the other hand there are more links than directory entries, there is no danger of spreading infection, but merely some disk space that is lost for use. It is sufficient to copy the file (if it has any entries and is useful) then use *clri* on its inode and remove any directory entries that do exist.

Finally, there may be inodes reported by *dcheck* that have 0 links and 0 entries. These occur on the root device when the system is stopped with pipes open, and on other file systems when the system stops with files that have been deleted while still open. A *clri* will free the inode, and an *icheck -s* will recover any missing blocks.

Why did it crash? UNIX types a message on the console typewriter when it voluntarily crashes. Here is the current list of such messages, with enough information to provide a hope at least of the remedy. The message has the form 'panic: ...', possibly accompanied by other information. Left unstated in all cases is the possibility that hardware or software error produced the message in some unexpected way.

blkdev
: The *getblk* routine was called with a nonexistent major device as argument. Definitely hardware or software error.

devtab
: Null device table entry for the major device used as argument to *getblk*. Definitely hardware or software error.

iinit
: An I/O error reading the super-block for the root file system during initialization.

out of inodes
: A mounted file system has no more i-nodes when creating a file. Sorry, the device isn't available; the *icheck* should tell you.

no fs
: A device has disappeared from the mounted-device table. Definitely hardware or software error.

no imt
: Like 'no fs', but produced elsewhere.

no inodes
: The in-core inode table is full. Try increasing NINODE in param.h. Shouldn't be a panic, just a user error.

no clock
: During initialization, neither the line nor programmable clock was found to exist.

swap error
: An unrecoverable I/O error during a swap. Really shouldn't be a panic, but it is hard to fix.

unlink − iget
: The directory containing a file being deleted can't be found. Hardware or software.

out of swap space
: A program needs to be swapped out, and there is no more swap space. It has to be increased. This really shouldn't be a panic, but there is no easy fix.

out of text
: A pure procedure program is being executed, and the table for such things is full. This shouldn't be a panic.

trap
: An unexpected trap has occurred within the system. This is accompanied by three numbers: a 'ka6', which is the contents of the segmentation register for the area in which the system's stack is kept; 'aps', which is the location where the hardware stored the program status word during the trap; and a 'trap type' which encodes which trap occurred. The trap types are:

0	bus error
1	illegal instruction
2	BPT/trace
3	IOT
4	power fail
5	EMT
6	recursive system call (TRAP instruction)
7	11/70 cache parity, or programmed interrupt
10	floating point trap
11	segmentation violation

In some of these cases it is possible for octal 20 to be added into the trap type; this indicates that the processor was in user mode when the trap occurred. If you wish to examine the stack after such a trap, either dump the system, or use the console switches to examine core; the required address mapping is described below.

Interpreting dumps. All file system problems should be taken care of before attempting to look at dumps. The dump should be read into the file */usr/sys/core; cp* (I) will do. At this point, you should execute *ps −alxk* and *who* to print the process table and the users who were on at the time of the crash. You should dump (*od* (I)) the first 30 bytes of */usr/sys/core.* Starting at location 4, the registers R0, R1, R2, R3, R4, R5, SP and KDSA6 (KISA6 for 11/40s) are stored. If the dump had to be restarted, R0 will not be correct. Next, take the value of KA6 (location 22(8) in the dump) multiplied by 100(8) and dump 1000(8) bytes starting from there. This is the per-process data associated with the process running at the time of the crash. Relabel the addresses 140000 to 141776. R5 is C's frame or display pointer. Stored at (R5) is the old R5 pointing to the previous stack frame. At (R5)+2 is the saved PC of the calling procedure. Trace this calling chain until you obtain an R5 value of 141756, which is where the user's R5 is stored. If the chain is broken, you have to look for a plausible R5, PC pair and continue from there. Each PC should be looked up in the system's name list using *db* (I) and its ':' command, to get a reverse calling order. In most cases this procedure will give an idea of what is wrong. A more complete discussion of system debugging is impossible here.

SEE ALSO

clri, icheck, dcheck, boot procedures (VIII)

BUGS

NAME

cron − clock daemon

SYNOPSIS

/etc/cron

DESCRIPTION

Cron executes commands at specified dates and times according to the instructions in the file /usr/lib/crontab. Since *cron* never exits, it should only be executed once. This is best done by running *cron* from the initialization process through the file /etc/rc; see *init* (VIII).

Crontab consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns to specify the minute (0-59), hour (0-23), day of the month (1-31), month of the year (1-12), and day of the week (1-7 with 1=monday). Each of these patterns may contain a number in the range above; two numbers separated by a minus meaning a range inclusive; a list of numbers separated by commas meaning any of the numbers; or an asterisk meaning all legal values. The sixth field is a string that is executed by the Shell at the specified times. A percent character in this field is translated to a new-line character. Only the first line (up to a % or end of line) of the command field is executed by the Shell. The other lines are made available to the command as standard input.

Crontab is examined by *cron* every hour. Thus it could take up to an hour for entries to become effective. If it receives a hangup signal, however, the table is examined immediately; so ‘kill −1 ...’ can be used.

FILES

/usr/lib/crontab

SEE ALSO

init(VIII), sh(I), kill (I)

DIAGNOSTICS

None − illegal lines in crontab are ignored.

BUGS

A more efficient algorithm could be used. The overhead in running *cron* is about one percent of the machine, exclusive of any commands executed.

NAME

dcheck – file system directory consistency check

SYNOPSIS

dcheck [–**i** numbers] [filesystem]

DESCRIPTION

Dcheck reads the directories in a file system and compares the link-count in each i-node with the number of directory entries by which it is referenced. If the file system is not specified, a set of default file systems is checked.

The –**i** flag is followed by a list of i-numbers; when one of those i-numbers turns up in a directory, the number, the i-number of the directory, and the name of the entry are reported.

The program is fastest if the raw version of the special file is used, since the i-list is read in large chunks.

FILES

Currently, /dev/rrk2 and /dev/rrp0 are the default file systems.

DIAGNOSTICS

When a file turns up for which the link-count and the number of directory entries disagree, the relevant facts are reported. Allocated files which have 0 link-count and no entries are also listed. The only dangerous situation occurs when there are more entries than links; if entries are removed, so the link-count drops to 0, the remaining entries point to thin air. They should be removed. When there are more links than entries, or there is an allocated file with neither links nor entries, some disk space may be lost but the situation will not degenerate.

SEE ALSO

icheck (VIII), fs (V), clri (VIII), ncheck (VIII)

BUGS

Since *dcheck* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.

NAME

df – disk free

SYNOPSIS

df [filesystem]

DESCRIPTION

Df prints out the number of free blocks available on a file system. If the file system is unspecified, the free space on all of the normally mounted file systems is printed.

FILES

/dev/rf?, /dev/rk?, /dev/rp?

SEE ALSO

icheck (VIII)

BUGS

NAME

dpd – data phone daemon

SYNOPSIS

/etc/dpd

DESCRIPTION

Dpd is the 201 data phone daemon. It is designed to submit jobs to the Honeywell 6070 computer via the GRTS interface.

Dpd uses the directory */usr/dpd.* The file *lock* in that directory is used to prevent two daemons from becoming active. After the daemon has successfully set the lock, it forks and the main path exits, thus spawning the daemon. The directory is scanned for files beginning with **df.** Each such file is submitted as a job. Each line of a job file must begin with a key character to specify what to do with the remainder of the line.

- **S** directs *dpd* to generate a unique snumb card. This card is generated by incrementing the first word of the file */usr/dpd/snumb* and converting that to three-digit octal concatenated with the station ID.
- **L** specifies that the remainder of the line is to be sent as a literal.
- **B** specifies that the rest of the line is a file name. That file is to be sent as binary cards.
- **F** is the same as **B** except a form feed is prepended to the file.
- **U** specifies that the rest of the line is a file name. After the job has been transmitted, the file is unlinked.
- **M** is followed by a user ID; after the job is sent, the snumb number and the first line of information in the file is mailed to the user to verify the sending of the job.

Any error encountered will cause the daemon to drop the call, wait up to 20 minutes and start over. This means that an improperly constructed *df* file may cause the same job to be submitted every 20 minutes.

While waiting, the daemon checks to see that the *lock* file still exists. If it is gone, the daemon will exit.

FILES

/dev/dn0, /dev/dp0, /usr/dpd/*

SEE ALSO

opr (I)

NAME

dump – incremental file system dump

SYNOPSIS

dump [key [arguments] filesystem]

DESCRIPTION

Dump makes an incremental file system dump on magtape of all files changed after a certain date. The *key* argument specifies the date and other options about the dump. *Key* consists of characters from the set **abcfiu0hds.**

a Normally files larger than 1000 blocks are not incrementally dump; this flag forces them to be dumped.

b The next argument is taken to be the maximum size of the dump tape in blocks (see **s**).

c If the tape overflows, increment the last character of its name and continue on that drive. (Normally it asks you to change tapes.)

f Place the dump on the next argument file instead of the tape.

i the dump date is taken from the entry in the file /etc/dtab corresponding to the last time this file system was dumped with the **-u** option.

u the date just prior to this dump is written on /etc/dtab upon successful completion of this dump. This file contains a date for every file system dumped with this option.

0 the dump date is taken as the epoch (beginning of time). Thus this option causes an entire file system dump to be taken.

h the dump date is some number of hours before the current date. The number of hours is taken from the next argument in *arguments*.

d the dump date is some number of days before the current date. The number of days is taken from the next argument in *arguments*.

s the size of the dump tape is specified in feet. The number of feet is taken from the next argument in *arguments*. It is assumed that there are 9 standard UNIX records per foot. When the specified size is reached, the dump will wait for reels to be changed. The default size is 2200 feet.

If no arguments are given, the *key* is assumed to be **i** and the file system is assumed to be /dev/rp0.

Full dumps should be taken on quiet file systems as follows:

```
dump 0u /dev/rp0
ncheck /dev/rp0
```

The *ncheck* will come in handy in case it is necessary to restore individual files from this dump. Incremental dumps should then be taken when desired by:

```
dump
```

When the incremental dumps get cumbersome, a new complete dump should be taken. In this way, a restore requires loading of the complete dump tape and only the latest incremental tape.

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. If the first block on the new tape is not writable, e.g. because you forgot the write ring, you get a chance to fix it. Generally, however, read or write failures are fatal.

FILES

/dev/mt0 magtape
/dev/rp0 default file system
/etc/dtab

-

SEE ALSO

restor (VIII), ncheck (VIII), dump (V)

BUGS

NAME

getty – set typewriter mode

SYNOPSIS

/etc/getty [char]

DESCRIPTION

Getty is invoked by *init* (VIII) immediately after a typewriter is opened following a dial-up. It reads the user's name and invokes the *login* command (I) with the name as argument. While reading the name *getty* attempts to adapt the system to the speed and type of terminal being used.

Init calls *getty* with an argument specified by the *ttys* file entry for the typewriter line. Arguments other than '0' can be used to make *getty* treat the line specially. Normally, it sets the speed of the interface to 300 baud, specifies that raw mode is to be used (break on every character), that echo is to be suppressed, and either parity allowed. It types the "login:" message, which includes the characters which put the Terminet 300 terminal into full-duplex and return the GSI terminal to non-graphic mode. Then the user's name is read, a character at a time. If a null character is received, it is assumed to be the result of the user pushing the "break" ("interrupt") key. The speed is then changed to 150 baud and the "login:" is typed again, this time including the character sequence which puts a Teletype 37 into full-duplex. If a subsequent null character is received, the speed is changed back to 300 baud.

The user's name is terminated by a new-line or carriage-return character. The latter results in the system being set to treat carriage returns appropriately (see *stty* (II)).

The user's name is scanned to see if it contains any lower-case alphabetic characters; if not, and if the name is nonempty, the system is told to map any future upper-case characters into the corresponding lower-case characters.

Finally, login is called with the user's name as argument.

SEE ALSO

init (VIII), login (I), stty (II), ttys (V)

BUGS

NAME

glob – generate command arguments

SYNOPSIS

/etc/glob command [arguments]

DESCRIPTION

Glob is used to expand arguments to the shell containing "*", "[", or "?". It is passed the argument list containing the metacharacters; *glob* expands the list and calls the indicated command. The actions of *glob* are detailed in the Shell writeup.

SEE

sh (I)

BUGS

NAME

icheck − file system storage consistency check

SYNOPSIS

icheck [−**s**] [−**b** numbers] [filesystem]

DESCRIPTION

Icheck examines a file system, builds a bit map of used blocks, and compares this bit map against the free list maintained on the file system. If the file system is not specified, a set of default file systems is checked. The normal output of *icheck* includes a report of

The number of blocks missing; i.e. not in any file nor in the free list,
The number of special files,
The total number of files,
The number of large and huge files,
The number of directories,
The number of indirect blocks, and the number of double-indirect blocks in huge files,
The number of blocks used in files,
The number of free blocks.

The −**s** flag causes *icheck* to ignore the actual free list and reconstruct a new one by rewriting the super-block of the file system. The file system should be dismounted while this is done; if this is not possible (for example if the root file system has to be salvaged) care should be taken that the system is quiescent and that it is rebooted immediately afterwards so that the old, bad in-core copy of the super-block will not continue to be used. Notice also that the words in the super-block which indicate the size of the free list and of the i-list are believed. If the super-block has been curdled these words will have to be patched. The −**s** flag causes the normal output reports to be suppressed.

Following the **−b** flag is a list of block numbers; whenever any of the named blocks turns up in a file, a diagnostic is produced.

Icheck is faster if the raw version of the special file is used, since it reads the i-list many blocks at a time.

FILES

Currently, /dev/rrk2 and /dev/rrp0 are the default file systems.

SEE ALSO

dcheck (VIII), ncheck (VIII), fs (V), clri (VIII), restor(VIII)

DIAGNOSTICS

For duplicate blocks and bad blocks (which lie outside the file system) *icheck* announces the difficulty, the i-number, and the kind of block involved. If a read error is encountered, the block number of the bad block is printed and *icheck* considers it to contain 0. ''Bad freeblock'' means that a block number outside the available space was encountered in the free list. ''*n* dups in free'' means that *n* blocks were found in the free list which duplicate blocks either in some file or in the earlier part of the free list.

BUGS

Since *icheck* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.
It believes even preposterous super-blocks and consequently can get core images.

NAME

init – process control initialization

SYNOPSIS

/etc/init

DESCRIPTION

Init is invoked inside UNIX as the last step in the boot procedure. Generally its role is to create a process for each typewriter on which a user may log in.

First, *init* checks to see if the console switches contain 173030. (This number is likely to vary between systems.) If so, the console typewriter **/dev/tty8** is opened for reading and writing and the Shell is invoked immediately. This feature is used to bring up a single-user system. When the system is brought up in this way, the *getty* and *login* routines mentioned below and described elsewhere are not used. If the Shell terminates, *init* starts over looking for the console switch setting.

Otherwise, *init* invokes a Shell, with input taken from the file */etc/rc*. This command file performs housekeeping like removing temporary files, mounting file systems, and starting daemons.

Then *init* reads the file */etc/ttys* and forks several times to create a process for each typewriter specified in the file. Each of these processes opens the appropriate typewriter for reading and writing. These channels thus receive file descriptors 0 and 1, the standard input and output. Opening the typewriter will usually involve a delay, since the *open* is not completed until someone is dialed up and carrier established on the channel. Then */etc/getty* is called with argument as specified by the last character of the *ttys* file line. *Getty* reads the user's name and invokes *login* (q.v.) to log in the user and execute the Shell.

Ultimately the Shell will terminate because of an end-of-file either typed explicitly or generated as a result of hanging up. The main path of *init*, which has been waiting for such an event, wakes up and removes the appropriate entry from the file *utmp*, which records current users, and makes an entry in */usr/adm/wtmp*, which maintains a history of logins and logouts. Then the appropriate typewriter is reopened and *getty* is reinvoked.

Init catches the *hangup* signal (signal #1) and interprets it to mean that the switches should be examined as in a reboot: if they indicate a multi-user system, the */etc/ttys* file is read again. The Shell process on each line which used to be active in *ttys* but is no longer there is terminated; a new process is created for each added line; lines unchanged in the file are undisturbed. Thus it is possible to drop or add phone lines without rebooting the system by changing the *ttys* file and sending a *hangup* signal to the *init* process: use ''kill –1 1.''

FILES

/dev/tty?, /etc/utmp, /usr/adm/wtmp, /etc/ttys, /etc/rc

SEE ALSO

login (I), kill (I), sh (I), ttys (V), getty (VIII)

-

NAME

lpd – line printer daemon

SYNOPSIS

/etc/lpd

DESCRIPTION

Lpd is the line printer daemon (spool area handler) invoked by *opr*. It uses the directory */usr/lpd*. The file *lock* in that directory is used to prevent two daemons from becoming active simultaneously. After the daemon has successfully set the lock, it scans the directory for files beginning with "df." Lines in each *df* file specify files to be printed in the same way as is done by the data-phone daemon dpd (VIII).

FILES

/usr/lpd/* spool area
/dev/lp printer

SEE ALSO

dpd (VIII), opr (I)

BUGS

NAME

mkfs – construct a file system

SYNOPSIS

/etc/mkfs special proto

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 boot procedures (VIII)). 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 i-list size in blocks (remember there are 16 i-nodes per block). 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* (I)).

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 i-list size is the file system size divided by 43 plus the size divided by 1000. (This corresponds to an average size of three blocks per file for a 4000 block file system and six blocks per file at 40,000.) 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

file system (V), directory (V), boot procedures (VIII)

BUGS

It is not possible to initialize a file larger than 64K bytes.
The size of the file system is restricted to 64K blocks.
There should be some way to specify links.

NAME

mknod – build special file

SYNOPSIS

/etc/mknod name [**c**] [**b**] major minor

DESCRIPTION

Mknod makes a directory entry and corresponding i-node for a special file. The first argument is the *name* of the entry. The second is **b** if the special file is block-type (disks, tape) or **c** if it is character-type (other devices). The last two arguments are numbers specifying the *major* device type and the *minor* device (e.g. unit, drive, or line number).

The assignment of major device numbers is specific to each system. They have to be dug out of the system source file *conf.c*.

SEE ALSO

mknod (II)

BUGS

NAME

mount – mount file system

SYNOPSIS

/etc/mount special file [–**r**]

DESCRIPTION

Mount announces to the system that a removable file system is present on the device corresponding to special file *special* (which must refer to a disk or possibly DECtape). The *file* must exist already; it becomes the name of the root of the newly mounted file system.

Mount maintains a table of mounted devices; if invoked without an argument it prints the table.

The optional last argument indicates that the file is to be mounted read-only. Physically write-protected and magnetic tape file systems must be mounted in this way or errors will occur when access times are updated, whether or not any explicit write is attempted.

SEE ALSO

mount (II), mtab (V), umount (VIII)

BUGS

Mounting file systems full of garbage will crash the system.

NAME

ncheck − generate names from i-numbers

SYNOPSIS

ncheck [−**i** numbers] [−**a**] [filesystem]

DESCRIPTION

Ncheck with no argument generates a pathname vs. i-number list of all files on a set of default file systems. The −**i** flag reduces the report to only those files whose i-numbers follow. −**a** allows printing of the names '.' and '..', which are ordinarily suppressed. A file system may be specified.

The full report is in no useful order, and probably should be sorted.

SEE ALSO

dcheck (VIII), icheck (VIII), sort (I)

BUGS

NAME

restor – incremental file system restore

SYNOPSIS

restor key [arguments]

DESCRIPTION

Restor is used to read magtapes dumped with the *dump* command. The *key* argument specifies what is to be done. *Key* is a character from the set **trxw.**

- **t** The date that the tape was made and the date that was specified in the *dump* command are printed. A list of all of the i-numbers on the tape is also given.
- **r** The tape is read and loaded into the file system specified in *arguments*. This should not be done lightly (see below).
- **x** Each file on the tape is individually extracted into a file whose name is the file's i-number. If there are *arguments,* they are interpreted as i-numbers and only they are extracted.
- **c** If the tape overflows, increment the last character of its name and continue on that drive. (Normally it asks you to change tapes.)
- **f** Read the dump from the next argument file instead of the tape.
- **i** All read and checksum errors are reported, but will not cause termination.
- **w** In conjunction with the **x** option, before each file is extracted, its i-number is typed out. To extract this file, you must respond with **y.**

The **x** option is used to retrieve individual files. If the i-number of the desired file is not known, it can be discovered by following the file system directory search algorithm. First retrieve the *root* directory whose i-number is 1. List this file with *ls –fi 1*. This will give names and i-numbers of sub-directories. Iterating, any file may be retrieved.

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

/dev/mt0

SEE ALSO

ls (I), dump (VIII), mkfs (VIII), clri (VIII)

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's* approach is to exit if anything is wrong.

NAME

sa – Shell accounting

SYNOPSIS

sa [–**abcjlnrstuv**] [file]

DESCRIPTION

When a user logs in, if the Shell is able to open the file */usr/adm/sha,* then as each command completes the Shell writes at the end of this file the name of the command, the user, system, and real time consumed, and the user ID. *Sa* reports on, cleans up, and generally maintains this and other accounting files. To turn accounting on and off, the accounting file must be created or destroyed externally.

Sa is able to condense the information in */usr/adm/sha* into a summary file */usr/adm/sht* 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 *sha* 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; *sha* 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.

j Instead of total minutes time for each category, give seconds per call.

l Separate system and user time; normally they are combined.

n Sort by number of calls.

r Reverse order of sort.

s Merge accounting file into summary file */usr/adm/sht* 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 day of the year, time, day of the week, 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.

FILES

/usr/adm/sha	accounting
/usr/adm/sht	summary

SEE ALSO

ac (VIII)

BUGS

NAME

su − become privileged user

SYNOPSIS

su

DESCRIPTION

Su allows one to become the super-user, who has all sorts of marvelous (and correspondingly dangerous) powers. In order for *su* to do its magic, the user must supply a password. If the password is correct, *su* will execute the Shell with the UID set to that of the super-user. To restore normal UID privileges, type an end-of-file to the super-user Shell.

The password demanded is that of the entry ‘‘root’’ in the system’s password file.

To remind the super-user of his responsibilities, the Shell substitutes ‘#’ for its usual prompt ‘%’.

SEE ALSO

sh (I)

NAME

sync – update the super block

SYNOPSIS

sync

DESCRIPTION

Sync executes the *sync* system primitive. If the system is to be stopped, *sync* must be called to insure file system integrity. See sync (II) for details.

SEE ALSO

sync (II)

BUGS

NAME

umount – dismount file system

SYNOPSIS

/etc/umount special

DESCRIPTION

Umount announces to the system that the removable file system previously mounted on special file *special* is to be removed.

SEE ALSO

mount (VIII), umount (II), mtab (V)

FILES

/etc/mtab mounted device table

DIAGNOSTICS

It complains if the special file is not mounted or if it is busy. The file system is busy if there is an open file on it or if someone has his current directory there.

BUGS

NAME

update – periodically update the super block

SYNOPSIS

update

DESCRIPTION

Update is a program that executes the *sync* primitive every 30 seconds. This insures that the file system is fairly up to date in case of a crash. This command should not be executed directly, but should be executed out of the initialization shell command file. See sync (II) for details.

SEE ALSO

sync (II), init (VIII)

BUGS

With *update* running, if the CPU is halted just as the *sync* is executed, a file system can be damaged. This is partially due to DEC hardware that writes zeros when NPR requests fail. A fix would be to have *sync* temporarily increment the system time by at least 30 seconds to trigger the execution of *update*. This would give 30 seconds grace to halt the CPU.

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 to all currently logged in users preceded by ‘‘Broadcast Message ...’’. It is used to warn all users, typically prior to shutting down the system.

The sender should be super-user to override any protections the users may have invoked.

FILES

/dev/tty?

SEE ALSO

mesg (I), write (I)

DIAGNOSTICS

‘‘Cannot send to ...’’ when the open on a user’s tty file fails.

BUGS

DOCUMENTS FOR USE WITH THE UNIX TIME-SHARING SYSTEM

Sixth Edition

The enclosed UNIX documentation is supplied
in accordance with the Software Agreement
you have with the Western Electric Company.

CONTENTS

SETTING UP UNIX − Sixth Edition

Enclosed are:

1. 'UNIX Programmer's Manual,' Sixth Edition.
2. Documents with the following titles:

   Setting Up UNIX − Sixth Edition
   The UNIX Time-Sharing System
   C Reference Manual
   Programming in C − A Tutorial
   UNIX Assembler Reference Manual
   A Tutorial Introduction to the ED Text Editor
   UNIX for Beginners
   RATFOR − A Preprocessor for a Rational Fortran
   YACC − Yet Another Compiler-Compiler
   NROFF Users' Manual
   The UNIX I/O System
   A Manual for the Tmg Compiler-writing Language
   On the Security of UNIX
   The M6 Macro Processor
   A System for Typesetting Mathematics
   DC − An Interactive Desk Calculator
   BC − An Arbitrary Precision Desk-Calculator Language
   The Portable C Library (on UNIX)
   UNIX Summary

3. The UNIX software on magtape or disk pack.

If you are set up to do it, it might be a good idea immediately to make a copy of the disk or tape to guard against disaster. The tape contains 12100 512-byte records followed by a single file mark; only the first 4000 512-byte blocks on the disk are significant.

The system as distributed corresponds to three fairly full RK packs. The first contains the binary version of all programs, and the source for the operating system itself; the second contains all remaining source programs; the third contains manuals intended to be printed using the formatting programs roff or nroff. The 'binary' disk is enough to run the system, but you will almost certainly want to modify some source programs.

Making a Disk From Tape

If your system is on magtape, perform the following bootstrap procedure to obtain a disk with the binaries.

1. Mount magtape on drive 0 at load point.
2. Mount formatted disk pack on drive 0.
3. Key in and execute at 100000

TU10	TU16
012700	(to be added)
172526	
010040	
012740	
060003	
000777	

The tape should move and the CPU loop. (The TU10 code is *not* the DEC bulk ROM for tape; it reads block 0, not block 1.)

4. Halt and restart the CPU at 0. The tape should rewind. The console should type '='.

5. Copy the magtape to disk by the following. This assumes TU10 and RK05; see 6 below for other devices. The machine's printouts are shown in italic (the '=' signs should be considered italic). Terminate each line you type by carriage return or line-feed.

```
= tmrk
disk offset
0
tape offset
100          (See 6 below)
count
1            (The tape should move)
= tmrk
disk offset
1
tape offset
101          (See 7 below)
count
3999         (The tape moves lots more)
=
```

To explain: the *tmrk* program copies tape to disk with the given offsets and counts. Its first use copies a bootstrap program to disk block 0; the second use copies the file system itself onto the disk. You may get back to '=' level by starting at 137000.

6. If you have TU16 tape say 'htrk' instead of 'tmrk' in the above example. If you have an RP03 disk, say 'tmrp' or 'htrp', and use a 99 instead of 100 tape offset. If you have an RP04 disk, use 'tmhp' or 'hthp' instead or 'tmrk', and use a 98 instead of 100 tape offset. The different offsets load bootstrap programs appropriate to the disk they will live on.

7. This procedure generates the 'binary' disk; the 'source' disk may be generated on another RK pack by using a tape offset of 4101 instead of 101. The 'document' disk is at offset 8101 instead of 101. Unless you have only a single RK drive, it is probably wise to wait on generating these disks. Better tools are available using UNIX itself.

Booting UNIX

Once the UNIX 'binary' disk is obtained, the system is booted by keying in and executing one of the following programs at 100000. These programs correspond to the DEC bulk ROMs for disks, since they read in and execute block 0 at location 0.

RK05	RP03	RP04
012700	012700	(to be added)
177414	176726	
005040	005040	
005040	005040	
010040	005040	
012740	010040	
000005	012740	
105710	000005	
002376	105710	
005007	002376	
	005007	

Now follow the indicated dialog, where '@' and '#' are prompts:

```
@ rkunix        (or 'rpunix' or 'hpunix')
mem = xxx
login: root
#
```

The *mem* message gives the memory available to user programs in .1K units. Most of the UNIX software will run with 120 (for 12K words), but some things require much more.

UNIX is now running, and the 'UNIX Programmer's manual' applies; references below of the form X-Y mean the subsection named X in section Y of the manual. The '#' is the prompt from the UNIX Shell, and indicates you are logged in as the super-user. The only valid user names are 'root' and 'bin'. The root is the super-user and bin is the owner of nearly every file in the file system.

Before UNIX is turned up completely, a few configuration dependent exercises must be performed. At this point, it would be wise to read all of the manuals and to augment this reading with hand to hand combat. It might be instructive to examine the Shell run files mentioned below.

Reconfiguration

The UNIX system running is configured to run on an 11/40 with the given disk, TU10 magtape and TU56 DECtape. This is almost certainly not the correct configuration. Print (cat-I) the file /usr/sys/run. This file is a set of Shell commands that will completely recompile the system source, install it in the correct libraries and build the three configurations for rk, rp and hp.

Using the Shell file as a guide, compile (cc-I) and rename (mv-I) the configuration program 'mkconf'. Run the configuration program and type into it a list of the controllers on your system. Choose from:

pc (PC11)
lp (LP11)
rf (RS11)
hs (RS03/RS04)
tc (TU56)
rk (RK03/RK05)
tm (TU10)
rp (RP03)
hp (RP04)
ht (TU16)
dc* (DC11)
kl* (KL11/DL11-ABC)
dl* (DL11-E)
dp (DP11)
dn (DN11)
dh (DH11)
dhdm (DM11-BB)

The devices marked with * should be preceded by a number specifying how many. (The console typewriter is automatically included; don't count it in the kl specification.) Mkconf will generate the two files l.s (trap vectors) and c.c (configuration table). Take a careful look at l.s to make sure that all the devices that you have are assembled in the correct interrupt vectors. If your configuration is non-standard, you will have to modify l.s to fit your configuration.

In the run Shell file, the 11/45 code is commented out. If you have an 11/45 you must also edit (ed-I) the file /usr/sys/conf/m45.s to set the assembly flag fpp to reflect if you have the FP11-B floating point unit. The main difference between an 11/40 and an 11/45 (or 11/70) system is that in the former instruction restart after a segmentation violation caused by overflowing a user stack must be handled by software, while in the latter machines there is hardware help. As mentioned above, the 11/45 and 11/70 systems include conditionally-enabled code to save the status of the floating point unit when switching users. The source for such things is in one of the two files m40.s and m45.s.

Another difference is that in 11/45 and 11/70 systems the instruction and data spaces are separated inside UNIX itself. Since the layout of addresses in the system is somewhat peculiar, and not directly supported by the link-editor *ld,* the *sysfix* program has to be run before the loaded output file can be booted.

There are certain magic numbers and configuration parameters imbedded in various device drivers that you may want to change. The device addresses of each device are defined in each driver. In case you have any non-standard device addresses, just change the address and recompile. (The device drivers are in the directory /usr/sys/dmr.)

The DC11 driver is set to run 14 lines. This can be changed in dc.c.

The DH11 driver will only handle a single DH with a full complement of 16 lines. If you have less, you may want to edit dh.c.

The DN11 driver will handle 3 DN's. Edit dn.c.

The DP11 driver can only handle a single DP. This cannot be easily changed.

The KL/DL driver is set up to run a single DL11-A, -B, or -C (the console) and no DL11-E's. To change this, edit kl.c to have NKL11 reflect the total number of DL11-ABC's and NDL11 to reflect the number of DL11-E's. So far as the driver is concerned, the difference between the devices is their addresses.

The line printer driver is set up to print the 96 character set on 80 column paper (LP11-H) with indenting. Edit lp.c.

All of the disk and tape drivers (rf.c, rk.c, rp.c, tm.c, tc.c, hs.c, hp.c, ht.c) are set up to run 8 drives and should not need to be changed. The big disk drivers (rp.c and hp.c) have partition tables in them which you may want to experiment with.

After all the corrections have been made, use /usr/sys/run as a guide to recompile the changed drivers, install them in /usr/sys/lib2 and to assemble the trap vectors (l.s), configuration table (c.c) and machine language assist (m40.s or m45.s). After all this, link edit the objects (ld-I) and if you have an 11/45, sysfix the result. The final object file (a.out) should be renamed /unix and booted. See Boot Procedures-VIII for a discussion of booting. (Note: remember, before booting, always perform a sync-VIII to force delayed output to the disk.)

Special Files

Next you must put in all of the special files in the directory /dev using mknod-VIII. Print the configuration file c.c created above. This is the major device switch of each device class (block and character). There is one line for each device configured in your system and a null line for place holding for those devices not configured. The block special devices are put in first by executing the following generic command for each disk or tape drive. (Note that some of these files already exist in the directory /dev. Examine each file with ls-I with −l flag to see if the file should be removed.)

/etc/mknod /dev/NAME b MAJOR MINOR

The NAME is selected from the following list:

c.c	NAME	device
rf	rf0	RS fixed head disk
tc	tap0	TU56 DECtape
rk	rk0	RK03 RK05 moving head disk
tm	mt0	TU10 TU16 magtape
rp	rp0	RP moving head disk
hs	hs0	RS03 RS04 fixed head disk
hp	hp0	RP04 moving head disk

The major device number is selected by counting the line number (from zero) of the device's entry in the block configuration table. Thus the first entry in the table bdevsw would be major device zero.

The minor device is the drive number, unit number or partition as described under each device in section IV. The last digit of the name (all given as 0 in the table above) should reflect the minor device number. For tapes where the unit is dial selectable, a special file may be made for each possible selection.

The same goes for the character devices. Here the names are arbitrary except that devices meant to be used for teletype access should be named /dev/ttyX, where X is any character. The files tty8 (console), mem, kmem, null are already correctly configured.

The disk and magtape drivers provide a 'raw' interface to the device which provides direct transmission between the user's core and the device and allows reading or writing large records. The raw device counts as a character device, and should have the name of the corresponding standard block special file with 'r' prepended. Thus the raw magtape files would be called /dev/rmtX.

When all the special files have been created, care should be taken to change the access modes (chmod-I) on these files to appropriate values.

The Source Disk

You should now extract the source disk. This can be done as described above or the UNIX command dd-I may be used. The disk image begins at block 4100 on the tape, so the command

dd if=/dev/mt0 of=/dev/rk1 count=4000 skip=4100

might be used to extract the disk to RK drive 1.

This disk should be mounted (mount-VIII) on /usr/source; it contains directories of source code. In each directory is a Shell file run that will recompile all the source in the directory. These run files should be consulted whenever you need to recompile.

Floating Point

UNIX only supports the 11/45 FP11-B floating point unit. For machines without this hardware, there is a user subroutine available that will catch illegal instruction traps and interpret floating point operations. (See fptrap-III.) The system as delivered has this code included in all commands that have floating point. This code is never used if the FP hardware is available and therefore does not need to be changed. The penalty is a little bit of disk space and loading time for the few floating commands.

The C compiler in /usr/source/c probably should be changed if floating point is available. The fpp flag in c0t.s should be set and C should be recompiled and reloaded and installed. This allows floating point C programs to be compiled without the −f flag and prevents the floating point interpreter from getting into new floating programs. (See /usr/source/c/run.)

Time Conversion

If your machine is not in the Eastern time zone, you must edit (ed-I) the subroutine /usr/source/s4/ctime.c to reflect your local time. The variable 'timezone' should be changed to reflect the time difference between local time and GMT. For EST, this is 5*60*60; for PST it would be 8*60*60. This routine also contains the names of the standard and Daylight Savings time zone; so 'EST' and 'EDT' might be changed to 'PST' and 'PDT' respectively. Notice that these two names are in upper case and escapes may be needed (tty-IV). Finally, there is a 'daylight' flag; when it is 1 it causes the time to shift to Daylight Savings automatically between the last Sundays in April and October (or other algorithms in 1974 and 1975). Normally this will not have to be reset. After ctime.c has been edited it should be compiled and installed in its library. (See /usr/source/s4/run.) Then you should (at your leisure) recompile and reinstall all programs performing time conversion. These include: (in s1) date, dump, ls, cron, (in s2) mail, pr, restor, who, sa and tp.

Disk Layout

If there are to be more file systems mounted than just the root, use mkfs-VIII to create the new file system and put its mounting in the file /etc/rc (see init-VIII and mount-VIII). (You might look at /etc/rc anyway to see what has been provided for you.)

There are two considerations in deciding how to adjust the arrangement of things on your disks: the most important is making sure there is adequate space for what is required; secondarily, throughput should be maximized. The RK disk (or its image) as distributed has 4000 blocks for file storage, and the remainder of the disk (872 blocks) is set aside for swap space. In our own system, which allows 14 simultaneous users, this amount of swap space is not quite enough, so we use 1872 blocks for this purpose; it is large enough so running out of swap space never occurs.

Many common system programs (C, the editor, the assembler etc.) create intermediate files in the /tmp directory, so the file system where this is stored also should be made large enough to accommodate most high-water marks. In an idle state, we have about 900 free blocks on the file system where /tmp resides, and hit the bottom every few days or so. (This causes a momentary disruption, but not a crash, as swap-space runout does.) All the programs that create files in /tmp try to take care to delete them, but most are not immune to events like being hung up upon, and can leave dregs. The directory should be examined every so often and the old files deleted.

Exhaustion of user-file space is certain to occur now and then; the only mechanisms for controlling this phenomenon are occasional use of du-I and threatening messages of the day and personal letters.

The efficiency with which UNIX is able to use the CPU is largely dictated by the configuration of disk controllers. For general time-sharing applications, the best strategy is to try to split user files, the root directory (including the /tmp directory) and the swap area among three controllers. In our own system, for example, we have user files on an RP, the root on an RF fixed-head disk, and swap on an RK. This is best for us since the RK has a faster transfer rate than the rather slow RF, and in swapping the transfer rate rather than access time is the dominant influence on throughput.

Once you have decided how to make best use of your hardware, the question is how to initialize it. If you have the equipment, the best way to move a file system is to dump it (dump-VIII) to magtape, use mkfs-VIII to create the new file system, and restore the tape. If you don't have magtape, dump accepts an

argument telling where to put the dump; you might use another disk or DECtape. Sometimes a file system has to be increased in logical size without copying. The super-block of the device has a word giving the highest address which can be allocated. For relatively small increases, this word can be patched using the debugger (db-I) and the free list reconstructed using icheck-VIII. The size should not be increased very greatly by this technique, however, since although the allocatable space will increase the maximum number of files will not (that is, the i-list size can't be changed). Read and understand the description given in file system-VI before playing around in this way.

If you have only an RP disk, see section rp-IV for some suggestions on how to lay out the information on it. The file systems distributed on tape, containing the binary, the source, and the manuals, are each only 4000 blocks long. Perhaps the simplest way to integrate the latter two into a large file system is to extract the tape into the upper part of the RP, dump it, and restore it into an empty, non-overlapping file system structure. If you have to merge a file system into another, existing one, the best bet is to use ncheck-VIII to get a list of names, then edit this list into a sequence of mkdir and cp commands which will serve as input to the Shell. (But notice that owner information is lost.)

New Users

Install new users by editing the password file /etc/passwd (passwd-V). You'll have to make current directories for the new users and change their owners to the newly installed name. Login as each user to make sure the password file is correctly edited. For example:

```
ed /etc/passwd
$a
joe::10:1::/usr/joe:
.
w
q
mkdir /usr/joe
chown joe /usr/joe
login joe
ls −la
login root
```

This will make a new login entry for joe. His default current directory is /usr/joe which has been created. The delivered password file has the user *ken* in it to be used as a prototype.

Multiple Users

If UNIX is to support simultaneous access from more than just the console teletype, the file /etc/ttys (ttys-V) has to be edited. For some historical reason tty8 is the name of the console typewriter. To add new typewriters be sure the device is configured and the special file exists, then set the first character of the appropriate line of /etc/ttys to 1 (or add a new line). Note that init.c will have to be recompiled if there are to be more than 20 typewriters. Also note that if the special file is inaccessible when init tries to create a process for it, the system will thrash trying and retrying to open it.

File System Health

Periodically (say every day or so) and always after a crash, you should check all the file systems for consistency (icheck, dcheck-VIII). It is quite important to execute sync (VIII) before rebooting or taking the machine down. This is done automatically every 30 seconds by the update program (VIII) when a multiple-user system is running, but you should do it anyway to make sure.

Dumping of the file system should be done regularly, since once the system is going it is very easy to become complacent. Just remember that our RP controller has failed three times, each time in such a way that all information on the disk was wiped out without any error status from the controller. Complete and incremental dumps are easily done with the dump command (VIII) but restoration of individual files is painful. Dumping of files by name is best done by tp (I) but the number of files is limited. Finally if there are enough drives entire disks can be copied using cp-I, or preferably with dd-I using the raw special files

and an appropriate block size. Note that there is no stand-alone program with UNIX that will restore any of these formats. Unless some action has been taken to prevent destruction of a running version of UNIX, you can find yourself stranded even though you have backup.

Odds and Ends

The programs dump, icheck, dcheck, ncheck, and df (source in /usr/source/s1 and /usr/source/s2) should be changed to reflect your default mounted file system devices. Print the first few lines of these programs and the changes will be obvious.

If you would like to share any UNIX compatible software with others, please let us know about it. If you find bugs in the software or the documentation, again let us know.

Lastly, there is a UNIX users' group forming. To get on their mailing list, send your name(s) and address to:

Prof. Melvin Ferentz
Physics Dept.
Brooklyn College of CUNY
Brooklyn, N.Y. 11210

Good luck.
Ken Thompson
Dennis Ritchie

-

The UNIX Time-Sharing System

Dennis M. Ritchie
Ken Thompson

Bell Laboratories
Murray Hill, N. J. 07974

ABSTRACT

UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40, 11/45 and 11/70 computers. It offers a number of features seldom found even in larger operating systems, including

1. A hierarchical file system incorporating demountable volumes,
2. Compatible file, device, and inter-process I/O,
3. The ability to initiate asynchronous processes,
4. System command language selectable on a per-user basis,
5. Over 100 subsystems including a dozen languages.

This paper discusses the nature and implementation of the file system and of the user command interface.

1. Introduction

There have been three versions of UNIX. The earliest version (circa 1969-70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. This paper describes only the PDP-11/40, /45 and /70[1] system, since it is more modern and many of the differences between it and older UNIX systems result from redesign of features found to be deficient or lacking.

Since PDP-11 UNIX became operational in February, 1971, about 100 installations have been put into service; they are generally smaller than the system described here. Most of them are engaged in applications such as the preparation and formatting of patent applications and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. Our own installation is used mainly for research in operating systems, languages, computer networks, and other topics in computer science, and also for document preparation.

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This is a revised version of an article appearing in the Communications of the ACM, Volume 17, Number 7 (July 1974) pp. 365-375. That article is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973.

Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: UNIX can run on hardware costing as little as $40,000, and less than two man-years were spent on the main system software. Yet UNIX contains a number of features seldom offered even in much larger systems. Hopefully, however, the users of UNIX will find that the most important characteristics of the system are its simplicity, elegance, and ease of use.

Besides the system proper, the major programs available under UNIX are

assembler,
text editor based on QED[2],
linking loader,
symbolic debugger,
compiler for a language resembling BCPL[3] with types and structures (C),
interpreter for a dialect of BASIC,
phototypesetting and equation setting programs
Fortran compiler,
Snobol interpreter,
top-down compiler-compiler (TMG[4]),
bottom-up compiler-compiler (YACC),
form letter generator,
macro processor (M6[5]),
permuted index program.

There is also a host of maintenance, utility, recreation and novelty programs. All of these programs were written locally. It is worth noting that the system is totally self-supporting. All UNIX software is maintained under UNIX; likewise, this paper and all other UNIX documents were generated and formatted by the UNIX editor and text formatting program.

2. Hardware and software environment

The PDP-11/45 on which our UNIX installation is implemented is a 16-bit word (8-bit byte) computer with 112K bytes of core memory; UNIX occupies 53K bytes. This system, however, includes a very large number of device drivers and enjoys a generous allotment of space for I/O buffers and system tables; a minimal system capable of running the software mentioned above can require as little as 64K bytes of core altogether.

Our PDP-11 has a 1M byte fixed-head disk, used for file system storage and swapping, four moving-head disk drives which each provide 2.5M bytes on removable disk cartridges, and a single moving-head disk drive which uses removable 40M byte disk packs. There are also a high-speed paper tape reader-punch, nine-track magnetic tape, and DECtape (a variety of magnetic tape facility in which individual records may be addressed and rewritten). Besides the console typewriter, there are 30 variable-speed communications interfaces attached to 100-series datasets and a 201 dataset interface used primarily for spooling printout to a communal line printer. There are also several one-of-a-kind devices including a Picturephone® interface, a voice response unit, a voice synthesizer, a phototypesetter, a digital switching network, and a satellite PDP-11/20 which generates vectors, curves, and characters on a Tektronix 611 storage-tube display.

The greater part of UNIX software is written in the above-mentioned C language[6]. Early versions of the operating system were written in assembly language, but during the summer of 1973, it was rewritten in C. The size of the new system is about one third greater than the old. Since the new system is not only much easier to understand and to modify but also includes many functional improvements, including multiprogramming and the ability to share reentrant code among several user programs, we considered this increase in size quite acceptable.

3. The File system

The most important role of UNIX is to provide a file system. From the point of view of the user, there are three kinds of files: ordinary disk files, directories, and special files.

3.1 Ordinary files

A file contains whatever information the user places on it, for example symbolic or binary (object) programs. No particular structuring is expected by the system. Files of text consist simply of a string of characters, with lines demarcated by the new-line character. Binary programs are sequences of words as they will appear in core memory when the program starts executing. A few user programs manipulate files with more structure; for example, the assembler generates, and the loader expects, an object file in a particular format. However, the structure of files is controlled by the programs which use them, not by the system.

3.2 Directories

Directories provide the mapping between the names of files and the files themselves, and thus induce a structure on the file system as a whole. Each user has a directory of his own files; he may also create subdirectories to contain groups of files conveniently treated together. A directory behaves exactly like an ordinary file except that it cannot be written on by unprivileged programs, so that the system controls the contents of directories. However, anyone with appropriate permission may read a directory just like any other file.

The system maintains several directories for its own use. One of these is the *root* directory. All files in the system can be found by tracing a path through a chain of directories until the desired file is reached. The starting point for such searches is often the root. Another system directory contains all the programs provided for general use; that is, all the *commands*. As will be seen, however, it is by no means necessary that a program reside in this directory for it to be executed.

Files are named by sequences of 14 or fewer characters. When the name of a file is specified to the system, it may be in the form of a *path name,* which is a sequence of directory names separated by slashes "/" and ending in a file name. If the sequence begins with a slash, the search begins in the root directory. The name */alpha/beta/gamma* causes the system to search the root for directory *alpha,* then to search *alpha* for *beta,* finally to find *gamma* in *beta*. *Gamma* may be an ordinary file, a directory, or a special file. As a limiting case, the name "/" refers to the root itself.

A path name not starting with "/" causes the system to begin the search in the user's current directory. Thus, the name *alpha/beta* specifies the file named *beta* in subdirectory *alpha* of the current directory. The simplest kind of name, for example *alpha*, refers to a file which itself is found in the current directory. As another limiting case, the null file name refers to the current directory.

The same non-directory file may appear in several directories under possibly different names. This feature is called *linking;* a directory entry for a file is sometimes called a link. UNIX differs from other systems in which linking is permitted in that all links to a file have equal status. That is, a file does not exist within a particular directory; the directory entry for a file consists merely of its name and a pointer to the information actually describing the file. Thus a file exists independently of any directory entry, although in practice a file is made to disappear along with the last link to it.

Each directory always has at least two entries. The name "**.**" in each directory refers to the directory itself. Thus a program may read the current directory under the name "**.**" without knowing its complete path name. The name "**..**" by convention refers to the parent of the directory in which it appears, that is, to the directory in which it was created.

The directory structure is constrained to have the form of a rooted tree. Except for the special entries "**.**" and "**..**", each directory must appear as an entry in exactly one other, which is its parent. The reason for this is to simplify the writing of programs which visit subtrees of the directory structure, and more important, to avoid the separation of portions of the hierarchy. If arbitrary links to directories were permitted, it would be quite difficult to detect when the last connection from the root to a directory was severed.

3.3 Special files

Special files constitute the most unusual feature of the UNIX file system. Each I/O device supported by UNIX is associated with at least one such file. Special files are read and written just like ordinary disk files, but requests to read or write result in activation of the associated device. An entry for each special file resides in directory */dev,* although a link may be made to one of these files just like an ordinary file. Thus, for example, to punch paper tape, one may write on the file */dev/ppt*. Special files exist for each communication line, each disk, each tape drive, and for physical core memory. Of course, the active disks and the core special file are protected from indiscriminate access.

There is a threefold advantage in treating I/O devices this way: file and device I/O are as similar as possible; file and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can be passed a device name; finally, special files are subject to the same protection mechanism as regular files.

3.4 Removable file systems

Although the root of the file system is always stored on the same device, it is not necessary that the entire file system hierarchy reside on this device. There is a *mount* system request which has two arguments: the name of an existing ordinary file, and the name of a special file whose associated storage volume (e. g. disk pack) should have the structure of an independent file system containing its own directory hierarchy. The effect of *mount* is to cause references to the heretofore ordinary file to refer instead to the root directory of the file system on the removable volume. In effect, *mount* replaces a leaf of the hierarchy tree (the ordinary file) by a whole new subtree (the hierarchy stored on the removable volume). After the *mount*, there is virtually no distinction between files on the removable volume and those in the permanent file system. In our installation, for example, the root directory resides on the fixed-head disk, and the large disk drive, which contains user's files, is mounted by the system initialization program; the four smaller disk drives are available to users for mounting their own disk packs. A mountable file system is generated by writing on its corresponding special file. A utility program is available to create an empty file system, or one may simply copy an existing file system.

There is only one exception to the rule of identical treatment of files on different devices: no link may exist between one file system hierarchy and another. This restriction is enforced so as to avoid the elaborate bookkeeping which would otherwise be required to assure removal of the links when the removable volume is finally dismounted. In particular, in the root directories of all file systems, removable or not, the name "**..**" refers to the directory itself instead of to its parent.

3.5 Protection

Although the access control scheme in UNIX is quite simple, it has some unusual features. Each user of the system is assigned a unique user identification number. When a file is created, it is marked with the user ID of its owner. Also given for new files is a set of seven protection bits. Six of these specify independently read, write, and execute permission for the owner of the file and for all other users.

If the seventh bit is on, the system will temporarily change the user identification of the current user to that of the creator of the file whenever the file is executed as a program. This change in user ID is effective only during the execution of the program which calls for it. The set-user-ID feature provides for privileged programs which may use files inaccessible to other users. For example, a program may keep an accounting file which should neither be read nor changed except by the program itself. If the set-user-identification bit is on for the program, it may access the file although this access might be forbidden to other programs invoked by the given program's user. Since the actual user ID of the invoker of any program is always available, set-user-ID programs may take any measures desired to satisfy themselves as to their invoker's credentials. This mechanism is used to allow users to execute the carefully-written commands which call privileged system entries. For example, there is a system entry invokable only by the "super-user" (below) which creates an empty directory. As indicated above, directories are expected to have entries for "**.**" and "**..**". The command which creates a directory is owned by the super-user and has the set-user-ID bit set. After it checks its invoker's authorization to create the specified directory, it creates it and makes the entries for "**.**" and "**..**".

Since anyone may set the set-user-ID bit on one of his own files, this mechanism is generally available without administrative intervention. For example, this protection scheme easily solves the MOO accounting problem posed in

[7].

The system recognizes one particular user ID (that of the ‘‘super-user’’) as exempt from the usual constraints on file access; thus (for example) programs may be written to dump and reload the file system without unwanted interference from the protection system.

3.6 I/O calls

The system calls to do I/O are designed to eliminate the differences between the various devices and styles of access. There is no distinction between ‘‘random’’ and ‘‘sequential’’ I/O, nor is any logical record size imposed by the system. The size of an ordinary file is determined by the highest byte written on it; no predetermination of the size of a file is necessary or possible.

To illustrate the essentials of I/O in UNIX, some of the basic calls are summarized below in an anonymous language which will indicate the required parameters without getting into the complexities of machine language programming. Each call to the system may potentially result in an error return, which for simplicity is not represented in the calling sequence.

To read or write a file assumed to exist already, it must be opened by the following call:

filep = open (name, flag)

Name indicates the name of the file. An arbitrary path name may be given. The *flag* argument indicates whether the file is to be read, written, or ‘‘updated,’’ that is read and written simultaneously.

The returned value *filep* is called a *file descriptor.* It is a small integer used to identify the file in subsequent calls to read, write or otherwise manipulate the file.

To create a new file or completely rewrite an old one, there is a *create* system call which creates the given file if it does not exist, or truncates it to zero length if it does exist. *Create* also opens the new file for writing and, like *open,* returns a file descriptor.

There are no user-visible locks in the file system, nor is there any restriction on the number of users who may have a file open for reading or writing. Although it is possible for the contents of a file to become scrambled when two users write on it simultaneously, in practice difficulties do not arise. We take the view that locks are neither necessary nor sufficient, in our environment, to prevent interference between users of the same file. They are unnecessary because we are not faced with large, single-file data bases maintained by independent processes. They are insufficient because locks in the ordinary sense, whereby one user is prevented from writing on a file which another user is reading, cannot prevent confusion when, for example, both users are editing a file with an editor which makes a copy of the file being edited.

It should be said that the system has sufficient internal interlocks to maintain the logical consistency of the file system when two users engage simultaneously in such inconvenient activities as writing on the same file, creating files in the same directory, or deleting each other’s open files.

Except as indicated below, reading and writing are sequential. This means that if a particular byte in the file was the last byte written (or read), the next I/O call implicitly refers to the first following byte. For each open file there is a pointer, maintained by the system, which indicates the next byte to be read or written. If *n* bytes are read or written, the pointer advances by *n* bytes.

Once a file is open, the following calls may be used.

n = read (filep, buffer, count)

n = write (filep, buffer, count)

Up to *count* bytes are transmitted between the file specified by *filep* and the byte array specified by *buffer.* The returned value *n* is the number of bytes actually transmitted. In the *write* case, *n* is the same as *count* except under exceptional conditions like I/O errors or end of physical medium on special files; in a *read,* however, *n* may without error be less than *count.* If the read pointer is so near the end of the file that reading *count* characters would cause reading beyond the end, only sufficient bytes are transmitted to reach the end of the file; also, typewriter-like devices

never return more than one line of input. When a *read* call returns with *n* equal to zero, it indicates the end of the file. For disk files this occurs when the read pointer becomes equal to the current size of the file. It is possible to generate an end-of-file from a typewriter by use of an escape sequence which depends on the device used.

Bytes written on a file affect only those implied by the position of the write pointer and the count; no other part of the file is changed. If the last byte lies beyond the end of the file, the file is grown as needed.

To do random (direct access) I/O it is only necessary to move the read or write pointer to the appropriate location in the file.

location = seek (filep, offset, base)

The pointer associated with *filep* is moved to a position *offset* bytes from the beginning of the file, from the current position of the pointer, or from the end of the file, depending on *base. Offset* may be negative. For some devices (e.g. paper tape and typewriters) seek calls are ignored. The actual offset from the beginning of the file to which the pointer was moved is returned in *location.*

3.6.1 Other I/O calls

There are several additional system entries having to do with I/O and with the file system which will not be discussed. For example: close a file, get the status of a file, change the protection mode or the owner of a file, create a directory, make a link to an existing file, delete a file.

4. Implementation of the file system

As mentioned in §3.2 above, a directory entry contains only a name for the associated file and a pointer to the file itself. This pointer is an integer called the *i-number* (for index number) of the file. When the file is accessed, its i-number is used as an index into a system table (the *i-list*) stored in a known part of the device on which the directory resides. The entry thereby found (the file's *i-node*) contains the description of the file:

1. its owner;
2. its protection bits;
3. the physical disk or tape addresses for the file contents;
4. its size;
5. time of last modification;
6. the number of links to the file; that is, the number of times it appears in a directory;
7. a bit indicating whether the file is a directory;
8. a bit indicating whether the file is a special file;
9. a bit indicating whether the file is "large" or "small."

The purpose of an *open* or *create* system call is to turn the path name given by the user into an i-number by searching the explicitly or implicitly named directories. Once a file is open, its device, i-number, and read/write pointer are stored in a system table indexed by the file descriptor returned by the *open* or *create.* Thus the file descriptor supplied during a subsequent call to read or write the file may be easily related to the information necessary to access the file.

When a new file is created, an i-node is allocated for it and a directory entry is made which contains the name of the file and the i-node number. Making a link to an existing file involves creating a directory entry with the new name, copying the i-number from the original file entry, and incrementing the link-count field of the i-node. Removing (deleting) a file is done by decrementing the link-count of the i-node specified by its directory entry and erasing the directory entry. If the link-count drops to 0, any disk blocks in the file are freed and the i-node is deallocated.

The space on all fixed or removable disks which contain a file system is divided into a number of 512-byte blocks logically addressed from 0 up to a limit which depends on the device. There is space in the i-node of each file for eight device addresses. A *small* (non-special) file fits into eight or fewer blocks; in this case the addresses of the blocks themselves are stored. For *large* (non-special) files, seven of the eight device addresses may point to indirect blocks each containing 256 addresses for the data blocks of the file. If required, the eighth word is the address of a

double-indirect block containing 256 more addresses of indirect blocks. Thus files may conceptually grow to (7+256)·256·512 bytes; actually they are restricted to 16,777,216 (2^{24}) bytes. Once opened, a small file (size 1 to 8 blocks) can be accessed directly. A large file (size 9 to 32768 blocks) requires one additional access to read below logical block 1792 (7·256) and two additional references above 1792.

The foregoing discussion applies to ordinary files. When an I/O request is made to a file whose i-node indicates that it is special, the last seven device address words are immaterial, and the first is interpreted as a pair of bytes which constitute an internal *device name.* These bytes specify respectively a device type and subdevice number. The device type indicates which system routine will deal with I/O on that device; the subdevice number selects, for example, a disk drive attached to a particular controller or one of several similar typewriter interfaces.

In this environment, the implementation of the *mount* system call (§3.4) is quite straightforward. *Mount* maintains a system table whose argument is the i-number and device name of the ordinary file specified during the *mount,* and whose corresponding value is the device name of the indicated special file. This table is searched for each (i-number, device)-pair which turns up while a path name is being scanned during an *open* or *create;* if a match is found, the i-number is replaced by 1 (which is the i-number of the root directory on all file systems), and the device name is replaced by the table value.

To the user, both reading and writing of files appear to be synchronous and unbuffered. That is, immediately after return from a *read* call the data are available, and conversely after a *write* the user's workspace may be reused. In fact the system maintains a rather complicated buffering mechanism which reduces greatly the number of I/O operations required to access a file. Suppose a *write* call is made specifying transmission of a single byte. UNIX will search its buffers to see whether the affected disk block currently resides in core memory; if not, it will be read in from the device. Then the affected byte is replaced in the buffer and an entry is made in a list of blocks to be written. The return from the *write* call may then take place, although the actual I/O may not be completed until a later time. Conversely, if a single byte is read, the system determines whether the secondary storage block in which the byte is located is already in one of the system's buffers; if so, the byte can be returned immediately. If not, the block is read into a buffer and the byte picked out.

The system recognizes when a program has made accesses to sequential blocks of a file, and asynchronously pre-reads the next block. This significantly reduces the running time of most programs while adding little to system overhead.

A program which reads or writes files in units of 512 bytes has an advantage over a program which reads or writes a single byte at a time, but the gain is not immense; it comes mainly from the avoidance of system overhead. A program which is used rarely or which does no great volume of I/O may quite reasonably read and write in units as small as it wishes.

The notion of the i-list is an unusual feature of UNIX. In practice, this method of organizing the file system has proved quite reliable and easy to deal with. To the system itself, one of its strengths is the fact that each file has a short, unambiguous name which is related in a simple way to the protection, addressing, and other information needed to access the file. It also permits a quite simple and rapid algorithm for checking the consistency of a file system, for example verification that the portions of each device containing useful information and those free to be allocated are disjoint and together exhaust the space on the device. This algorithm is independent of the directory hierarchy, since it need only scan the linearly-organized i-list. At the same time the notion of the i-list induces certain peculiarities not found in other file system organizations. For example, there is the question of who is to be charged for the space a file occupies, since all directory entries for a file have equal status. Charging the owner of a file is unfair in general, since one user may create a file, another may link to it, and the first user may delete the file. The first user is still the owner of the file, but it should be charged to the second user. The simplest reasonably fair algorithm seems to be to spread the charges equally among users who have links to a file. The current version of UNIX avoids the issue by not charging any fees at all.

4.1 Efficiency of the file system

To provide an indication of the overall efficiency of UNIX and of the file system in particular, timings were made of the assembly of a 8848-line program. The assembly was run alone on the machine; the total clock time was 32 seconds, for a rate of 276 lines per second. The time was divided as follows: 66% assembler execution time, 21% system overhead, 13% disk wait time. We will not attempt any interpretation of these figures nor any comparison with other systems, but merely note that we are generally satisfied with the overall performance of the system.

5. Processes and images

An *image* is a computer execution environment. It includes a core image, general register values, status of open files, current directory and the like. An image is the current state of a pseudo-computer.

A *process* is the execution of an image. While the processor is executing on behalf of a process, the image must reside in core; during the execution of other processes it remains in core unless the appearance of an active, higher-priority process forces it to be swapped out to the fixed-head disk.

The user-core part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual address space begins a non-shared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the hardware's stack pointer fluctuates.

5.1 Processes

Except while UNIX is bootstrapping itself into operation, a new process can come into existence only by use of the *fork* system call:

processid = fork (label)

When *fork* is executed by a process, it splits into two independently executing processes. The two processes have independent copies of the original core image, and share any open files. The new processes differ only in that one is considered the parent process: in the parent, control returns directly from the *fork*, while in the child, control is passed to location *label.* The *processid* returned by the *fork* call is the identification of the other process.

Because the return points in the parent and child process are not the same, each image existing after a *fork* may determine whether it is the parent or child process.

5.2 Pipes

Processes may communicate with related processes using the same system *read* and *write* calls that are used for file system I/O. The call

filep = pipe ( )

returns a file descriptor *filep* and creates an inter-process channel called a *pipe*. This channel, like other open files, is passed from parent to child process in the image by the *fork* call. A *read* using a pipe file descriptor waits until another process writes using the file descriptor for the same pipe. At this point, data are passed between the images of the two processes. Neither process need know that a pipe, rather than an ordinary file, is involved.

Although inter-process communication via pipes is a quite valuable tool (see §6.2), it is not a completely general mechanism, since the pipe must be set up by a common ancestor of the processes involved.

5.3 Execution of programs

Another major system primitive is invoked by

execute (file, arg_1, arg_2, . . . , arg_n)

which requests the system to read in and execute the program named by *file*, passing it string arguments arg_1, arg_2,

. . . , arg_n. All the code and data in the process using *execute* is replaced from the *file*, but open files, current directory, and inter-process relationships are unaltered. Only if the call fails, for example because *file* could not be found or because its execute-permission bit was not set, does a return take place from the *execute* primitive; it resembles a "jump" machine instruction rather than a subroutine call.

5.4 Process synchronization

Another process control system call

processid = wait ( )

causes its caller to suspend execution until one of its children has completed execution. Then *wait* returns the *processid* of the terminated process. An error return is taken if the calling process has no descendants. Certain status from the child process is also available.

5.5 Termination

Lastly,

exit (status)

terminates a process, destroys its image, closes its open files, and generally obliterates it. When the parent is notified through the *wait* primitive, the indicated *status* is available to the parent. Processes may also terminate as a result of various illegal actions or user-generated signals (§7 below).

6. The Shell

For most users, communication with UNIX is carried on with the aid of a program called the Shell. The Shell is a command line interpreter: it reads lines typed by the user and interprets them as requests to execute other programs. In simplest form, a command line consists of the command name followed by arguments to the command, all separated by spaces:

command arg_1 arg_2 . . . arg_n

The Shell splits up the command name and the arguments into separate strings. Then a file with name *command* is sought; *command* may be a path name including the "/" character to specify any file in the system. If *command* is found, it is brought into core and executed. The arguments collected by the Shell are accessible to the command. When the command is finished, the Shell resumes its own execution, and indicates its readiness to accept another command by typing a prompt character.

If file *command* cannot be found, the Shell prefixes the string */bin/* to *command* and attempts again to find the file. Directory */bin* contains all the commands intended to be generally used.

6.1 Standard I/O

The discussion of I/O in §3 above seems to imply that every file used by a program must be opened or created by the program in order to get a file descriptor for the file. Programs executed by the Shell, however, start off with two open files which have file descriptors 0 and 1. As such a program begins execution, file 1 is open for writing, and is best understood as the standard output file. Except under circumstances indicated below, this file is the user's typewriter. Thus programs which wish to write informative or diagnostic information ordinarily use file descriptor 1. Conversely, file 0 starts off open for reading, and programs which wish to read messages typed by the user usually read this file.

The Shell is able to change the standard assignments of these file descriptors from the user's typewriter printer and keyboard. If one of the arguments to a command is prefixed by ">", file descriptor 1 will, for the duration of the command, refer to the file named after the ">". For example,

ls

ordinarily lists, on the typewriter, the names of the files in the current directory. The command

```
ls >there
```

creates a file called *there* and places the listing there. Thus the argument ‘‘>there’’ means, ‘‘place output on *there*.’’ On the other hand,

```
ed
```

ordinarily enters the editor, which takes requests from the user via his typewriter. The command

```
ed <script
```

interprets *script* as a file of editor commands; thus ‘‘<script’’ means, ‘‘take input from *script*.’’

Although the file name following ‘‘<’’ or ‘‘>’’ appears to be an argument to the command, in fact it is interpreted completely by the Shell and is not passed to the command at all. Thus no special coding to handle I/O redirection is needed within each command; the command need merely use the standard file descriptors 0 and 1 where appropriate.

6.2 Filters

An extension of the standard I/O notion is used to direct output from one command to the input of another. A sequence of commands separated by vertical bars causes the Shell to execute all the commands simultaneously and to arrange that the standard output of each command be delivered to the standard input of the next command in the sequence. Thus in the command line

```
ls | pr −2 | opr
```

ls lists the names of the files in the current directory; its output is passed to *pr*, which paginates its input with dated headings. The argument ‘‘−2’’ means double column. Likewise the output from *pr* is input to *opr*. This command spools its input onto a file for off-line printing.

This procedure could have been carried out more clumsily by

```
ls >temp1
pr −2 <temp1 >temp2
opr <temp2
```

followed by removal of the temporary files. In the absence of the ability to redirect output and input, a still clumsier method would have been to require the *ls* command to accept user requests to paginate its output, to print in multi-column format, and to arrange that its output be delivered off-line. Actually it would be surprising, and in fact unwise for efficiency reasons, to expect authors of commands such as *ls* to provide such a wide variety of output options.

A program such as *pr* which copies its standard input to its standard output (with processing) is called a *filter*. Some filters which we have found useful perform character transliteration, sorting of the input, and encryption and decryption.

6.3 Command Separators; Multitasking

Another feature provided by the Shell is relatively straightforward. Commands need not be on different lines; instead they may be separated by semicolons.

```
ls; ed
```

will first list the contents of the current directory, then enter the editor.

A related feature is more interesting. If a command is followed by ‘‘&’’, the Shell will not wait for the command to finish before prompting again; instead, it is ready immediately to accept a new command. For example,

```
as source >output &
```

causes *source* to be assembled, with diagnostic output going to *output;* no matter how long the assembly takes, the Shell returns immediately. When the Shell does not wait for the completion of a command, the identification of the

process running that command is printed. This identification may be used to wait for the completion of the command or to terminate it. The "&" may be used several times in a line:

```
as source >output & ls >files &
```

does both the assembly and the listing in the background. In the examples above using "&", an output file other than the typewriter was provided; if this had not been done, the outputs of the various commands would have been intermingled.

The Shell also allows parentheses in the above operations. For example

```
( date; ls ) >x &
```

prints the current date and time followed by a list of the current directory onto the file *x*. The Shell also returns immediately for another request.

6.4 The Shell as a Command; Command Files

The Shell is itself a command, and may be called recursively. Suppose file *tryout* contains the lines

```
as source
mv a.out testprog
testprog
```

The *mv* command causes the file *a.out* to be renamed *testprog*. *A.out* is the (binary) output of the assembler, ready to be executed. Thus if the three lines above were typed on the console, *source* would be assembled, the resulting program renamed *testprog*, and *testprog* executed. When the lines are in *tryout*, the command

```
sh <tryout
```

would cause the Shell *sh* to execute the commands sequentially.

The Shell has further capabilities, including the ability to substitute parameters and to construct argument lists from a specified subset of the file names in a directory. It is also possible to execute commands conditionally on character string comparisons or on existence of given files and to perform transfers of control within filed command sequences.

6.5 Implementation of the Shell

The outline of the operation of the Shell can now be understood. Most of the time, the Shell is waiting for the user to type a command. When the new-line character ending the line is typed, the Shell's *read* call returns. The Shell analyzes the command line, putting the arguments in a form appropriate for *execute*. Then *fork* is called. The child process, whose code of course is still that of the Shell, attempts to perform an *execute* with the appropriate arguments. If successful, this will bring in and start execution of the program whose name was given. Meanwhile, the other process resulting from the *fork*, which is the parent process, *wait* s for the child process to die. When this happens, the Shell knows the command is finished, so it types its prompt and reads the typewriter to obtain another command.

Given this framework, the implementation of background processes is trivial; whenever a command line contains "&", the Shell merely refrains from waiting for the process which it created to execute the command.

Happily, all of this mechanism meshes very nicely with the notion of standard input and output files. When a process is created by the *fork* primitive, it inherits not only the core image of its parent but also all the files currently open in its parent, including those with file descriptors 0 and 1. The Shell, of course, uses these files to read command lines and to write its prompts and diagnostics, and in the ordinary case its children_the command programs_inherit them automatically. When an argument with "<" or ">" is given however, the offspring process, just before it performs *execute,* makes the standard I/O file descriptor 0 or 1 respectively refer to the named file. This is easy because, by agreement, the smallest unused file descriptor is assigned when a new file is *open* ed (or *create* d); it is only necessary to close file 0 (or 1) and open the named file. Because the process in which the command program runs simply terminates when it is through, the association between a file specified after "<" or ">" and file descriptor 0 or 1 is ended automatically when the process dies. Therefore the Shell need not know the actual names of the

files which are its own standard input and output, since it need never reopen them.

Filters are straightforward extensions of standard I/O redirection with pipes used instead of files.

In ordinary circumstances, the main loop of the Shell never terminates. (The main loop includes that branch of the return from *fork* belonging to the parent process; that is, the branch which does a *wait*, then reads another command line.) The one thing which causes the Shell to terminate is discovering an end-of-file condition on its input file. Thus, when the Shell is executed as a command with a given input file, as in

```
sh <comfile
```

the commands in *comfile* will be executed until the end of *comfile* is reached; then the instance of the Shell invoked by *sh* will terminate. Since this Shell process is the child of another instance of the Shell, the *wait* executed in the latter will return, and another command may be processed.

6.6 Initialization

The instances of the Shell to which users type commands are themselves children of another process. The last step in the initialization of UNIX is the creation of a single process and the invocation (via *execute*) of a program called *init*. The role of *init* is to create one process for each typewriter channel which may be dialed up by a user. The various subinstances of *init* open the appropriate typewriters for input and output. Since when *init* was invoked there were no files open, in each process the typewriter keyboard will receive file descriptor 0 and the printer file descriptor 1. Each process types out a message requesting that the user log in and waits, reading the typewriter, for a reply. At the outset, no one is logged in, so each process simply hangs. Finally someone types his name or other identification. The appropriate instance of *init* wakes up, receives the log-in line, and reads a password file. If the user name is found, and if he is able to supply the correct password, *init* changes to the user's default current directory, sets the process's user ID to that of the person logging in, and performs an *execute* of the Shell. At this point the Shell is ready to receive commands and the logging-in protocol is complete.

Meanwhile, the mainstream path of *init* (the parent of all the subinstances of itself which will later become Shells) does a *wait*. If one of the child processes terminates, either because a Shell found an end of file or because a user typed an incorrect name or password, this path of *init* simply recreates the defunct process, which in turn reopens the appropriate input and output files and types another login message. Thus a user may log out simply by typing the end-of-file sequence in place of a command to the Shell.

6.7 Other programs as Shell

The Shell as described above is designed to allow users full access to the facilities of the system, since it will invoke the execution of any program with appropriate protection mode. Sometimes, however, a different interface to the system is desirable, and this feature is easily arranged.

Recall that after a user has successfully logged in by supplying his name and password, *init* ordinarily invokes the Shell to interpret command lines. The user's entry in the password file may contain the name of a program to be invoked after login instead of the Shell. This program is free to interpret the user's messages in any way it wishes.

For example, the password file entries for users of a secretarial editing system specify that the editor *ed* is to be used instead of the Shell. Thus when editing system users log in, they are inside the editor and can begin work immediately; also, they can be prevented from invoking UNIX programs not intended for their use. In practice, it has proved desirable to allow a temporary escape from the editor to execute the formatting program and other utilities.

Several of the games (e.g., chess, blackjack, 3D tic-tac-toe) available on UNIX illustrate a much more severely restricted environment. For each of these an entry exists in the password file specifying that the appropriate game-playing program is to be invoked instead of the Shell. People who log in as a player of one of the games find themselves limited to the game and unable to investigate the presumably more interesting offerings of UNIX as a whole.

7. Traps

The PDP-11 hardware detects a number of program faults, such as references to non-existent memory, unimplemented instructions, and odd addresses used where an even address is required. Such faults cause the processor to trap to a system routine. When an illegal action is caught, unless other arrangements have been made, the system terminates the process and writes the user's image on file *core* in the current directory. A debugger can be used to determine the state of the program at the time of the fault.

Programs which are looping, which produce unwanted output, or about which the user has second thoughts may be halted by the use of the *interrupt* signal, which is generated by typing the ''delete'' character. Unless special action has been taken, this signal simply causes the program to cease execution without producing a core image file.

There is also a *quit* signal which is used to force a core image to be produced. Thus programs which loop unexpectedly may be halted and the core image examined without prearrangement.

The hardware-generated faults and the interrupt and quit signals can, by request, be either ignored or caught by the process. For example, the Shell ignores quits to prevent a quit from logging the user out. The editor catches interrupts and returns to its command level. This is useful for stopping long printouts without losing work in progress (the editor manipulates a copy of the file it is editing). In systems without floating point hardware, unimplemented instructions are caught and floating point instructions are interpreted.

8. Perspective

Perhaps paradoxically, the success of UNIX is largely due to the fact that it was not designed to meet any predefined objectives. The first version was written when one of us (Thompson), dissatisfied with the available computer facilities, discovered a little-used PDP-7 and set out to create a more hospitable environment. This essentially personal effort was sufficiently successful to gain the interest of the remaining author and others, and later to justify the acquisition of the PDP-11/20, specifically to support a text editing and formatting system. When in turn the 11/20 was outgrown, UNIX had proved useful enough to persuade management to invest in the PDP-11/45. Our goals throughout the effort, when articulated at all, have always concerned themselves with building a comfortable relationship with the machine and with exploring ideas and inventions in operating systems. We have not been faced with the need to satisfy someone else's requirements, and for this freedom we are grateful.

Three considerations which influenced the design of UNIX are visible in retrospect.

First: since we are programmers, we naturally designed the system to make it easy to write, test, and run programs. The most important expression of our desire for programming convenience was that the system was arranged for interactive use, even though the original version only supported one user. We believe that a properly-designed interactive system is much more productive and satisfying to use than a ''batch'' system. Moreover such a system is rather easily adaptable to non-interactive use, while the converse is not true.

Second: there have always been fairly severe size constraints on the system and its software. Given the partially antagonistic desires for reasonable efficiency and expressive power, the size constraint has encouraged not only economy but a certain elegance of design. This may be a thinly disguised version of the ''salvation through suffering'' philosophy, but in our case it worked.

Third: nearly from the start, the system was able to, and did, maintain itself. This fact is more important than it might seem. If designers of a system are forced to use that system they quickly become aware of its functional and superficial deficiencies and are strongly motivated to correct them before it is too late. Since all source programs were always available and easily modified on-line, we were willing to revise and rewrite the system and its software when new ideas were invented, discovered, or suggested by others.

The aspects of UNIX discussed in this paper exhibit clearly at least the first two of these design considerations. The interface to the file system, for example, is extremely convenient from a programming standpoint. The lowest possible interface level is designed to eliminate distinctions between the various devices and files and between direct and sequential access. No large ''access method'' routines are required to insulate the programmer from the system calls; in fact all user programs either call the system directly or use a small library program, only tens of instructions long, which buffers a number of characters and reads or writes them all at once.

Another important aspect of programming convenience is that there are no ‘‘control blocks’’ with a complicated structure partially maintained by and depended on by the file system or other system calls. Generally speaking, the contents of a program's address space are the property of the program, and we have tried to avoid placing restrictions on the data structures within that address space.

Given the requirement that all programs should be usable with any file or device as input or output, it is also desirable from a space-efficiency standpoint to push device-dependent considerations into the operating system itself. The only alternatives seem to be to load routines for dealing with each device with all programs, which is expensive in space, or to depend on some means of dynamically linking to the routine appropriate to each device when it is actually needed, which is expensive either in overhead or in hardware.

Likewise, the process control scheme and command interface have proved both convenient and efficient. Since the Shell operates as an ordinary, swappable user program, it consumes no wired-down space in the system proper, and it may be made as powerful as desired at little cost. In particular, given the framework in which the Shell executes as a process which spawns other processes to perform commands, the notions of I/O redirection, background processes, command files, and user-selectable system interfaces all become essentially trivial to implement.

8.1 Influences

The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system.

The *fork* operation, essentially as we implemented it, was present in the Berkeley time sharing system[8]. On a number of points we were influenced by Multics, which suggested the particular form of the I/O system calls[9] and both the name of the Shell and its general functions. The notion that the Shell should create a process for each command was also suggested to us by the early design of Multics, although in that system it was later dropped for efficiency reasons. A similar scheme is used by TENEX[10].

9. Statistics

The following numbers are presented to suggest the scale of our operation. Those of our users not involved in document preparation tend to use the system for program development, especially language work. There are few important ‘‘applications’’ programs.

Overall, we have

100	user population
14	maximum simultaneous users
380	directories
4800	files
66300	512-byte secondary storage blocks used

There is a ‘‘background’’ process that runs at the lowest possible priority; it is used to soak up any idle CPU time. It has been used to produce a million-digit approximation to the constant $e-2$, and is now solving all rook-and-pawn vs. rook chess endgames. Not counting this background work, we average daily

2400	commands
5.5	CPU hours
100	connect hours
32	different users
100	logins

Acknowledgements. We are grateful to R.H. Canaday, L.L. Cherry, and L.E. McMahon for their contributions to UNIX. We are particularly appreciative of the inventiveness, thoughtful criticism, and constant support of R. Morris, M.D. McIlroy, and J.F. Ossanna.

References

1. Digital Equipment Corporation. *PDP-11/40 Processor Handbook* (1972), *PDP-11/45 Processor Handbook* (1971), and *PDP-11/70 Processor Handbook* (1975).
2. Deutsch, L.P., and Lampson, B.W. An online editor. *Comm. ACM 10,* 12 (Dec. 1967), 793-799, 803.
3. Richards, M. BCPL: A tool for compiler writing and system programming. Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J., pp. 557-566.
4. McClure, R.M. TMG—A syntax directed compiler. Proc. ACM 20th Nat. Conf., ACM, 1965, New York, pp. 262-274.
5. Hall, A.D. The M6 macroprocessor. Computing Science Tech. Rep. #2, Bell Telephone Laboratories, 1969.
6. Ritchie, D.M. C reference manual. Unpublished memorandum, Bell Telephone Laboratories (1973).
7. Aleph-null. Computer Recreations. *Software Practice and Experience 1,* 2 (Apr.-June 1971), 201-204.
8. Deutch, L.P. and Lampson, B.W. SDS 930 time-sharing system preliminary reference manual. Doc. 30.10.10, Project GENIE, Univ. Cal. at Berkeley (Apr. 1965).
9. Feiertag, R.J., and Organick, E.I. The Multics input-output system. Proc. Third Symposium on Operating Systems Principles. Oct. 18-20, 1971, ACM, New York, pp. 35-41.
10. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time sharing system for the PDP-10. *Comm. ACM 15.,* 3 (March 1972) 135-143.

C Reference Manual

Dennis M. Ritchie
Bell Telephone Laboratories
Murray Hill, New Jersey 07974

1. Introduction

C is a computer language based on the earlier language B [1]. The languages and their compilers differ in two major ways: C introduces the notion of types, and defines appropriate extra syntax and semantics; also, C on the PDP-11 is a true compiler, producing machine code where B produced interpretive code.

Most of the software for the UNIX time-sharing system [2] is written in C, as is the operating system itself. C is also available on the HIS 6070 computer at Murray Hill and and on the IBM System/370 at Holmdel [3]. This paper is a manual only for the C language itself as implemented on the PDP-11. However, hints are given occasionally in the text of implementation-dependent features.

The UNIX Programmer's Manual [4] describes the library routines available to C programs under UNIX, and also the procedures for compiling programs under that system. ''The GCOS C Library'' by Lesk and Barres [5] describes routines available under that system as well as compilation procedures. Many of these routines, particularly the ones having to do with I/O, are also provided under UNIX. Finally, ''Programming in C– A Tutorial,'' by B. W. Kernighan [6], is as useful as promised by its title and the author's previous introductions to allegedly impenetrable subjects.

2. Lexical conventions

There are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. In general blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs.

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 `*/`.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore ''_'' counts as alphabetic. Upper and lower case letters are considered different. No more than the first eight characters are significant, and only the first seven for external identifiers.

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
int                     break
char                    continue
float                   if
double                  else
struct                  for
auto                    do
extern                  while
register                switch
static                  case
goto                    default
return                  entry
sizeof
```

The `entry` keyword is not currently implemented by any compiler but is reserved for future use.

2.3 Constants

There are several kinds of constants, as follows:

2.3.1 Integer constants

An integer constant is a sequence of digits. An integer is taken to be octal if it begins with `0`, decimal otherwise. The digits `8` and `9` have octal value 10 and 11 respectively.

2.3.2 Character constants

A character constant is 1 or 2 characters enclosed in single quotes " ´ ". Within a character constant a single quote must be preceded by a back-slash "\". Certain non-graphic characters, and "\" itself, may be escaped according to the following table:

BS	\b
NL	\n
CR	\r
HT	\t
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 a null character.

Character constants behave exactly like integers (not, in particular, like objects of character type). In conformity with the addressing structure of the PDP-11, a character constant of length 1 has the code for the given character in the low-order byte and 0 in the high-order byte; a character constant of length 2 has the code for the first character in the low byte and that for the second character in the high-order byte. Character constants with more than one character are inherently machine-dependent and should be avoided.

2.3.3 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an `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.

2.4 Strings

A string is a sequence of characters surrounded by double quotes " **"** ". A string has the type array-of-characters (see below) and refers to an area of storage initialized with the given characters. 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 character " **"** " must be preceded by a "\" ; in addition, the same escapes as described for character constants may be used.

3. Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in `gothic`. Alternatives are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript "opt," so that

{ expression$_{opt}$ *}*

would indicate an optional expression in braces.

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 function, and are discarded on return; static variables are local to a function, but retain their values independently of invocations of the function; external variables are independent of any function. Register variables are stored in the fast registers of the machine; like automatic variables they are local to each function and disappear on return.

C supports four fundamental types of objects: characters, integers, single-, and double-precision floating-point numbers.

Characters (declared, and hereinafter called, `char`) are chosen from the ASCII set; they occupy the rightmost seven bits of an 8-bit byte. It is also possible to interpret `char`s as signed, 2's complement 8-bit numbers.

Integers (`int`) are represented in 16-bit 2's complement notation.

Single precision floating point (`float`) quantities have magnitude in the range approximately $10^{\pm 38}$ or 0; their precision is 24 bits or about seven decimal digits.

Double-precision floating-point (`double`) quantities have the same range as `float`s and a precision of 56 bits or about 17 decimal digits.

Besides the four fundamental 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 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.

6.1 Characters and integers

A `char` object may be used anywhere an `int` may be. In all cases the `char` is converted to an `int` by propagating its sign through the upper 8 bits of the resultant integer. This is consistent with the two's complement representation used for both characters and integers. (However, the sign-propagation feature disappears in other implementations.)

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 Float and double; integer and character

All `int`s and `char`s may be converted without loss of significance to `float` or `double`. Conversion of `float` or `double` to `int` or `char` takes place with truncation towards 0. Erroneous results can be expected if the magnitude of the result exceeds 32,767 (for `int`) or 127 (for `char`).

6.4 Pointers and integers

Integers and pointers may be added and compared; in such a case the `int` 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.

7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Thus 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 an appendix.

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.

7.1 Primary expressions

Primary expressions involving `.`, `->`, subscripting, and function calls group left to right.

7.1.1 *identifier*

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. However, if the type of the identifier is "array of ...", 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 ...".

7.1.2 *constant*

A decimal, octal, character, or floating constant is a primary expression. Its type is `int` in the first three cases, `double` in the last.

7.1.3 *string*

A string is a primary expression. Its type is originally "array of `char`"; but following the same rule as in §7.1.1 for identifiers, this is modified to "pointer to `char`" and the result is a pointer to the first character in the string.

7.1.4 (*expression*)

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.

7.1.5 *primary-expression* `[` *expression* `]`

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 §§ 7.1.1, 7.2.1, and 7.4.1 on identifiers, *, and + respectively; §14.3 below summarizes the implications.

7.1.6 *primary-expression* `(` $expression\text{-}list_{opt}$ `)`

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` are converted to `int`.

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 possibly affect the values of the actual parameters. On the other hand, it is perfectly possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points.

Recursive calls to any function are permissible.

7.1.7 *primary-lvalue* `.` *member-of-structure*

An lvalue expression followed by a dot followed by the name of a member of a structure is a primary expression. The object referred to by the lvalue is assumed to have the same form as the structure containing the structure member. The result of the expression is an lvalue appropriately offset from the origin of the given lvalue whose type is that of the named structure member. The given lvalue is not required to have any particular type.

Structures are discussed in §8.5.

7.1.8 *primary-expression* `->` *member-of-structure*

The primary-expression is assumed to be a pointer which points to an object of the same form as the structure of which the member-of-structure is a part. The result is an lvalue appropriately offset from the origin of the pointed-to structure whose type is that of the named structure member. The type of the primary-expression need not in fact be pointer; it is sufficient that it be a pointer, character, or integer.

Except for the relaxation of the requirement that E1 be of pointer type, the expression "E1->MOS" is exactly equivalent to "(*E1).MOS".

7.2 Unary operators

Expressions with unary operators group right-to-left.

7.2.1 `*` *expression*

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 "...".

7.2.2 `&` *lvalue-expression*

The result of the unary `&` operator is a pointer to the object referred to by the lvalue-expression. If the type of the lvalue-expression is "...", the type of the result is "pointer to ...".

7.2.3 `-` *expression*

The result is the negative of the expression, and has the same type. The type of the expression must be `char`, `int`, `float`, or `double`.

7.2.4 `!` *expression*

The result of the logical negation operator `!` is 1 if the value of the expression is 0, 0 if the value of the expression is non-zero. The type of the result is `int`. This operator is applicable only to `int`s or `char`s.

7.2.5 `~` *expression*

The ~ operator yields the one's complement of its operand. The type of the expression must be `int` or `char`, and the result is `int`.

7.2.6 ++ *lvalue-expression*

The object referred to by the lvalue expression is incremented. The value is the new value of the lvalue expression and the type is the type of the lvalue. If the expression is `int` or `char`, it is incremented by 1; if it is a pointer to an object, it is incremented by the length of the object. ++ is applicable only to these types. (Not, for example, to `float` or `double`.)

7.2.7 –– *lvalue-expression*

The object referred to by the lvalue expression is decremented analogously to the ++ operator.

7.2.8 *lvalue-expression* ++

The result is the value of the object referred to by the lvalue expression. After the result is noted, the object referred to by the lvalue is incremented in the same manner as for the prefix ++ operator: by 1 for an `int` or `char`, by the length of the pointed-to object for a pointer. The type of the result is the same as the type of the lvalue-expression.

7.2.9 *lvalue-expression* ––

The result of the expression is the value of the object referred to by the the lvalue expression. After the result is noted, the object referred to by the lvalue expression is decremented in a way analogous to the postfix ++ operator.

7.2.10 `sizeof` *expression*

The `sizeof` operator yields the size, in bytes, of its operand. 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.

7.3 Multiplicative operators

The multiplicative operators `*`, `/`, and `%` group left-to-right.

7.3.1 *expression* `*` *expression*

The binary `*` operator indicates multiplication. If both operands are `int` or `char`, the result is `int`; if one is `int` or `char` and one `float` or `double`, the former is converted to `double`, and the result is `double`; if both are `float` or `double`, the result is `double`. No other combinations are allowed.

7.3.2 *expression* / *expression*

The binary / operator indicates division. The same type considerations as for multiplication apply.

7.3.3 *expression* `%` *expression*

The binary `%` operator yields the remainder from the division of the first expression by the second. Both operands must be `int` or `char`, and the result is `int`. In the current implementation, the remainder has the same sign as the dividend.

7.4 Additive operators

The additive operators + and – group left-to-right.

7.4.1 *expression* + *expression*

The result is the sum of the expressions. If both operands are `int` or `char`, the result is `int`. If both are `float` or `double`, the result is `double`. If one is `char` or `int` and one is `float` or `double`, the former is converted to `double` and the result is `double`. If an `int` or `char` is added to a pointer, the former is converted by multiplying it by the length of the object to which the pointer points and the result is a pointer of the same type as the original pointer. Thus if P is a pointer to an object, the expression "P+1" is a pointer to another object of the same type as the first and immediately following it in storage.

No other type combinations are allowed.

7.4.2 *expression* − *expression*

The result is the difference of the operands. If both operands are `int`, `char`, `float`, or `double`, the same type considerations as for + apply. If an `int` or `char` is subtracted from a pointer, the former is converted in the same way as explained under + above.

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.

7.5.1 *expression* << *expression*
7.5.2 *expression* >> *expression*

Both operands must be `int` or `char`, and the result is `int`. The second operand should be non-negative. The value of "E1<<E2" is E1 (interpreted as a bit pattern 16 bits long) left-shifted E2 bits; vacated bits are 0-filled. The value of "E1>>E2" is E1 (interpreted as a two's complement, 16-bit quantity) arithmetically right-shifted E2 bit positions. Vacated bits are filled by a copy of the sign bit of E1. [Note: the use of arithmetic rather than logical shift does not survive transportation between machines.]

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.

7.6.1 *expression* < *expression*
7.6.2 *expression* > *expression*
7.6.3 *expression* <= *expression*
7.6.4 *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. Operand conversion is exactly the same as for the + operator except that pointers of any kind may be compared; the result in this case depends on the relative locations in storage of the pointed-to objects. It does not seem to be very meaningful to compare pointers with integers other than 0.

7.7 Equality operators

7.7.1 *expression* == *expression*
7.7.2 *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).

7.8 *expression* & *expression*

The & operator groups left-to-right. Both operands must be `int` or `char`; the result is an `int` which is the bitwise logical `and` function of the operands.

7.9 *expression* ^ *expression*

The ^ operator groups left-to-right. The operands must be `int` or `char`; the result is an `int` which is the bit-wise exclusive `or` function of its operands.

7.10 *expression* | *expression*

The | operator groups left-to-right. The operands must be `int` or `char`; the result is an `int` which is the bit-wise inclusive `or` of its operands.

7.11 *expression* && *expression*

The && operator 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.

7.12 *expression* || *expression*

The || operator 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.

7.13 *expression* ? *expression* : *expression*

Conditional expressions group left-to-right. 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 the types of the second and third operand are the same, the result has their common type; otherwise the same conversion rules as for + apply. 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.

7.14.1 *lvalue* = *expression*

The value of the expression replaces that of the object referred to by the lvalue. The operands need not have the same type, but both must be `int`, `char`, `float`, `double`, or pointer. If neither operand is a pointer, the assignment takes place as expected, possibly preceded by conversion of the expression on the right.

When both operands are `int` or pointers of any kind, no conversion ever takes place; the value of the expression is simply stored into the object referred to by the lvalue. Thus it is possible to generate pointers which will cause addressing exceptions when used.

7.14.2 *lvalue* =+ *expression*
7.14.3 *lvalue* =− *expression*
7.14.4 *lvalue* =∗ *expression*
7.14.5 *lvalue* =/ *expression*
7.14.6 *lvalue* =% *expression*
7.14.7 *lvalue* =>> *expression*
7.14.8 *lvalue* =<< *expression*
7.14.9 *lvalue* =& *expression*
7.14.10 *lvalue* =^ *expression*
7.14.11 *lvalue* =| *expression*

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. Moreover, expressions like "i =+ p" in which a pointer is added to an integer, are forbidden.

7.15 *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. It should be avoided in situations where comma is given a special meaning, for example in actual arguments to function calls (§7.1.6) and lists of initializers (§10.2).

8. Declarations

Declarations are used within function definitions 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 at most one type-specifier and at most one storage class specifier.

decl-specifiers:
type-specifier
sc-specifier
type-specifier sc-specifier
sc-specifier type-specifier

8.1 Storage class specifiers

The sc-specifiers are:

sc-specifier:
`auto`
`static`
`extern`
`register`

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 (see below) for the given identifiers somewhere outside the function in which they are declared.

There are some severe restrictions on `register` identifiers: there can be at most 3 register identifiers in any function, and the type of a register identifier can only be `int`, `char`, or pointer (not `float`, `double`, structure, function, or array). Also the address-of operator `&` cannot be applied to such identifiers. Except for these restrictions (in return for which one is rewarded with faster, smaller code), register identifiers behave as if they were automatic. In fact implementations of C are free to treat `register` as synonymous with `auto`.

If the sc-specifier is missing from a declaration, it is generally taken to be `auto`.

8.2 Type specifiers

The type-specifiers are

type-specifier:
`int`
`char`
`float`
`double`
`struct` *{ type-decl-list }*
`struct` *identifier { type-decl-list }*
`struct` *identifier*

The `struct` specifier is discussed in §8.5. If the type-specifier is missing from a declaration, it is generally taken to be `int`.

8.3 Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators.

declarator-list:
declarator
declarator , *declarator-list*

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 [*constant-expression*$_{opt}$]
(*declarator*)

The grouping in this definition 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.

If a declarator has the form

∗ D

for D a declarator, then the contained identifier has the type "pointer to ...", where "..." is the type which the identifier would have had if the declarator had been simply D.

If a declarator has the form

D ( )

then the contained identifier has the type "function returning ...", where "..." is the type which the identifier would have had if the declarator had been simply D.

A declarator may have the form

D[constant-expression]

or

D[ ]

In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is `int`. in the second the constant 1 is used. (Constant expressions are defined precisely in §15.) Such a declarator makes the contained identifier have type "array." If the unadorned declarator D would specify a non-array of type "...", then the declarator "D[i]" yields a 1-dimensional array with rank i of objects of type "...". If the unadorned declarator D would specify an n-dimensional array with rank $i_1 \times i_2 \times \ldots \times i_n$, then the declarator "D[i_{n+1}]" yields an $(n+1)$-dimensional array with rank $i_1 \times i_2 \times \ldots \times i_n \times i_{n+1}$.

An array may be constructed from one of the basic types, from a pointer, from a structure, or from another array (to generate a multi-dimensional array).

Finally, parentheses in declarators do not alter the type of the contained identifier except insofar as they alter the binding of the components of the declarator.

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays, structures 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 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. Also

```
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 declarations

Recall that one of the forms for a structure specifier is

`struct` *{ type-decl-list }*

The *type-decl-list* is a sequence of type declarations for the members of the structure:

type-decl-list:
type-declaration
type-declaration type-decl-list

A type declaration is just a declaration which does not mention a storage class (the storage class "member of structure" here being understood by context).

type-declaration:
type-specifier declarator-list **;**

Within the structure, the objects declared have addresses which increase as their declarations are read left-to-right. Each component of a structure begins on an addressing boundary appropriate to its type. On the PDP-11 the only requirement is that non-characters begin on a word boundary; therefore, there may be 1-byte, unnamed holes in a structure, and all structures have an even length in bytes.

Another form of structure specifier is

`struct` *identifier { type-decl-list }*

This form is the same as the one just discussed, except that the identifier is remembered as the *structure tag* of the structure specified by the list. A subsequent declaration may then be given using the structure tag but without the list, as in the third form of structure specifier:

`struct` *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 however absurd to declare a structure which contains an instance of itself, as distinct from a pointer to an instance of itself.

A simple example of a structure declaration, taken from §16.2 where its use is illustrated more fully, 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 following declaration makes sense:

```
struct tnode s, *sp;
```

which declares *s* to be a structure of the given sort and *sp* to be a pointer to a structure of the given sort.

The names of structure members and structure tags may be the same as ordinary variables, since a distinction can be made by context. However, names of tags and members must be distinct. The same member name can appear in different structures only if the two members are of the same type and if their origin with respect to their structure is the same; thus separate structures can share a common initial segment.

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

So that several statements can be used where one is expected, the compound statement is provided:

compound-statement:
{ statement-list }

statement-list:
statement
statement statement-list

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 elseless `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 typically 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 expression must be `int` or `char`. The statement is typically compound. Each statement within the statement may be labelled with case prefixes as follows:

`case` *constant-expression* `:`

where the constant expression must be `int` or `char`. No two of the case constants in a 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 in an undefined order. 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. In the absence of a `default` prefix none of the statements in the switch is executed.

Case or default prefixes in themselves do not alter the flow of control.

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’’.

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 no value is returned. 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 expression ;
```

The expression should be a label (§§9.12, 14.4) or an expression of type ‘‘pointer to `int`’’ which evaluates to a label. It is illegal to transfer to a label not located in the current function unless some extra-language provision has been made to adjust the stack correctly.

9.12 Labelled statement

Any statement may be preceded by label prefixes of the form

```
identifier :
```

which serve to declare the identifier as a label. More details on the semantics of labels are given in §14.4 below.

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. External definitions may be given for functions, for simple variables, and for arrays. They are used both to declare and to reserve storage for objects. An external definition declares an identifier to have storage class `extern` and a specified type. The type-specifier (§8.2) may be empty, in which case the type is taken to be `int`.

10.1 External function definitions

Function definitions have the form

function-definition:
    type-specifier$_{opt}$ *function-declarator function-body*

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:
type-decl-list function-statement

The purpose of the type-decl-list is to give the types of the formal parameters. No other identifiers should be declared in this list, and formal parameters should be declared only here.

The function-statement is just a compound statement which may have declarations at the start.

function-statement:
{ declaration-list$_{opt}$ *statement-list }*

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 type-decl-list for the formal parameters; "{ ... }" is the function-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 neither structures nor functions can be passed to a function, it is useless to declare a formal parameter to be a structure or function (pointers to structures or functions are of course permitted).

A free `return` statement is supplied at the end of each function definition, so running off the end causes control, but no value, to be returned to the caller.

10.2 External data definitions

An external data definition has the form

data-definition:
`extern`$_{opt}$ *type-specifier*$_{opt}$ *init-declarator-list*$_{opt}$;

The optional `extern` specifier is discussed in § 11.2. If given, the init-declarator-list is a comma-separated list of declarators each of which may be followed by an initializer for the declarator.

init-declarator-list:
init-declarator
init-declarator , *init-declarator-list*

init-declarator:
declarator initializer$_{opt}$

Each initializer represents the initial value for the corresponding object being defined (and declared).

initializer:
constant
{ constant-expression-list }

constant-expression-list:
constant-expression
constant-expression , *constant-expression-list*

Thus an initializer consists of a constant-valued expression, or comma-separated list of expressions, inside braces. The braces may be dropped when the expression is just a plain constant. The exact meaning of a constant expression is discussed in §15. The expression list is used to initialize arrays; see below.

The type of the identifier being defined should be compatible with the type of the initializer: a `double` constant may initialize a `float` or `double` identifier; a non-floating-point expression may initialize an `int`, `char`, or pointer.

An initializer for an array may contain a comma-separated list of compile-time expressions. The length of the array is taken to be the maximum of the number of expressions in the list and the square-bracketed constant in the array's declarator. This constant may be missing, in which case 1 is used. The expressions initialize successive members of the array starting at the origin (subscript 0) of the array. The acceptable expressions for an array of type "array of ..." are the same as those for type "...". As a special case, a single string may be given as the initializer for an array of `char`s; in this case, the characters in the string are taken as the initializing values.

Structures can be initialized, but this operation is incompletely implemented and machine-dependent. Basically the structure is regarded as a sequence of words and the initializers are placed into those words. Structure initialization, using a comma-separated list in braces, is safe if all the members of the structure are integers or pointers but is otherwise ill-advised.

The initial value of any externally-defined object not explicitly initialized is guaranteed to be 0.

11. Scope rules

A complete 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

C is not a block-structured language; this may fairly be considered a defect. The lexical scope of names declared in external definitions extends from their definition through the end of the file in which they appear. The lexical scope of names declared at the head of functions (either as formal parameters or in the declarations heading the statements constituting the function itself) is the body of the function.

It is an error to redeclare identifiers already declared in the current context, unless the new declaration specifies the same type and storage class as already possessed by the identifiers.

11.2 Scope of externals

If a function declares an identifier 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 extent specified in the definition are compatible with those specified by each function which references the data.

In PDP-11 C, it is explicitly permitted for (compatible) external definitions of the same identifier to be present in several of the separately-compiled pieces of a complete program, or even twice within the same program file, with the important limitation that the identifier may be initialized in at most one of the definitions. In other operating systems, however, the compiler must know in just which file the storage for the identifier is allocated, and in which file the identifier is merely being referred to. In the implementations of C for such systems, the appearance of the `extern` keyword before 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 file where storage is allocated.

In PDP-11 C none of this nonsense is necessary and the `extern` specifier is ignored in external definitions.

12. Compiler control lines

When a line of a C program begins with the character `#`, it is interpreted not by the compiler itself, but by a preprocessor which is capable of replacing instances of given identifiers with arbitrary token-strings and of inserting named files into the source program. In order to cause this preprocessor to be invoked, it is necessary that the very first line of the program begin with `#`. Since null lines are ignored by the preprocessor, this line need contain no other information.

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 (except within compiler control lines). The replacement token-string has comments removed from it, and it is surrounded with blanks. No rescanning of the replacement string is attempted. This facility is most valuable for definition of ''manifest constants'', as in

```
# define tabsize 100
...
int table[tabsize];
```

12.2 File inclusion

Large C programs often contain many external data definitions. Since the lexical scope of external definitions extends to the end of the program file, it is good practice to put all the external definitions for data at the start of the program file, so that the functions defined within the file need not repeat tedious and error-prone declarations for each external identifier they use. It is also useful to put a heavily used structure definition at the start and use its structure tag to declare the `auto` pointers to the structure used within functions. To further exploit this technique when a large C program consists of several files, a compiler control line of the form

`# include "`*filename*`"`

results in the replacement of that line by the entire contents of the file *filename*.

13. Implicit declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. Sometimes 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 currently declared is contextually declared to be ''function returning `int`''.

Undefined identifiers not followed by `(` are assumed to be labels which will be defined later in the function. (Since a label is not an lvalue, this accounts for the ''Lvalue required'' error message sometimes noticed when an undeclared identifier is used.) Naturally, appearance of an identifier as a label declares it as such.

For some purposes it is best to consider formal parameters as belonging to their own storage class. In practice, C treats parameters as if they were automatic (except that, as mentioned above, formal parameter arrays and `floats` are treated specially).

14. Types revisited

This section summarizes the operations which can be performed on objects of certain types.

14.1 Structures

There are only two things that can be done with a structure: pick out one of its members (by means of the `.` or `->` operators); 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.

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* was declared explicitly in the calling routine since its first appearance 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 $i \times j \times \ldots \times k$, then E appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \ldots \times k$. 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 Labels

Labels do not have a type of their own; they are treated as having type "array of `int`". Label variables should be declared "pointer to `int`"; before execution of a `goto` referring to the variable, a label (or an expression deriving from a label) should be assigned to the variable.

Label variables are a bad idea in general; the `switch` statement makes them almost always unnecessary.

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

```
- ~
```

Parentheses can be used for grouping, but not for function calls.

A bit more latitude is permitted for initializers; besides constant expressions as discussed above, one can also apply the unary `&` operator to external scalars, and to external arrays subscripted with a constant expression. The unary `&` can also be applied implicitly by appearance of unsubscripted external arrays. The rule here is that initializers must evaluate either to a constant or to the address of an external identifier plus or minus a constant.

16. Examples.

These examples are intended to illustrate some typical C constructions as well as a serviceable style of writing C programs.

16.1 Inner product

This function returns the inner product of its array arguments.

```
double inner(v1, v2, n)
double v1[], v2[];
{
        double sum;
        int i;
        sum = 0.0;
        for (i=0; i<n; i++)
                sum =+ v1[i] * v2[i];
        return(sum);
}
```

The following version is somewhat more efficient, but perhaps a little less clear. It uses the facts that parameter arrays are really pointers, and that all parameters are passed by value.

```
double inner(v1, v2, n)
double *v1, *v2;
{
        double sum;
        sum = 0.0;
        while(n--)
                sum =+ *v1++ * *v2++;
        return(sum);
}
```

The declarations for the parameters are really exactly the same as in the last example. In the first case array declarations "`[]`" were given to emphasize that the parameters would be referred to as arrays; in the second, pointer declarations were given because the indirection operator and ++ were used.

16.2 Tree and character processing

Here is a complete C program (courtesy of R. Haight) which reads a document and produces an alphabetized list of words found therein together with the number of occurrences of each word. The method keeps a binary tree of words such that the left descendant tree for each word has all the words lexicographically smaller than the given word, and the right descendant has all the larger words. Both the insertion and the printing routine are recursive.

The program calls the library routines *getchar* to pick up characters and *exit* to terminate execution. *Printf* is

called to print the results according to a format string. A version of *printf* is given below (§16.3).

Because all the external definitions for data are given at the top, no `extern` declarations are necessary within the functions. To stay within the rules, a type declaration is given for each non-integer function when the function is used before it is defined. However, since all such functions return pointers which are simply assigned to other pointers, no actual harm would result from leaving out the declarations; the supposedly `int` function values would be assigned without error or complaint.

```
# define nwords 100                    /* number of different words */
# define wsize 20                      /* max chars per word */
struct tnode {                         /* the basic structure */
      char tword[wsize];
      int count;
      struct tnode *left;
      struct tnode *right;
};

struct tnode space[nwords];            /* the words themselves */
int nnodes nwords;                     /* number of remaining slots */
struct tnode *spacep space;            /* next available slot */
struct tnode *freep;                   /* free list */
/*
* The main routine reads words until end-of-file ('\0' returned from "getchar")
* "tree" is called to sort each word into the tree.
*/
main()
{
      struct tnode *top, *tree();
      char c, word[wsize];
      int i;

      i = top = 0;
      while (c=getchar())
            if ('a'<=c && c<='z' || 'A'<=c && c <='Z') {
                  if (i<wsize-1)
                        word[i++] = c;
            } else
                  if (i) {
                        word[i++] = '\0';
                        top = tree(top, word);
                        i = 0;
                  }
      tprint(top);
}
/*
* The central routine.  If the subtree pointer is null, allocate a new node for it.
* If the new word and the node's word are the same, increase the node's count.
* Otherwise, recursively sort the word into the left or right subtree according
* as the argument word is less or greater than the node's word.
*/
struct tnode *tree(p, word)
struct tnode *p;
char word[];
{
      struct tnode *alloc();
      int cond;

      /* Is pointer null? */
      if (p==0) {
            p = alloc();
```

```
            copy(word, p->tword);
            p->count = 1;
            p->right = p->left = 0;
            return(p);
      }
      /* Is word repeated? */
      if ((cond=compar(p->tword, word)) == 0) {
            p->count++;
            return(p);
      }
      /* Sort into left or right */
      if (cond<0)
            p->left = tree(p->left, word);
      else
            p->right = tree(p->right, word);
      return(p);
}
/*
* Print the tree by printing the left subtree, the given node, and the right subtree
*/
tprint(p)
struct tnode *p;
{
      while (p) {
            tprint(p->left);
            printf("%d:  %s\n", p->count, p->tword);
            p = p->right;
      }
}
/*
* String comparison: return number (>, =, <) 0
* according as s1 (>, =, <) s2.
*/
compar(s1, s2)
char *s1, *s2;
{
      int c1, c2;

      while((c1 = *s1++) == (c2 = *s2++))
            if (c1=='\0')
                  return(0);
      return(c2-c1);
}
/*
* String copy: copy s1 into s2 until the null
* character appears.
*/
copy(s1, s2)
char *s1, *s2;
{
      while(*s2++ = *s1++);
}
/*
* Node allocation: return pointer to a free node.
* Bomb out when all are gone.  Just for fun, there
* is a mechanism for using nodes that have been
* freed, even though no one here calls "free."
*/
struct tnode *alloc()
```

```
{
        struct tnode *t;

        if (freep) {
                t = freep;
                freep = freep->left;
                return(t);
        }
        if (--nnodes < 0) {
                printf("Out of space\n");
                exit();
        }
        return(spacep++);
}
/*
* The uncalled routine which puts a node on the free list.
*/
free(p)
struct tnode *p;
{
        p->left = freep;
        freep = p;
}
```

To illustrate a slightly different technique of handling the same problem, we will repeat fragments of this example with the tree nodes treated explicitly as members of an array. The fundamental change is to deal with the subscript of the array member under discussion, instead of a pointer to it. The `struct` declaration becomes

```
struct tnode {
        char tword[wsize];
        int count;
        int left;
        int right;
};
```

and *alloc* becomes

```
alloc()
{
        int t;

        t = --nnodes;
        if (t<=0) {
                printf("Out of space\n");
                exit();
        }
        return(t);
}
```

The *free* stuff has disappeared because if we deal with exclusively with subscripts some sort of map has to be kept, which is too much trouble.

Now the *tree* routine returns a subscript also, and it becomes:

```
tree(p, word)
char word[];
{
        int cond;

        if (p==0) {
                p = alloc();
                copy(word, space[p].tword);
```

```
                space[p].count = 1;
                space[p].right = space[p].left = 0;
                return(p);
        }
        if ((cond=compar(space[p].tword, word)) == 0) {
                space[p].count++;
                return(p);
        }
        if (cond<0)
                space[p].left = tree(space[p].left, word);
        else
                space[p].right = tree(space[p].right, word);
        return(p);
}
```

The other routines are changed similarly. It must be pointed out that this version is noticeably less efficient than the first because of the multiplications which must be done to compute an offset in *space* corresponding to the subscripts.

The observation that subscripts (like "a[i]") are less efficient than pointer indirection (like "*ap") holds true independently of whether or not structures are involved. There are of course many situations where subscripts are indispensable, and others where the loss in efficiency is worth a gain in clarity.

16.3 Formatted output

Here is a simplified version of the *printf* routine, which is available in the C library. It accepts a string (character array) as first argument, and prints subsequent arguments according to specifications contained in this format string. Most characters in the string are simply copied to the output; two-character sequences beginning with "%" specify that the next argument should be printed in a style as follows:

%d	decimal number
%o	octal number
%c	ASCII character, or 2 characters if upper character is not null
%s	string (null-terminated array of characters)
%f	floating-point number

The actual parameters for each function call are laid out contiguously in increasing storage locations; therefore, a function with a variable number of arguments may take the address of (say) its first argument, and access the remaining arguments by use of subscripting (regarding the arguments as an array) or by indirection combined with pointer incrementation.

If in such a situation the arguments have mixed types, or if in general one wishes to insist that an lvalue should be treated as having a given type, then `struct` declarations like those illustrated below will be useful. It should be evident, though, that such techniques are implementation dependent.

Printf depends as well on the fact that `char` and `float` arguments are widened respectively to `int` and `double`, so there are effectively only two sizes of arguments to deal with. *Printf* calls the library routines *putchar* to write out single characters and *ftoa* to dispose of floating-point numbers.

```
printf(fmt, args)
char fmt[];
{
        char *s;
        struct { char **charpp; };
        struct { double *doublep; };
        int *ap, x, c;

        ap = &args;             /* argument pointer */
        for (;;) {
                while((c = *fmt++) != '%') {
                        if(c == '\0')
                                return;
```

```
                        putchar(c);
                }
                switch (c = *fmt++) {
                /* decimal */
                case 'd':
                        x = *ap++;
                        if(x < 0) {
                                x = -x;
                                if(x<0) {  /* is - infinity */
                                        printf("-32768");
                                        continue;
                                }
                                putchar('-');
                        }
                        printd(x);
                        continue;
                /* octal */
                case 'o':
                        printo(*ap++);
                        continue;
                /* float, double */
                case 'f':
                        /* let ftoa do the real work */
                        ftoa(*ap.doublep++);
                        continue;
                /* character */
                case 'c':
                        putchar(*ap++);
                        continue;
                /* string */
                case 's':
                        s = *ap.charpp++;
                        while(c = *s++)
                                putchar(c);
                        continue;
                }
                putchar(c);
        }
}
/*
* Print n in decimal; n must be non-negative
*/
printd(n)
{
        int a;
        if (a=n/10)
                printd(a);
        putchar(n%10 + '0');
}
/*
* Print n in octal, with exactly 1 leading 0
*/
printo(n)
{
        if (n)
                printo( (n>>3)&017777);
        putchar( (n&07)+'0');
}
```

REFERENCES

1. Johnson, S. C., and Kernighan, B. W. ''The Programming Language B.'' Comp. Sci. Tech. Rep. #8., Bell Laboratories, 1972.

2. Ritchie, D. M., and Thompson, K. L. ''The UNIX Time-sharing System.'' C. ACM 7, 17, July, 1974, pp. 365-375.

3. Peterson, T. G., and Lesk, M. E. ''A User's Guide to the C Language on the IBM 370.'' Internal Memorandum, Bell Laboratories, 1974.

4. Thompson, K. L., and Ritchie, D. M. *UNIX Programmer's Manual.* Bell Laboratories, 1973.

5. Lesk, M. E., and Barres, B. A. ''The GCOS C Library.'' Internal memorandum, Bell Laboratories, 1974.

6. Kernighan, B. W. ''Programming in C– A Tutorial.'' Unpublished internal memorandum, Bell Laboratories, 1974.

APPENDIX 1
Syntax Summary

1. Expressions.

expression:
primary
`*` *expression*
`&` *expression*
`-` *expression*
`!` *expression*
`~`*expression*
`++` *lvalue*
`--` *lvalue*
lvalue `++`
lvalue `--`
`sizeof` *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
```

have priority below the primary operators but higher than any binary operator, and group right-to-left. Binary operators and the conditional operator all group left-to-right, and have priority decreasing as indicated:

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.

2. Declarations.

declaration:
decl-specifiers declarator-list$_{opt}$ **;**

decl-specifiers:
type-specifier
sc-specifier
type-specifier sc-specifier
sc-specifier type-specifier

sc-specifier:
`auto`
`static`
`extern`
`register`

type-specifier:
`int`
`char`
`float`
`double`
`struct { ` *type-decl-list* ` }`
`struct` *identifier* `{` *type-decl-list* `}`
`struct` *identifier*

declarator-list:
declarator
declarator **,** *declarator-list*

declarator:
identifier
`*` *declarator*
declarator `( )`
declarator `[` *constant-expression*$_{opt}$ `]`
`(` *declarator* `)`

type-decl-list:
type-declaration
type-declaration type-decl-list

type-declaration:
type-specifier declarator-list **;**

3. Statements.

statement:
expression **;**
{ statement-list }

`if` (*expression*) *statement*
`if` (*expression*) *statement* `else` *statement*
`while` (*expression*) *statement*
`for` ($expression_{opt}$ `;` $expression_{opt}$ `;` $expression_{opt}$) `statement`
`switch` (*expression*) *statement*
`case` *constant-expression* `:` *statement*
`default` `:` *statement*
`break` `;`
`continue` `;`
`return` `;`
`return` (*expression*) `;`
`goto` *expression* `;`
identifier `:` *statement*
`;`

statement-list:
statement
statement statement-list

4. External definitions.

program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition:
$type\text{-}specifier_{opt}$ *function-declarator function-body*

function-declarator:
declarator ($parameter\text{-}list_{opt}$)

parameter-list:
identifier
identifier `,` *parameter-list*

function-body:
type-decl-list function-statement

function-statement:
`{` $declaration\text{-}list_{opt}$ *statement-list* `}`

data-definition:
`extern`$_{opt}$ $type\text{-}specifier_{opt}$ $init\text{-}declarator\text{-}list_{opt}$ `;`

init-declarator-list:
init-declarator
init-declarator `,` *init-declarator-list*

init-declarator:
declarator $initializer_{opt}$

initializer:
constant
`{` *constant-expression-list* `}`

constant-expression-list:
constant-expression
constant-expression , *constant-expression-list*

constant-expression:
expression

5. Preprocessor

`# define` *identifier token-string*

`# include` "*filename*"

APPENDIX 2
Implementation Peculiarities

This Appendix briefly summarizes the differences between the implementations of C on the PDP-11 under UNIX and on the HIS 6070 under GCOS; it includes some known bugs in each implementation. Each entry is keyed by an indicator as follows:

h	hard to fix
g	GCOS version should probably be changed
u	UNIX version should probably be changed
d	Inherent difference likely to remain

This list was prepared by M. E. Lesk, S. C. Johnson, E. N. Pinson, and the author.

A. Bugs or differences from C language specifications

hg	A.1)	GCOS does not do type conversions in "?:".
hg	A.2)	GCOS has a bug in `int` and `real` comparisons; the numbers are compared by subtraction, and the difference must not overflow.
g	A.3)	When *x* is a `float`, the construction "test ? –x : x" is illegal on GCOS.
hg	A.4)	"p1–>p2 =+ 2" causes a compiler error, where p1 and p2 are pointers.
u	A.5)	On UNIX, the expression in a `return` statement is *not* converted to the type of the function, as promised.
hug	A.6)	`entry` statement is not implemented at all.

B. Implementation differences

d	B.1)	Sizes of character constants differ; UNIX: 2, GCOS: 4.
d	B.2)	Table sizes in compilers differ.
d	B.3)	`char`s and `int`s have different sizes; `char`s are 8 bits on UNIX, 9 on GCOS; words are 16 bits on UNIX and 36 on GCOS. There are corresponding differences in representations of `float`s and `double`s.
d	B.4)	Character arrays stored left to right in a word in GCOS, right to left in UNIX.
g	B.5)	Passing of floats and doubles differs; UNIX passes on stack, GCOS passes pointer (hidden to normal user).
g	B.6)	Structures and strings are aligned on a word boundary in UNIX, not aligned in GCOS.
g	B.7)	GCOS preprocessor supports #rename, #escape; UNIX has only #define, #include.
u	B.8)	Preprocessor is not invoked on UNIX unless first character of file is "#".
u	B.9)	The external definition "static int ..." is legal on GCOS, but gets a diagnostic on UNIX. (On GCOS it means an identifier global to the routines in the file but invisible to routines compiled separately.)
g	B.10)	A compound statement on GCOS must contain one ";" but on UNIX may be empty.
g	B.11)	On GCOS case distinctions in identifiers and keywords are ignored; on UNIX case is significant everywhere, with keywords in lower case.

C. Syntax Differences

g	C.1)	UNIX allows broader classes of initialization; on GCOS an initializer must be a constant, name, or string. Similarly, GCOS is much stickier about wanting braces around initializers and in particular they must be present for array initialization.
g	C.2)	"int extern" illegal on GCOS; must have "extern int" (storage class before type).
g	C.3)	Externals on GCOS must have a type (not defaulted to `int`).
u	C.4)	GCOS allows initialization of internal `static` (same syntax as for external definitions).
g	C.5)	integer–>... is not allowed on GCOS.
g	C.6)	Some operators on pointers are illegal on GCOS (<, >).

g	C.7)	`register` storage class means something on UNIX, but is not accepted on GCOS.
g	C.8)	Scope holes: ''int x; f( ) {int x;}'' is illegal on UNIX but defines two variables on GCOS.
g	C.9)	When function names are used as arguments on UNIX, either ''fname'' or ''&fname'' may be used to get a pointer to the function; on GCOS ''&fname'' generates a doubly-indirect pointer. (Note that both are wrong since the ''&'' is supposed to be supplied for free.)

D. Operating System Dependencies

d	D.1)	GCOS allocates external scalars by SYMREF; UNIX allocates external scalars as labelled common; as a result there may be many uninitialized external definitions of the same variable on UNIX but only one on GCOS.
d	D.2)	External names differ in allowable length and character set; on UNIX, 7 characters and both cases; on GCOS 6 characters and only one case.

E. Semantic Differences

hg	E.1)	''int i, *p; p=i; i=p;'' does nothing on UNIX, does something on GCOS (destroys right half of i) .
d	E.2)	''>>'' means arithmetic shift on UNIX, logical on GCOS.
d	E.3)	When a `char` is converted to integer, the result is always positive on GCOS but can be negative on UNIX.
d	E.4)	Arguments of subroutines are evaluated left-to-right on GCOS, right-to-left on UNIX.

Programming in C — A Tutorial

Brian W. Kernighan

Bell Laboratories, Murray Hill, N. J.

1. Introduction

C is a computer language available on the GCOS and UNIX operating systems at Murray Hill and (in preliminary form) on OS/360 at Holmdel. C lets you write your programs clearly and simply — it has decent control flow facilities so your code can be read straight down the page, without labels or GOTO's; it lets you write code that is compact without being too cryptic; it encourages modularity and good program organization; and it provides good data-structuring facilities.

This memorandum is a tutorial to make learning C as painless as possible. The first part concentrates on the central features of C; the second part discusses those parts of the language which are useful (usually for getting more efficient and smaller code) but which are not necessary for the new user. This is *not* a reference manual. Details and special cases will be skipped ruthlessly, and no attempt will be made to cover every language feature. The order of presentation is hopefully pedagogical instead of logical. Users who would like the full story should consult the *C Reference Manual* by D. M. Ritchie [1], which should be read for details anyway. Runtime support is described in [2] and [3]; you will have to read one of these to learn how to compile and run a C program.

We will assume that you are familiar with the mysteries of creating files, text editing, and the like in the operating system you run on, and that you have programmed in some language before.

2. A Simple C Program

```
main( ) {
        printf("hello, world");
}
```

A C program consists of one or more *functions*, which are similar to the functions and subroutines of a Fortran program or the procedures of PL/I, and perhaps some external data definitions. `main` is such a function, and in fact all C programs must have a `main`. Execution of the program begins at the first statement of `main`. `main` will usually invoke other functions to perform its job, some coming from the same program, and others from libraries.

One method of communicating data between functions is by arguments. The parentheses following the function name surround the argument list; here `main` is a function of no arguments, indicated by ( ). The {} enclose the statements of the function. Individual statements end with a semicolon but are otherwise free-format.

`printf` is a library function which will format and print output on the terminal (unless some other destination is specified). In this case it prints

```
hello, world
```

A function is invoked by naming it, followed by a list of arguments in parentheses. There is no CALL statement as in Fortran or PL/I.

3. A Working C Program; Variables; Types and Type Declarations

Here's a bigger program that adds three integers and prints their sum.

```
main( ) {
      int a, b, c, sum;
      a = 1;  b = 2;  c = 3;
      sum = a + b + c;
      printf("sum is %d", sum);
}
```

Arithmetic and the assignment statements are much the same as in Fortran (except for the semicolons) or PL/I. The format of C programs is quite free. We can put several statements on a line if we want, or we can split a statement among several lines if it seems desirable. The split may be between any of the operators or variables, but *not* in the middle of a name or operator. As a matter of style, spaces, tabs, and newlines should be used freely to enhance readability.

C has four fundamental *types* of variables:

`int`	integer (PDP-11: 16 bits; H6070: 36 bits; IBM360: 32 bits)
`char`	one byte character (PDP-11, IBM360: 8 bits; H6070: 9 bits)
`float`	single-precision floating point
`double`	double-precision floating point

There are also *arrays* and *structures* of these basic types, *pointers* to them and *functions* that return them, all of which we will meet shortly.

All variables in a C program must be declared, although this can sometimes be done implicitly by context. Declarations must precede executable statements. The declaration

```
int a, b, c, sum;
```

declares `a, b, c,` and `sum` to be integers.

Variable names have one to eight characters, chosen from A-Z, a-z, 0-9, and _, and start with a non-digit. Stylistically, it's much better to use only a single case and give functions and external variables names that are unique in the first six characters. (Function and external variable names are used by various assemblers, some of which are limited in the size and case of identifiers they can handle.) Furthermore, keywords and library functions may only be recognized in one case.

4. Constants

We have already seen decimal integer constants in the previous example — 1, 2, and 3. Since C is often used for system programming and bit-manipulation, octal numbers are an important part of the language. In C, any number that begins with 0 (zero!) is an octal integer (and hence can't have any 8's or 9's in it). Thus 0777 is an octal constant, with decimal value 511.

A "character" is one byte (an inherently machine-dependent concept). Most often this is expressed as a *character constant,* which is one character enclosed in single quotes. However, it may be any quantity that fits in a byte, as in `flags` below:

```
char quest, newline, flags;
quest = '?';
newline = '\n';
flags = 077;
```

The sequence '\n' is C notation for "newline character", which, when printed, skips the terminal to the beginning of the next line. Notice that '\n' represents only a single character. There are several other "escapes" like '\n' for representing hard-to-get or invisible characters, such as '\t' for tab, '\b' for backspace, '\0' for end of file, and '\\' for the backslash itself.

`float` and `double` constants are discussed in section 26.

5. Simple I/O — getchar, putchar, printf

```
main( ) {
        char c;
        c = getchar( );
        putchar(c);
}
```

`getchar` and `putchar` are the basic I/O library functions in C. `getchar` fetches one character from the standard input (usually the terminal) each time it is called, and returns that character as the value of the function. When it reaches the end of whatever file it is reading, thereafter it returns the character represented by '\0' (ascii NUL, which has value zero). We will see how to use this very shortly.

`putchar` puts one character out on the standard output (usually the terminal) each time it is called. So the program above reads one character and writes it back out. By itself, this isn't very interesting, but observe that if we put a loop around this, and add a test for end of file, we have a complete program for copying one file to another.

`printf` is a more complicated function for producing formatted output. We will talk about only the simplest use of it. Basically, `printf` uses its first argument as formatting information, and any successive arguments as variables to be output. Thus

```
printf ("hello, world\n");
```

is the simplest use — the string "hello, world\n" is printed out. No formatting information, no variables, so the string is dumped out verbatim. The newline is necessary to put this out on a line by itself. (The construction

```
"hello, world\n"
```

is really an array of `chars`. More about this shortly.)

More complicated, if `sum` is 6,

```
printf ("sum is %d\n", sum);
```

prints

```
sum is 6
```

Within the first argument of `printf`, the characters "%d" signify that the next argument in the argument list is to be printed as a base 10 number.

Other useful formatting commands are "%c" to print out a single character, "%s" to print out an entire string, and "%o" to print a number as octal instead of decimal (no leading zero). For example,

```
n = 511;
printf ("What is the value of %d in octal?", n);
printf ("  %s! %d decimal is %o octal\n", "Right", n, n);
```

prints

```
What is the value of 511 in octal?  Right! 511 decimal is 777
octal
```

Notice that there is no newline at the end of the first output line. Successive calls to `printf` (and/or `putchar`, for that matter) simply put out characters. No newlines are printed unless you ask for them. Similarly, on input, characters are read one at a time as you ask for them. Each line is generally terminated

by a newline (\n), but there is otherwise no concept of record.

6. If; relational operators; compound statements

The basic conditional-testing statement in C is the `if` statement:

```
c = getchar( );
if( c == '?' )
      printf("why did you type a question mark?\n");
```

The simplest form of `if` is

```
if (expression) statement
```

The condition to be tested is any expression enclosed in parentheses. It is followed by a statement. The expression is evaluated, and if its value is non-zero, the statement is executed. There's an optional `else` clause, to be described soon.

The character sequence '==' is one of the relational operators in C; here is the complete set:

```
==    equal to (.EQ. to Fortraners)
!=    not equal to
>     greater than
<     less than
>=    greater than or equal to
<=    less than or equal to
```

The value of ''expression relation expression'' is 1 if the relation is true, and 0 if false. Don't forget that the equality test is '=='; a single '=' causes an assignment, not a test, and invariably leads to disaster.

Tests can be combined with the operators `'&&'` (AND), `'||'` (OR), and `'!'` (NOT). For example, we can test whether a character is blank or tab or newline with

```
if( c==' ' || c=='\t' || c=='\n' )  . . .
```

C guarantees that `'&&'` and `'||'` are evaluated left to right — we shall soon see cases where this matters.

One of the nice things about C is that the `statement` part of an `if` can be made arbitrarily complicated by enclosing a set of statements in {}. As a simple example, suppose we want to ensure that `a` is bigger than `b`, as part of a sort routine. The interchange of `a` and `b` takes three statements in C, grouped together by {}:

```
if (a < b) {
      t = a;
      a = b;
      b = t;
}
```

As a general rule in C, anywhere you can use a simple statement, you can use any compound statement, which is just a number of simple or compound ones enclosed in {}. There is no semicolon after the } of a compound statement, but there *is* a semicolon after the last non-compound statement inside the {}.

The ability to replace single statements by complex ones at will is one feature that makes C much more pleasant to use than Fortran. Logic (like the exchange in the previous example) which would require several GOTO's and labels in Fortran can and should be done in C without any, using compound statements.

7. While Statement; Assignment within an Expression; Null Statement

The basic looping mechanism in C is the `while` statement. Here's a program that copies its input to its output a character at a time. Remember that '\0' marks the end of file.

```
main( ) {
```

```
        char c;
        while( (c=getchar( )) != '\0' )
              putchar(c);
}
```

The `while` statement is a loop, whose general form is

```
while (expression) statement
```

Its meaning is

```
(a) evaluate the expression
(b) if its value is true (i.e., not zero)
            do the statement, and go back to (a)
```

Because the expression is tested before the statement is executed, the statement part can be executed zero times, which is often desirable. As in the `if` statement, the expression and the statement can both be arbitrarily complicated, although we haven't seen that yet. Our example gets the character, assigns it to `c`, and then tests if it's a '\0''. If it is not a '\0', the statement part of the `while` is executed, printing the character. The `while` then repeats. When the input character is finally a '\0', the `while` terminates, and so does `main`.

Notice that we used an assignment statement

```
c = getchar( )
```

within an expression. This is a handy notational shortcut which often produces clearer code. (In fact it is often the only way to write the code cleanly. As an exercise, re-write the file-copy without using an assignment inside an expression.) It works because an assignment statement has a value, just as any other expression does. Its value is the value of the right hand side. This also implies that we can use multiple assignments like

```
x = y = z = 0;
```

Evaluation goes from right to left.

By the way, the extra parentheses in the assignment statement within the conditional were really necessary: if we had said

```
c = getchar( ) != '\0'
```

`c` would be set to 0 or 1 depending on whether the character fetched was an end of file or not. This is because in the absence of parentheses the assignment operator '=' is evaluated after the relational operator '!='. When in doubt, or even if not, parenthesize.

Since `putchar(c)` returns `c` as its function value, we could also copy the input to the output by nesting the calls to `getchar` and `putchar`:

```
main( ) {
        while( putchar(getchar( )) != '\0' ) ;
}
```

What statement is being repeated? None, or technically, the *null* statement, because all the work is really done within the test part of the `while`. This version is slightly different from the previous one, because the final '\0' is copied to the output before we decide to stop.

8. Arithmetic

The arithmetic operators are the usual '+', '–', '*', and '/' (truncating integer division if the operands are both `int`), and the remainder or mod operator '%':

```
x = a%b;
```

sets `x` to the remainder after `a` is divided by `b` (i.e., `a` mod `b`). The results are machine dependent unless

a and b are both positive.

In arithmetic, char variables can usually be treated like int variables. Arithmetic on characters is quite legal, and often makes sense:

```
c = c + 'A' - 'a';
```

converts a single lower case ascii character stored in c to upper case, making use of the fact that corresponding ascii letters are a fixed distance apart. The rule governing this arithmetic is that all chars are converted to int before the arithmetic is done. Beware that conversion may involve sign-extension — if the leftmost bit of a character is 1, the resulting integer might be negative. (This doesn't happen with genuine characters on any current machine.)

So to convert a file into lower case:

```
main( ) {
        char c;
        while( (c=getchar( )) != '\0' )
                if( 'A'<=c && c<='Z' )
                        putchar(c+'a'-'A');
                else
                        putchar(c);
}
```

Characters have different sizes on different machines. Further, this code won't work on an IBM machine, because the letters in the ebcdic alphabet are not contiguous.

9. Else Clause; Conditional Expressions

We just used an else after an if. The most general form of if is

```
if (expression) statement1 else statement2
```

the else part is optional, but often useful. The canonical example sets x to the minimum of a and b:

```
if (a < b)
        x = a;
else
        x = b;
```

Observe that there's a semicolon after x=a.

C provides an alternate form of conditional which is often more concise. It is called the "conditional expression" because it is a conditional which actually has a value and can be used anywhere an expression can. The value of

```
a<b ? a : b;
```

is a if a is less than b; it is b otherwise. In general, the form

```
expr1 ? expr2 : expr3
```

means "evaluate expr1. If it is not zero, the value of the whole thing is expr2; otherwise the value is expr3."

To set x to the minimum of a and b, then:

```
x = (a<b ? a : b);
```

The parentheses aren't necessary because '?:' is evaluated before '=', but safety first.

Going a step further, we could write the loop in the lower-case program as

```
while( (c=getchar( )) != '\0' )
        putchar( ('A'<=c && c<='Z') ? c-'A'+'a' : c );
```

If's and else's can be used to construct logic that branches one of several ways and then rejoins, a common programming structure, in this way:

```
if(. . .)
      {. . .}
else if(. . .)
      {. . .}
else if(. . .)
      {. . .}
else
      {. . .}
```

The conditions are tested in order, and exactly one block is executed — either the first one whose if is satisfied, or the one for the last else. When this block is finished, the next statement executed is the one after the last else. If no action is to be taken for the "default" case, omit the last else.

For example, to count letters, digits and others in a file, we could write

```
main( ) {
      int let, dig, other, c;
      let = dig = other = 0;
      while( (c=getchar( )) != '\0' )
            if( ('A'<=c && c<='Z') || ('a'<=c && c<='z') )  ++let;
            else if( '0'<=c && c<='9' )  ++dig;
            else  ++other;
      printf("%d letters, %d digits, %d others\n", let, dig, other);
}
```

The '++' operator means "increment by 1"; we will get to it in the next section.

10. Increment and Decrement Operators

In addition to the usual '–', C also has two other interesting unary operators, '++' (increment) and '––' (decrement). Suppose we want to count the lines in a file.

```
main( ) {
      int c,n;
      n = 0;
      while( (c=getchar( )) != '\0' )
            if( c == '\n' )
                  ++n;
      printf("%d lines\n", n);
}
```

++n is equivalent to n=n+1 but clearer, particularly when n is a complicated expression. '++' and '––' can be applied only to int's and char's (and pointers which we haven't got to yet).

The unusual feature of '++' and '––' is that they can be used either before or after a variable. The value of ++k is the value of k *after* it has been incremented. The value of k++ is k *before* it is incremented. Suppose k is 5. Then

```
x = ++k;
```

increments k to 6 and then sets x to the resulting value, i.e., to 6. But

```
x = k++;
```

first sets x to to 5, and *then* increments k to 6. The incrementing effect of ++k and k++ is the same, but their values are respectively 5 and 6. We shall soon see examples where both of these uses are important.

11. Arrays

In C, as in Fortran or PL/I, it is possible to make arrays whose elements are basic types. Thus we can make an array of 10 integers with the declaration

```
int x[10];
```

The square brackets mean *subscripting;* parentheses are used only for function references. Array indexes begin at *zero,* so the elements of `x` are

```
x[0], x[1], x[2], . . . , x[9]
```

If an array has `n` elements, the largest subscript is `n-1` .

Multiple-dimension arrays are provided, though not much used above two dimensions. The declaration and use look like

```
int name[10] [20];
n = name[i+j] [1] + name[k] [2];
```

Subscripts can be arbitrary integer expressions. Multi-dimension arrays are stored by row (opposite to Fortran), so the rightmost subscript varies fastest; `name` has 10 rows and 20 columns.

Here is a program which reads a line, stores it in a buffer, and prints its length (excluding the newline at the end).

```
main( ) {
      int n, c;
      char line[100];
      n = 0;
      while( (c=getchar( )) != '\n' ) {
            if( n < 100 )
                  line[n] = c;
            n++;
      }
      printf("length = %d\n", n);
}
```

As a more complicated problem, suppose we want to print the count for each line in the input, still storing the first 100 characters of each line. Try it as an exercise before looking at the solution:

```
main( ) {
      int n, c; char line[100];
      n = 0;
      while( (c=getchar( )) != '\0' )
            if( c == '\n' ) {
                  printf("%d0, n);
                  n = 0;
            }
            else {
                  if( n < 100 ) line[n] = c;
                  n++;
            }
}
```

12. Character Arrays; Strings

Text is usually kept as an array of characters, as we did with `line[ ]` in the example above. By convention in C, the last character in a character array should be a '\0' because most programs that manipulate character arrays expect it. For example, `printf` uses the '\0' to detect the end of a character array

when printing it out with a '%s'.

We can copy a character array `s` into another `t` like this:

```
i = 0;
while( (t[i]=s[i]) != '\0' )
      i++;
```

Most of the time we have to put in our own '\0' at the end of a string; if we want to print the line with `printf`, it's necessary. This code prints the character count before the line:

```
main( ) {
      int n;
      char line[100];
      n = 0;
      while( (line[n++]=getchar( )) != '\n' );
      line[n] = '\0';
      printf("%d:\t%s", n, line);
}
```

Here we increment `n` in the subscript itself, but only after the previous value has been used. The character is read, placed in `line[n]`, and only then `n` is incremented.

There is one place and one place only where C puts in the '\0' at the end of a character array for you, and that is in the construction

```
"stuff between double quotes"
```

The compiler puts a '\0' at the end automatically. Text enclosed in double quotes is called a *string;* its properties are precisely those of an (initialized) array of characters.

13. For Statement

The `for` statement is a somewhat generalized `while` that lets us put the initialization and increment parts of a loop into a single statement along with the test. The general form of the `for` is

```
for( initialization; expression; increment )
      statement
```

The meaning is exactly

```
initialization;
while( expression ) {
      statement
      increment;
}
```

Thus, the following code does the same array copy as the example in the previous section:

```
for( i=0; (t[i]=s[i]) != '\0'; i++ );
```

This slightly more ornate example adds up the elements of an array:

```
sum = 0;
for( i=0; i<n; i++)
      sum = sum + array[i];
```

In the `for` statement, the initialization can be left out if you want, but the semicolon has to be there. The increment is also optional. It is *not* followed by a semicolon. The second clause, the test, works the same way as in the `while`: if the expression is true (not zero) do another loop, otherwise get on with the next statement. As with the `while`, the `for` loop may be done zero times. If the expression is left out, it is taken to be always true, so

```
for( ; ; ) . . .
```

and

```
while( 1 ) ...
```

are both infinite loops.

You might ask why we use a `for` since it's so much like a `while`. (You might also ask why we use a `while` because...) The `for` is usually preferable because it keeps the code where it's used and sometimes eliminates the need for compound statements, as in this code that zeros a two-dimensional array:

```
for( i=0; i<n; i++ )
      for( j=0; j<m; j++ )
            array[i][j] = 0;
```

14. Functions; Comments

Suppose we want, as part of a larger program, to count the occurrences of the ascii characters in some input text. Let us also map illegal characters (those with value>127 or <0) into one pile. Since this is presumably an isolated part of the program, good practice dictates making it a separate function. Here is one way:

```
main( ) {
      int hist[129];          /* 128 legal chars + 1 illegal group */
      ...
      count(hist, 128); /* count the letters into hist */
      printf( ... );        /* comments look like this; use them */
      ...         /* anywhere blanks, tabs or newlines could appear */
}

count(buf, size)
    int size, buf[ ]; {
      int i, c;
      for( i=0; i<=size; i++ )
            buf[i] = 0;             /* set buf to zero */
      while( (c=getchar( )) != '\0' ) {   /* read til eof */
            if( c > size || c < 0 )
                  c = size;         /* fix illegal input */
            buf[c]++;
      }
      return;
}
```

We have already seen many examples of calling a function, so let us concentrate on how to *define* one. Since `count` has two arguments, we need to declare them, as shown, giving their types, and in the case of `buf`, the fact that it is an array. The declarations of arguments go *between* the argument list and the opening '{'. There is no need to specify the size of the array `buf`, for it is defined outside of `count`.

The `return` statement simply says to go back to the calling routine. In fact, we could have omitted it, since a return is implied at the end of a function.

What if we wanted `count` to return a value, say the number of characters read? The `return` statement allows for this too:

```
      int i, c, nchar;
      nchar = 0;
      ...
      while( (c=getchar( )) != '\0' ) {
            if( c > size || c < 0 )
                  c = size;
            buf[c]++;
            nchar++;
```

```
        }
        return(nchar);
```

Any expression can appear within the parentheses. Here is a function to compute the minimum of two integers:

```
min(a, b)
    int a, b; {
      return( a < b ? a : b );
}
```

To copy a character array, we could write the function

```
strcopy(s1, s2)         /* copies s1 to s2 */
    char s1[ ], s2[ ]; {
      int i;
      for( i = 0; (s2[i] = s1[i]) != '\0'; i++ );
}
```

As is often the case, all the work is done by the assignment statement embedded in the test part of the `for`. Again, the declarations of the arguments `s1` and `s2` omit the sizes, because they don't matter to `strcopy`. (In the section on pointers, we will see a more efficient way to do a string copy.)

There is a subtlety in function usage which can trap the unsuspecting Fortran programmer. Simple variables (not arrays) are passed in C by "call by value", which means that the called function is given a copy of its arguments, and doesn't know their addresses. This makes it impossible to change the value of one of the actual input arguments.

There are two ways out of this dilemma. One is to make special arrangements to pass to the function the address of a variable instead of its value. The other is to make the variable a global or external variable, which is known to each function by its name. We will discuss both possibilities in the next few sections.

15. Local and External Variables

If we say

```
f( ) {
      int x;
      • • •
}
g( ) {
      int x;
      • • •
}
```

each `x` is *local* to its own routine — the `x` in `f` is unrelated to the `x` in `g`. (Local variables are also called "automatic".) Furthermore each local variable in a routine appears only when the function is called, and *disappears* when the function is exited. Local variables have no memory from one call to the next and must be explicitly initialized upon each entry. (There is a `static` storage class for making local variables with memory; we won't discuss it.)

As opposed to local variables, *external variables* are defined external to all functions, and are (potentially) available to all functions. External storage always remains in existence. To make variables external we have to *define* them external to all functions, and, wherever we want to use them, make a *declaration.*

```
main( ) {
      extern int nchar, hist[ ];
      • • •
      count( );
      • • •
}
```

```
count( ) {
      extern int nchar, hist[ ];
      int i, c;
      • • •
}

int   hist[129];  /* space for histogram */
int   nchar;            /* character count */
```

Roughly speaking, any function that wishes to access an external variable must contain an `extern` declaration for it. The declaration is the same as others, except for the added keyword `extern`. Furthermore, there must somewhere be a *definition* of the external variables external to all functions.

External variables can be initialized; they are set to zero if not explicitly initialized. In its simplest form, initialization is done by putting the value (which must be a constant) after the definition:

```
int   nchar 0;
char  flag  'f';
 etc.
```

This is discussed further in a later section.

This ends our discussion of what might be called the central core of C. You now have enough to write quite substantial C programs, and it would probably be a good idea if you paused long enough to do so. The rest of this tutorial will describe some more ornate constructions, useful but not essential.

16. Pointers

A *pointer* in C is the address of something. It is a rare case indeed when we care what the specific address itself is, but pointers are a quite common way to get at the contents of something. The unary operator '&' is used to produce the address of an object, if it has one. Thus

```
int a, b;
b = &a;
```

puts the address of `a` into `b`. We can't do much with it except print it or pass it to some other routine, because we haven't given `b` the right kind of declaration. But if we declare that `b` is indeed a *pointer* to an integer, we're in good shape:

```
int a, *b, c;
b = &a;
c = *b;
```

`b` contains the address of `a` and '`c = *b`' means to use the value in `b` as an address, i.e., as a pointer. The effect is that we get back the contents of `a`, albeit rather indirectly. (It's always the case that '`*&x`' is the same as `x` if `x` has an address.)

The most frequent use of pointers in C is for walking efficiently along arrays. In fact, in the implementation of an array, the array name represents the address of the zeroth element of the array, so you can't use it on the left side of an expression. (You can't change the address of something by assigning to it.) If we say

```
char *y;
char x[100];
```

`y` is of type pointer to character (although it doesn't yet point anywhere). We can make `y` point to an element of `x` by either of

```
y = &x[0];
y = x;
```

Since `x` is the address of `x[0]` this is legal and consistent.

Now `'*y'` gives `x[0]`. More importantly,

```
*(y+1)      gives x[1]
*(y+i)      gives x[i]
```

and the sequence

```
y = &x[0];
y++;
```

leaves `y` pointing at `x[1]`.

Let's use pointers in a function `length` that computes how long a character array is. Remember that by convention all character arrays are terminated with a '\0'. (And if they aren't, this program will blow up inevitably.) The old way:

```
length(s)
    char s[ ]; {
      int n;
      for( n=0; s[n] != '\0'; )
            n++;
      return(n);
}
```

Rewriting with pointers gives

```
length(s)
    char *s; {
      int n;
      for( n=0; *s != '\0'; s++ )
            n++;
      return(n);
}
```

You can now see why we have to say what kind of thing `s` points to — if we're to increment it with `s++` we have to increment it by the right amount.

The pointer version is more efficient (this is almost always true) but even more compact is

```
for( n=0; *s++ != '\0'; n++ );
```

The `'*s'` returns a character; the `'++'` increments the pointer so we'll get the next character next time around. As you can see, as we make things more efficient, we also make them less clear. But `'*s++'` is an idiom so common that you have to know it.

Going a step further, here's our function `strcopy` that copies a character array `s` to another `t`.

```
strcopy(s,t)
    char *s, *t; {
      while(*t++ = *s++);
}
```

We have omitted the test against '\0', because '\0' is identically zero; you will often see the code this way. (You *must* have a space after the '=': see section 25.)

For arguments to a function, and there only, the declarations

```
char s[ ];
char *s;
```

are equivalent — a pointer to a type, or an array of unspecified size of that type, are the same thing.

If this all seems mysterious, copy these forms until they become second nature. You don't often need anything more complicated.

17. Function Arguments

Look back at the function `strcopy` in the previous section. We passed it two string names as arguments, then proceeded to clobber both of them by incrementation. So how come we don't lose the original strings in the function that called `strcopy`?

As we said before, C is a "call by value" language: when you make a function call like `f(x)`, the *value* of `x` is passed, not its address. So there's no way to *alter* `x` from inside `f`. If `x` is an array (`char x[10]`) this isn't a problem, because `x` *is* an address anyway, and you're not trying to change it, just what it addresses. This is why `strcopy` works as it does. And it's convenient not to have to worry about making temporary copies of the input arguments.

But what if `x` is a scalar and you do want to change it? In that case, you have to pass the *address* of `x` to `f`, and then use it as a pointer. Thus for example, to interchange two integers, we must write

```
flip(x, y)
    int *x, *y; {
      int temp;
      temp = *x;
      *x = *y;
      *y = temp;
}
```

and to call `flip`, we have to pass the addresses of the variables:

```
flip (&a, &b);
```

18. Multiple Levels of Pointers; Program Arguments

When a C program is called, the arguments on the command line are made available to the main program as an argument count `argc` and an array of character strings `argv` containing the arguments. Manipulating these arguments is one of the most common uses of multiple levels of pointers ("pointer to pointer to ..."). By convention, `argc` is greater than zero; the first argument (in `argv[0]`) is the command name itself.

Here is a program that simply echoes its arguments.

```
main(argc, argv)
    int argc;
    char **argv; {
      int i;
      for( i=1; i < argc; i++ )
            printf("%s ", argv[i]);
      putchar('\n');
}
```

Step by step: `main` is called with two arguments, the argument count and the array of arguments. `argv` is a pointer to an array, whose individual elements are pointers to arrays of characters. The zeroth argument is the name of the command itself, so we start to print with the first argument, until we've printed them all. Each `argv[i]` is a character array, so we use a `'%s'` in the `printf`.

You will sometimes see the declaration of `argv` written as

```
char *argv[ ];
```

which is equivalent. But we can't use `char argv[][]`, because both dimensions are variable and there would be no way to figure out how big the array is.

Here's a bigger example using `argc` and `argv`. A common convention in C programs is that if the first argument is '–', it indicates a flag of some sort. For example, suppose we want a program to be callable as

```
prog –abc arg1 arg2 ...
```

where the '–' argument is optional; if it is present, it may be followed by any combination of a, b, and c.

```
main(argc, argv)
    int argc;
    char **argv; {
       . . .
       aflag = bflag = cflag  = 0;
       if( argc > 1 && argv[1][0] == '-' ) {
             for( i=1; (c=argv[1][i]) != '\0'; i++ )
                   if( c=='a' )
                         aflag++;
                   else if( c=='b' )
                         bflag++;
                   else if( c=='c' )
                         cflag++;
                   else
                         printf("%c?\n", c);
             --argc;
             ++argv;
       }
       . . .
```

There are several things worth noticing about this code. First, there is a real need for the left-to-right evaluation that && provides; we don't want to look at `argv[1]` unless we know it's there. Second, the statements

```
--argc;
++argv;
```

let us march along the argument list by one position, so we can skip over the flag argument as if it had never existed — the rest of the program is independent of whether or not there was a flag argument. This only works because `argv` is a pointer which can be incremented.

19. The Switch Statement; Break; Continue

The `switch` statement can be used to replace the multi-way test we used in the last example. When the tests are like this:

```
if( c == 'a' ) . . .
else if( c == 'b' ) . . .
else if( c == 'c' ) . . .
else . . .
```

testing a value against a series of *constants,* the switch statement is often clearer and usually gives better code. Use it like this:

```
switch( c ) {

case 'a':
      aflag++;
      break;
case 'b':
      bflag++;
      break;
case 'c':
      cflag++;
      break;
default:
      printf("%c?\n", c);
```

```
        break;
}
```

The `case` statements label the various actions we want; `default` gets done if none of the other cases are satisfied. (A `default` is optional; if it isn't there, and none of the cases match, you just fall out the bottom.)

The `break` statement in this example is new. It is there because the cases are just labels, and after you do one of them, you *fall through* to the next unless you take some explicit action to escape. This is a mixed blessing. On the positive side, you can have multiple cases on a single statement; we might want to allow both upper and lower case letters in our flag field, so we could say

```
case 'a':   case 'A':       . . .
case 'b':   case 'B':       . . .
 etc.
```

But what if we just want to get out after doing `case 'a'` ? We could get out of a `case` of the `switch` with a label and a `goto`, but this is really ugly. The `break` statement lets us exit without either `goto` or label.

```
switch( c ) {

case 'a':
      aflag++;
      break;
case 'b':
      bflag++;
      break;
   . . .
}
/* the break statements get us here directly */
```

The `break` statement also works in `for` and `while` statements — it causes an immediate exit from the loop.

The `continue` statement works *only* inside `for`'s and `while`'s; it causes the next iteration of the loop to be started. This means it goes to the increment part of the `for` and the test part of the `while`. We could have used a `continue` in our example to get on with the next iteration of the `for`, but it seems clearer to use `break` instead.

20. Structures

The main use of structures is to lump together collections of disparate variable types, so they can conveniently be treated as a unit. For example, if we were writing a compiler or assembler, we might need for each identifier information like its name (a character array), its source line number (an integer), some type information (a character, perhaps), and probably a usage count (another integer).

```
char  id[10];
int   line;
char  type;
int   usage;
```

We can make a structure out of this quite easily. We first tell C what the structure will look like, that is, what kinds of things it contains; after that we can actually reserve storage for it, either in the same statement or separately. The simplest thing is to define it and allocate storage all at once:

```
struct {
      char  id[10];
      int   line;
      char  type;
      int   usage;
```

```
} sym;
```

This defines `sym` to be a structure with the specified shape; `id, line, type` and `usage` are *members* of the structure. The way we refer to any particular member of the structure is

```
structure-name . member
```

as in

```
sym.type = 077;
if( sym.usage == 0 ) ...
while( sym.id[j++] ) ...
  etc.
```

Although the names of structure members never stand alone, they still have to be unique — there can't be another `id` or `usage` in some other structure.

So far we haven't gained much. The advantages of structures start to come when we have arrays of structures, or when we want to pass complicated data layouts between functions. Suppose we wanted to make a symbol table for up to 100 identifiers. We could extend our definitions like

```
char  id[100][10];
int   line[100];
char  type[100];
int   usage[100];
```

but a structure lets us rearrange this spread-out information so all the data about a single identifer is collected into one lump:

```
struct {
      char  id[10];
      int   line;
      char  type;
      int   usage;
} sym[100];
```

This makes `sym` an array of structures; each array element has the specified shape. Now we can refer to members as

```
sym[i].usage++;   /* increment usage of i-th identifier */
for( j=0; sym[i].id[j++] != '\0'; ) ...
  etc.
```

Thus to print a list of all identifiers that haven't been used, together with their line number,

```
for( i=0; i<nsym; i++ )
      if( sym[i].usage == 0 )
            printf("%d\t%s\n", sym[i].line, sym[i].id);
```

Suppose we now want to write a function `lookup(name)` which will tell us if `name` already exists in `sym`, by giving its index, or that it doesn't, by returning a −1. We can't pass a structure to a function directly — we have to either define it externally, or pass a pointer to it. Let's try the first way first.

```
int   nsym  0;    /* current length of symbol table */

struct {
      char  id[10];
      int   line;
      char  type;
      int   usage;
} sym[100];       /* symbol table */

main( ) {
      ...
```

```
        if( (index = lookup(newname)) >= 0 )
              sym[index].usage++;          /* already there ... */
        else
              install(newname, newline, newtype);
        ...
}
lookup(s)
    char *s; {
        int i;
        extern struct {
              char  id[10];
              int   line;
              char  type;
              int   usage;
        } sym[ ];
        for( i=0; i<nsym; i++ )
              if( compar(s, sym[i].id) > 0 )
                    return(i);
        return(-1);
}

compar(s1,s2)           /*  return 1 if s1==s2, 0 otherwise */
    char *s1, *s2; {
        while( *s1++ == *s2 )
              if( *s2++ == '\0' )
                    return(1);
        return(0);
}
```

The declaration of the structure in `lookup` isn't needed if the external definition precedes its use in the same source file, as we shall see in a moment.

Now what if we want to use pointers?

```
struct  symtag {
        char  id[10];
        int   line;
        char  type;
        int   usage;
} sym[100], *psym;

        psym = &sym[0];  /* or p = sym; */
```

This makes `psym` a pointer to our kind of structure (the symbol table), then initializes it to point to the first element of `sym`.

Notice that we added something after the word `struct`: a "tag" called `symtag`. This puts a name on our structure definition so we can refer to it later without repeating the definition. It's not necessary but useful. In fact we could have said

```
struct      symtag {
        ... structure definition
};
```

which wouldn't have assigned any storage at all, and then said

```
struct      symtag      sym[100];
struct      symtag      *psym;
```

which would define the array and the pointer. This could be condensed further, to

```
struct      symtag      sym[100], *psym;
```

The way we actually refer to an member of a structure by a pointer is like this:

```
ptr -> structure-member
```

The symbol '->' means we're pointing at a member of a structure; '->' is only used in that context. `ptr` is a pointer to the (base of) a structure that contains the structure member. The expression `ptr->structure-member` refers to the indicated member of the pointed-to structure. Thus we have constructions like:

```
psym->type = 1;
psym->id[0] = 'a';
```

and so on.

For more complicated pointer expressions, it's wise to use parentheses to make it clear who goes with what. For example,

```
struct { int x, *y; } *p;
p->x++increments x
++p->xso does this!
(++p)->x    increments p before getting x
*p->y++     uses y as a pointer, then increments it
*(p->y)++   so does this
*(p++)->y   uses y as a pointer, then increments p
```

The way to remember these is that `->`, `.` (dot), `( )` and `[ ]` bind very tightly. An expression involving one of these is treated as a unit. `p->x`, `a[i]`, `y.x` and `f(b)` are names exactly as `abc` is.

If `p` is a pointer to a structure, any arithmetic on `p` takes into account the acutal size of the structure. For instance, `p++` increments `p` by the correct amount to get the next element of the array of structures. But don't assume that the size of a structure is the sum of the sizes of its members — because of alignments of different sized objects, there may be "holes" in a structure.

Enough theory. Here is the lookup example, this time with pointers.

```
struct symtag {
      char  id[10];
      int   line;
      char  type;
      int   usage;
} sym[100];

main( ) {
      struct symtag *lookup( );
      struct symtag *psym;
      . . .
      if( (psym = lookup(newname)) )     /* non-zero pointer */
            psym -> usage++;        /* means already there */
      else
            install(newname, newline, newtype);
      . . .
}

struct symtag *lookup(s)
    char *s; {
      struct symtag *p;
      for( p=sym; p < &sym[nsym]; p++ )
            if( compar(s, p->id) > 0)
                  return(p);
```

```
        return(0);
}
```

The function `compar` doesn't change: '`p->id`' refers to a string.

In `main` we test the pointer returned by `lookup` against zero, relying on the fact that a pointer is by definition never zero when it really points at something. The other pointer manipulations are trivial.

The only complexity is the set of lines like

```
struct symtag *lookup( );
```

This brings us to an area that we will treat only hurriedly — the question of function types. So far, all of our functions have returned integers (or characters, which are much the same). What do we do when the function returns something else, like a pointer to a structure? The rule is that any function that doesn't return an `int` has to say explicitly what it does return. The type information goes before the function name (which can make the name hard to see). Examples:

```
char f(a)
    int a; {
       . . .
}

int *g( ) { . . . }

struct symtag *lookup(s) char *s; { . . . }
```

The function `f` returns a character, `g` returns a pointer to an integer, and `lookup` returns a pointer to a structure that looks like `symtag`. And if we're going to use one of these functions, we have to make a declaration where we use it, as we did in `main` above.

Notice th parallelism between the declarations

```
        struct symtag *lookup( );
        struct symtag *psym;
```

In effect, this says that `lookup( )` and `psym` are both used the same way — as a pointer to a strcture — even though one is a variable and the other is a function.

21. Initialization of Variables

An external variable may be initialized at compile time by following its name with an initializing value when it is defined. The initializing value has to be something whose value is known at compile time, like a constant.

```
int   x     0;    /* "0" could be any constant */
int   a     'a';
char  flag  0177;
int   *p    &y[1];      /* p now points to y[1] */
```

An external array can be initialized by following its name with a list of initializations enclosed in braces:

```
int   x[4]  {0,1,2,3};        /* makes x[i] = i */
int   y[ ]  {0,1,2,3};        /* makes y big enough for 4 values */
char  *msg  "syntax error\n"; /* braces unnecessary here */
char *keyword[ ]{
      "if",
      "else",
      "for",
      "while",
      "break",
      "continue",
      0
```

```
};
```

This last one is very useful — it makes `keyword` an array of pointers to character strings, with a zero at the end so we can identify the last element easily. A simple lookup routine could scan this until it either finds a match or encounters a zero keyword pointer:

```
lookup(str)       /* search for str in keyword[ ] */
    char *str; {
      int i,j,r;
      for( i=0; keyword[i] != 0; i++) {
            for( j=0; (r=keyword[i][j]) == str[j] && r != '\0'; j++ );
            if( r == str[j] )
                  return(i);
      }
      return(-1);
}
```

Sorry — neither local variables nor structures can be initialized.

22. Scope Rules: Who Knows About What

A complete C program need not be compiled all at once; the source text of the program may be kept in several files, and previously compiled routines may be loaded from libraries. How do we arrange that data gets passed from one routine to another? We have already seen how to use function arguments and values, so let us talk about external data. Warning: the words *declaration* and *definition* are used precisely in this section; don't treat them as the same thing.

A major shortcut exists for making `extern` declarations. If the definition of a variable appears *before* its use in some function, no `extern` declaration is needed within the function. Thus, if a file contains

```
f1( ) { ... }
int foo;
f2( ) { ... foo = 1; ... }
f3( ) { ... if ( foo ) ... }
```

no declaration of `foo` is needed in either `f2` or or `f3`, because the external definition of `foo` appears before them. But if `f1` wants to use `foo`, it has to contain the declaration

```
f1( ) {
      extern int foo;
      ...
}
```

This is true also of any function that exists on another file — if it wants `foo` it has to use an `extern` declaration for it. (If somewhere there is an `extern` declaration for something, there must also eventually be an external definition of it, or you'll get an "undefined symbol" message.)

There are some hidden pitfalls in external declarations and definitions if you use multiple source files. To avoid them, first, define and initialize each external variable only once in the entire set of files:

```
int   foo   0;
```

You can get away with multiple external definitions on UNIX, but not on GCOS, so don't ask for trouble. Multiple initializations are illegal everywhere. Second, at the beginning of any file that contains functions needing a variable whose definition is in some other file, put in an `extern` declaration, outside of any function:

```
extern      int   foo;
f1( ) { ... }
    etc.
```

The #include compiler control line, to be discussed shortly, lets you make a single copy of the external declarations for a program and then stick them into each of the source files making up the program.

23. #define, #include

C provides a very limited macro facility. You can say

```
#define     name        something
```

and thereafter anywhere "name" appears as a token, "something" will be substituted. This is particularly useful in parametering the sizes of arrays:

```
#define     ARRAYSIZE   100
      int   arr[ARRAYSIZE];
        . . .
      while( i++ < ARRAYSIZE ) . . .
```

(now we can alter the entire program by changing only the define) or in setting up mysterious constants:

```
#define     SET         01
#define     INTERRUPT   02    /* interrupt bit */
#define     ENABLED     04
   . . .
if( x & (SET | INTERRUPT | ENABLED) ) . . .
```

Now we have meaningful words instead of mysterious constants. (The mysterious operators '&' (AND) and '|' (OR) will be covered in the next section.) It's an excellent practice to write programs without any literal constants except in #define statements.

There are several warnings about #define. First, there's no semicolon at the end of a #define; all the text from the name to the end of the line (except for comments) is taken to be the "something". When it's put into the text, blanks are placed around it. Good style typically makes the name in the #define upper case — this makes parameters more visible. Definitions affect things only after they occur, and only within the file in which they occur. Defines can't be nested. Last, if there is a #define in a file, then the first character of the file *must* be a '#', to signal the preprocessor that definitions exist.

The other control word known to C is #include. To include one file in your source at compilation time, say

```
#include "filename"
```

This is useful for putting a lot of heavily used data definitions and #define statements at the beginning of a file to be compiled. As with #define, the first line of a file containing a #include has to begin with a '#'. And #include can't be nested — an included file can't contain another #include.

24. Bit Operators

C has several operators for logical bit-operations. For example,

```
x = x & 0177;
```

forms the bit-wise AND of x and 0177, effectively retaining only the last seven bits of x. Other operators are

|	inclusive OR
^	(circumflex) exclusive OR
~	(tilde) 1's complement
!	logical NOT
<<	left shift (as in x<<2)
>>	right shift (arithmetic on PDP-11; logical on H6070, IBM360)

25. Assignment Operators

An unusual feature of C is that the normal binary operators like '+', '–', etc. can be combined with the assignment operator '=' to form new assignment operators. For example,

```
x =- 10;
```

uses the assignment operator '=–' to decrement x by 10, and

```
x =& 0177
```

forms the AND of x and 0177. This convention is a useful notational shortcut, particularly if x is a complicated expression. The classic example is summing an array:

```
for( sum=i=0; i<n; i++ )
      sum =+ array[i];
```

But the spaces around the operator are critical! For instance,

```
x = -10;
```

sets x to –10, while

```
x =- 10;
```

subtracts 10 from x. When no space is present,

```
x=-10;
```

also decreases x by 10. This is quite contrary to the experience of most programmers. In particular, watch out for things like

```
c=*s++;
y=&x[0];
```

both of which are almost certainly not what you wanted. Newer versions of various compilers are courteous enough to warn you about the ambiguity.

Because all other operators in an expression are evaluated before the assignment operator, the order of evaluation should be watched carefully:

```
x = x<<y | z;
```

means "shift x left y places, then OR with z, and store in x.'' But

```
x =<< y | z;
```

means "shift x left by y|z places", which is rather different.

26. Floating Point

We've skipped over floating point so far, and the treatment here will be hasty. C has single and double precision numbers (where the precision depends on the machine at hand). For example,

```
double sum;
float avg, y[10];
sum = 0.0;
for( i=0; i<n; i++ )
      sum =+ y[i];
avg = sum/n;
```

forms the sum and average of the array y.

All floating arithmetic is done in double precision. Mixed mode arithmetic is legal; if an arithmetic operator in an expression has both operands int or char, the arithmetic done is integer, but if one operand is int or char and the other is float or double, both operands are converted to double. Thus if i and j are int and x is float,

```
(x+i)/j          converts i and j to float
x + i/j          does i/j integer, then converts
```

Type conversion may be made by assignment; for instance,

```
int m, n;
float x, y;
m = x;
y = n;
```

converts x to integer (truncating toward zero), and n to floating point.

Floating constants are just like those in Fortran or PL/I, except that the exponent letter is 'e' instead of 'E'. Thus:

```
pi = 3.14159;
large = 1.23456789e10;
```

printf will format floating point numbers: ''%w.df'' in the format string will print the corresponding variable in a field w digits wide, with d decimal places. An e instead of an f will produce exponential notation.

27. Horrors! goto's and labels

C has a goto statement and labels, so you can branch about the way you used to. But most of the time goto's aren't needed. (How many have we used up to this point?) The code can almost always be more clearly expressed by for/while, if/else, and compound statements.

One use of goto's with some legitimacy is in a program which contains a long loop, where a while(1) would be too extended. Then you might write

```
mainloop:
   . . .
   goto mainloop;
```

Another use is to implement a break out of more than one level of for or while. goto's can only branch to labels within the same function.

28. Acknowledgements

I am indebted to a veritable host of readers who made valuable criticisms on several drafts of this tutorial. They ranged in experience from complete beginners through several implementors of C compilers to the C language designer himself. Needless to say, this is a wide enough spectrum of opinion that no one is satisfied (including me); comments and suggestions are still welcome, so that some future version might be improved.

References

C is an extension of B, which was designed by D. M. Ritchie and K. L. Thompson [4]. The C language design and UNIX implementation are the work of D. M. Ritchie. The GCOS version was begun by A. Snyder and B. A. Barres, and completed by S. C. Johnson and M. E. Lesk. The IBM version is primarily due to T. G. Peterson, with the assistance of M. E. Lesk.

[1] D. M. Ritchie, *C Reference Manual.* Bell Labs, Jan. 1974.

[2] M. E. Lesk & B. A. Barres, *The GCOS C Library.* Bell Labs, Jan. 1974.

[3] D. M. Ritchie & K. Thompson, *UNIX Programmer's Manual.* 5th Edition, Bell Labs, 1974.

[4] S. C. Johnson & B. W. Kernighan, *The Programming Language B.* Computer Science Technical Report 8, Bell Labs, 1972.

UNIX Assembler Reference Manual

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey

0. Introduction

This document describes the usage and input syntax of the UNIX PDP-11 assembler *as*. The details of the PDP-11 are not described; consult the DEC documents "PDP-11/20 Handbook" and "PDP-11/45 Handbook."

The input syntax of the UNIX assembler is generally similar to that of the DEC assembler PAL-11R, although its internal workings and output format are unrelated. It may be useful to read the publication DEC-11-ASDB-D, which describes PAL-11R, although naturally one must use care in assuming that its rules apply to *as*.

As is a rather ordinary two-pass assembler without macro capabilities. It produces an output file which contains relocation information and a complete symbol table; thus the output is acceptable to the UNIX link-editor *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.

1. Usage

as is used as follows:

as [**—**] $file_1$ ...

If the optional "—" argument is given, all undefined symbols in the current assembly will be made undefined-external. See the **.globl** directive below.

The other arguments name files which are concatenated and assembled. Thus programs may be written in several pieces and assembled together.

The output of the assembler is placed on the file *a.out* in the current directory. If there were no unresolved external references, and no errors detected, *a.out* is made executable; otherwise, if it is produced at all, it is made non-executable.

2. Lexical conventions

Assembler tokens include identifiers (alternatively, "symbols" or "names"), temporary symbols, constants, and operators.

2.1 Identifiers

An identifier consists of a sequence of alphanumeric characters (including period "**.**", underscore "_", and tilde "~" as alphanumeric) of which the first may not be numeric. Only the first eight characters are significant. When a name begins with a tilde, the tilde is discarded and that occurrence of the identifier generates a unique entry in the symbol table which can match no other occurrence of the identifier. This feature is used by the C compiler to place names of local variables in the output symbol table without having to worry about making them unique.

2.2 Temporary symbols

A temporary symbol consists of a digit followed by "f" or "b". Temporary symbols are discussed fully in §5.1.

2.3 Constants

An octal constant consists of a sequence of digits; "8" and "9" are taken to have octal value 10 and 11. The constant is truncated to 16 bits and interpreted in two's complement notation.

A decimal constant consists of a sequence of digits terminated by a decimal point "**.**". The magnitude of the constant should be representable in 15 bits; i.e., be less than 32,768.

A single-character constant consists of a single quote "**'**" followed by an ASCII character not a new-line. Certain dual-character escape sequences are acceptable in place of the ASCII character to represent new-line and other non-graphics (see *String statements*, §5.5). The constant's value has the code for the given character in the least significant byte of the word and is null-padded on the left.

A double-character constant consists of a double quote "**"**" followed by a pair of ASCII characters not including new-line. Certain dual-character escape sequences are acceptable in place of either of the ASCII characters to represent new-line and other non-graphics (see *String statements*, §5.5). The constant's value has the code for the first given character in the least significant byte and that for the second character in the most significant byte.

2.4 Operators

There are several single- and double-character operators; see §6.

2.5 Blanks

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

2.6 Comments

The character "**/**" introduces a comment, which extends through the end of the line on which it appears. Comments are ignored by the assembler.

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 assembler begins, and it is the one into which instructions are typically placed. The UNIX 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 "—n" 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, 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 begins at the lowest 8K byte boundary after the text segment.

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. Typically the bss segment is set up by statements exemplified by

```
lab: . = .+10
```

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.

4. The location counter

One special symbol, "**.**", is the location counter. Its value at any time is the offset within the appropriate segment of 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 assign-

ment is to increase the value of "**.**", the required number of null bytes are generated (but see *Segments* above).

5. Statements

A source program is composed of a sequence of *statements*. Statements are separated either by new-lines or by semicolons. There are five kinds of statements: null statements, expression statements, assignment statements, string statements, and keyword statements.

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

5.1 Labels

There are two kinds of label: name labels and numeric labels. A name label consists of a name 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 digit *0* to *9* followed by a colon (:). Such a label serves to define temporary symbols of the form "*n*b" and "*n*f", where *n* is the digit of the label. As in the case of name labels, a numeric label assigns the current value and type of "**.**" to the temporary symbol. However, several numeric labels with the same digit may be used within the same assembly. References of the form "*n*f" refer to the first numeric label "*n*:" *f*orward from the reference; "*n*b" symbols refer to the first "*n*:" label *b*ackward from the reference. This sort of temporary label was introduced by Knuth [*The Art of Computer Programming, Vol I: Fundamental Algorithms*]. Such labels tend to conserve both the symbol table space of the assembler and the inventive powers of the programmer.

5.2 Null statements

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

5.3 Expression statements

An expression statement consists of an arithmetic expression not beginning with a keyword. The assembler computes its (16-bit) value and places it in the output stream, together with the appropriate relocation bits.

5.4 Assignment statements

An assignment statement consists of an identifier, an equals 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.

5.5 String statements

A string statement generates a sequence of bytes containing ASCII characters. A string statement consists of a left string quote "<" followed by a sequence of ASCII characters not including newline, followed by a right string quote ">". Any of the ASCII characters may be replaced by a two-character escape sequence to represent certain non-graphic characters, as follows:

\n	NL	(012)
\t	HT	(011)
\e	EOT	(004)
\0	NUL	(000)
\r	CR	(015)
\a	ACK	(006)
\p	PFX	(033)
\\	\	
\>	>	

The last two are included so that the escape character and the right string quote may be represented. The same escape sequences may also be used within single- and double-character constants (see §2.3 above).

5.6 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.

6. Expressions

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

All operators in expressions are fundamentally binary in nature; if an operand is missing on the left, a 0 of absolute type is assumed. Arithmetic is two's complement and has 16 bits of precision. All operators have equal precedence, and expressions are evaluated strictly left to right except for the effect of brackets.

6.1 Expression operators

The operators are:

(blank)	when there is no operator between operands, the effect is exactly the same as if a "+" had appeared.
+	addition
—	subtraction
*	multiplication
\/	division (note that plain " / " starts a comment)
&	bitwise **and**
|	bitwise **or**
>>	logical right shift
<<	logical left shift
%	modulo
!	*a* ! *b* is *a* **or** (**not** *b*); i.e., the **or** of the first operand and the one's complement of the second; most common use is as a unary.
^	result has the value of first operand and the type of the second; most often used to define new machine instructions with syntax identical to existing instructions.

Expressions may be grouped by use of square brackets " [ ] ". (Round parentheses are reserved for address modes.)

6.2 Types

The assembler deals with a number of types of expressions. Most types are attached to keywords and 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. It is an error to attempt to assemble an undefined expression in pass 2; in pass 1, it is not (except that certain keywords require operands which are not undefined).

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 one 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.

register
: The symbols

  r0 ... **r5**
  fr0 ... **fr5**
  sp
  pc

  are predefined as register symbols. Either they or symbols defined from them must be used to refer to the six general-purpose, six floating-point, and the 2 special-purpose machine registers. The behavior of the floating register names is identical to that of the corresponding general register names; the former are provided as a mnemonic aid.

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.

6.3 Type propagation in expressions

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, or with a register expression, the result has the register or other type. As a consequence, one can refer to r3 as ''r0+3''. If two operands of ''other type'' are combined, the result has the numerically larger type (not that this fact is very useful, since the values are not made public). 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.
^	This operator follows no other rule than that the result has the value of the first operand and the type of the second.
others	It is illegal to apply these operators to any but absolute symbols.

7. Pseudo-operations

The keywords listed below introduce statements which generate data in unusual forms or 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.

7.1 **.byte** *expression* [**,** *expression*] . . .

The *expression*s in the comma-separated list are truncated to 8 bits and assembled in successive bytes. The expressions must be absolute. This statement and the string statement above are the only ones which assemble data one byte at at time.

7.2 **.even**

If the location counter "**.**" is odd, it is advanced by one so the next statement will be assembled at a word boundary.

7.3 **.if** *expression*

The *expression* must be absolute and defined in pass 1. If its value is nonzero, the **.if** is ignored; if zero, the statements between the **.if** and the matching **.endif** (below) are ignored. **.if** may be nested. The effect of **.if** cannot extend beyond the end of the input file in which it appears. (The statements are not totally ignored, in the following sense: **.if**s and **.endif**s are scanned for, and moreover all names are entered in the symbol table. Thus names occurring only inside an **.if** will show up as undefined if the symbol table is listed.)

7.4 **.endif**

This statement marks the end of a conditionally-assembled section of code. See **.if** above.

7.5 **.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 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.

As discussed in §1, it is possible to force the assembler to make all otherwise undefined symbols external.

7.6 **.text**
7.7 **.data**
7.8 **.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.

7.9 **.comm** *name* **,** *expression*

Provided the *name* is not defined elsewhere, this statement is equivalent to

```
.globl  name
name = expression ^ name
```

That is, the type of *name* is ''undefined external'', and its value 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.

8. Machine instructions

Because of the rather complicated instruction and addressing structure of the PDP-11, the syntax of machine instruction statements is varied. Although the following sections give the syntax in detail, the 11/20 and 11/45 handbooks should be consulted on the semantics.

8.1 Sources and Destinations

The syntax of general source and destination addresses is the same. Each must have one of the following forms, where *reg* is a register symbol, and *expr* is any sort of expression:

syntax	words	mode
reg	0	0+*reg*
(*reg*) +	0	2+*reg*
— (*reg*)	0	4+*reg*
expr (*reg*)	1	6+*reg*
(*reg*)	0	1+*reg*
* *reg*	0	1+*reg*
* (*reg*) +	0	3+*reg*
* — (*reg*)	0	5+*reg*
* (*reg*)	1	7+*reg*
* *expr* (*reg*)	1	7+*reg*
expr	1	67
$ *expr*	1	27
* *expr*	1	77
* $ *expr*	1	37

The *words* column gives the number of address words generated; the *mode* column gives the octal address-mode number. The syntax of the address forms is identical to that in DEC assemblers, except that ''*'' has been substituted for ''@'' and ''$'' for ''#''; the UNIX typing conventions make ''@'' and ''#'' rather inconvenient.

Notice that mode ''*reg'' is identical to ''(reg)''; that ''*(reg)'' generates an index word (namely, 0); and that addresses consisting of an unadorned expression are assembled as pc-relative references independent of the type of the expression. To force a non-relative reference, the form ''*$expr'' can be used, but notice that further indirection is

impossible.

8.3 Simple machine instructions

The following instructions are defined as absolute symbols:

clc
clv
clz
cln
sec
sev
sez
sen

They therefore require no special syntax. The PDP-11 hardware allows more than one of the ‘‘clear’’ class, or alternatively more than one of the ‘‘set’’ class to be **or**-ed together; this may be expressed as follows:

clc | clv

8.4 Branch

The following instructions take an expression as operand. The expression must lie in the same segment as the reference, cannot be undefined-external, and its value cannot differ from the current location of ‘‘**.**’’ by more than 254 bytes:

br	**blos**	
bne	**bvc**	
beq	**bvs**	
bge	**bhis**	
blt	**bec**	(= **bcc**)
bgt	**bcc**	
ble	**blo**	
bpl	**bcs**	
bmi	**bes**	(= **bcs**)
bhi		

bes (‘‘branch on error set’’) and **bec** (‘‘branch on error clear’’) are intended to test the error bit returned by system calls (which is the c-bit).

8.5 Extended branch instructions

The following symbols are followed by an expression representing an address in the same segment as ‘‘**.**’’. If the target address is close enough, a branch-type instruction is generated; if the address is too far away, a **jmp** will be used.

jbr	**jlos**
jne	**jvc**
jeq	**jvs**
jge	**jhis**
jlt	**jec**
jgt	**jcc**
jle	**jlo**
jpl	**jcs**
jmi	**jes**
jhi	

jbr turns into a plain **jmp** if its target is too remote; the others (whose names are contructed by replacing the ‘‘b’’ in the branch instruction’s name by ‘‘j’’) turn into the converse branch over a **jmp** to the target address.

8.6 Single operand instructions

The following symbols are names of single-operand machine instructions. The form of address expected is discussed in §8.1 above.

clr	**sbcb**
clrb	**ror**
com	**rorb**
comb	**rol**
inc	**rolb**
incb	**asr**
dec	**asrb**
decb	**asl**
neg	**aslb**
negb	**jmp**
adc	**swab**
adcb	**tst**
sbc	**tstb**

8.7 Double operand instructions

The following instructions take a general source and destination (§8.1), separated by a comma, as operands.

mov
movb
cmp
cmpb
bit
bitb
bic
bicb
bis
bisb
add
sub

8.8 Miscellaneous instructions

The following instructions have more specialized syntax. Here *reg* is a register name, *src* and *dst* a general source or destination (§8.1), and *expr* is an expression:

jsr	*reg,dst*	
rts	*reg*	
sys	*expr*	
ash	*src , reg*	(or, **als**)
ashc	*src , reg*	(or, **alsc**)
mul	*src , reg*	(or, **mpy**)
div	*src , reg*	(or, **dvd**)
xor	*reg , dst*	
sxt	*dst*	
mark	*expr*	
sob	*reg , expr*	

sys is another name for the **trap** instruction. It is used to code system calls. Its operand is required to be expressible in 6 bits. The alternative forms for **ash**, **ashc**, **mul**, and **div** are provided to avoid conflict with EAE register names should they be needed.

The expression in **mark** must be expressible in six bits, and the expression in **sob** must be in the same segment as "**.**", must not be external-undefined, must be less than "**.**", and must be within 510 bytes of "**.**".

8.9 Floating-point unit instructions

The following floating-point operations are defined, with syntax as indicated:

cfcc
setf
setd

seti		
setl		
clrf	*fdst*	
negf	*fdst*	
absf	*fdst*	
tstf	*fsrc*	
movf	*fsrc, freg*	(= ldf)
movf	*freg, fdst*	(= stf)
movif	*src, freg*	(= ldcif)
movfi	*freg, dst*	(= stcfi)
movof	*fsrc, freg*	(= ldcdf)
movfo	*freg, fdst*	(= stcfd)
movie	*src, freg*	(= ldexp)
movei	*freg, dst*	(= stexp)
addf	*fsrc, freg*	
subf	*fsrc, freg*	
mulf	*fsrc, freg*	
divf	*fsrc, freg*	
cmpf	*fsrc, freg*	
modf	*fsrc, freg*	
ldfps	*src*	
stfps	*dst*	
stst	*dst*	

fsrc, *fdst*, and *freg* mean floating-point source, destination, and register respectively. Their syntax is identical to that for their non-floating counterparts, but note that only floating registers 0—3 can be a *freg*.

The names of several of the operations have been changed to bring out an analogy with certain fixed-point instructions. The only strange case is **movf**, which turns into either **stf** or **ldf** depending respectively on whether its first operand is or is not a register. Warning: **ldf** sets the floating condition codes, **stf** does not.

9. Other symbols

9.1 `..`

The symbol "`..`" is the *relocation counter*. Just before each assembled word is placed in the output stream, the current value of this symbol is added to the word if the word refers to a text, data or bss segment location. If the output word is a pc-relative address word which refers to an absolute location, the value of "`..`" is subtracted.

Thus the value of "`..`" can be taken to mean the starting core location of the program. In UNIX systems with relocation hardware, the initial value of "`..`" is 0.

The value of "`..`" may be changed by assignment. Such a course of action is sometimes necessary, but the consequences should be carefully thought out. It is particularly ticklish to change "`..`" midway in an assembly or to do so in a program which will be treated by the loader, which has its own notions of "`..`".

9.2 System calls

The following absolute symbols may be used to code calls to the UNIX system (see the **sys** instruction above).

break	**nice**
chdir	**open**
chmod	**read**
chown	**seek**
close	**setuid**
creat	**signal**
exec	**stat**
exit	**stime**
fork	**stty**
fstat	**tell**
getuid	**time**
gtty	**umount**

link	**unlink**
makdir	**wait**
mdate	**write**
mount	

Warning: the **wait** system call is not the same as the **wait** instruction, which is not defined in the assembler.

10. Diagnostics

When an input file cannot be read, its name followed by a question mark is typed and assembly ceases. When syntactic or semantic errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

)	parentheses error
]	parentheses error
>	string not terminated properly
*	indirection (*) used illegally
.	illegal assignment to "**.**"
A	error in address
B	branch address is odd or too remote
E	error in expression
F	error in local ("f" or "b") type symbol
G	garbage (unknown) character
I	end of file inside an **.if**
M	multiply defined symbol as label
O	word quantity assembled at odd address
P	phase error— "**.**" different in pass 1 and 2
R	relocation error
U	undefined symbol
X	syntax error

A Tutorial Introduction to the UNIX Text Editor

B. W. Kernighan

Bell Laboratories, Murray Hill, N. J.

Introduction

Ed is a ''text editor'', that is, an interactive program for creating and modifying ''text'', using directions provided by a user at a terminal. The text is often a document like this one, or a program or perhaps data for a program.

This introduction is meant to simplify learning *ed.* The recommended way to learn *ed* is to read this document, simultaneously using *ed* to follow the examples, then to read the description in section I of the UNIX manual, all the while experimenting with *ed.* (Solicitation of advice from experienced users is also useful.)

Do the exercises! They cover material not completely discussed in the actual text. An appendix summarizes the commands.

Disclaimer

This is an introduction and a tutorial. For this reason, no attempt is made to cover more than a part of the facilities that *ed* offers (although this fraction includes the most useful and frequently used parts). Also, there is not enough space to explain basic UNIX procedures. We will assume that you know how to log on to UNIX, and that you have at least a vague understanding of what a file is.

You must also know what character to type as the end-of-line on your particular terminal. This is a ''newline'' on Model 37 Teletypes, and ''return'' on most others. Throughout, we will refer to this character, whatever it is, as ''newline''.

Getting Started

We'll assume that you have logged in to UNIX and it has just said ''%''. The easiest way to get *ed* is to type

```
ed      (followed by a newline)
```

You are now ready to go – *ed* is waiting for you to tell it what to do.

Creating Text – the Append command "a"

As our first problem, suppose we want to create some text starting from scratch. Perhaps we are typing the very first draft of a paper; clearly it will have to start somewhere, and undergo modifications later. This section will show how to get some text in, just to get started. Later we'll talk about how to change it.

When *ed* is first started, it is rather like working with a blank piece of paper – there is no text or information present. This must be supplied by the person using *ed;* it is usually done by typing in the text, or by reading it into *ed* from a file. We will start by typing in some text, and return shortly to how to read files.

First a bit of terminology. In *ed* jargon, the text being worked on is said to be ''kept in a buffer.'' Think of the buffer as a work space, if you like, or simply as the information that you are going to be editing. In effect the buffer is like the piece of paper, on which we will write things, then change some of them, and finally file the whole thing away for another day.

The user tells *ed* what to do to his text by typing instructions called ''commands.'' Most commands consist of a single letter, which must be typed in lower case. Each command is typed on a separate line. (Sometimes the command is preceded by information about what line or lines of text are to be affected – we will discuss these shortly.) *Ed* makes no response to most commands – there is no prompting or typing of messages like ''ready''. (This silence is preferred by experienced users, but sometimes a hangup for beginners.)

The first command is *append,* written as the letter

```
a
```

all by itself. It means ''append (or add) text lines to the buffer, as I type them in.'' Appending is rather like writing fresh material on a piece of paper.

So to enter lines of text into the buffer, we just type an ''a'' followed by a newline, followed by the lines of text we want, like this:

```
a
```

```
Now is the time
for all good men
to come to the aid of their party.
.
```

The only way to stop appending is to type a line that contains only a period. The "**.**" is used to tell *ed* that we have finished appending. (Even experienced users forget that terminating "**.**" sometimes. If *ed* seems to be ignoring you, type an extra line with just "**.**" on it. You may then find you've added some garbage lines to your text, which you'll have to take out later.)

After the append command has been done, the buffer will contain the three lines

```
Now is the time
for all good men
to come to the aid of their party.
```

The "a" and "**.**" aren't there, because they are not text.

To add more text to what we already have, just issue another "a" command, and continue typing.

Error Messages – "?"

If at any time you make an error in the commands you type to *ed,* it will tell you by typing

```
?
```

This is about as cryptic as it can be, but with practice, you can usually figure out how you goofed.

Writing text out as a file – the Write command "w"

It's likely that we'll want to save our text for later use. To write out the contents of the buffer onto a file, we use the *write* command

```
w
```

followed by the filename we want to write on. This will copy the buffer's contents onto the specified file (destroying any previous information on the file). To save the text on a file named "junk", for example, type

```
w junk
```

Leave a space between "w" and the file name. *Ed* will respond by printing the number of characters it wrote out. In our case, *ed* would respond with

```
68
```

(Remember that blanks and the newline character at the end of each line are included in the character count.) Writing a file just makes a copy of the text – the buffer's contents are not disturbed, so we can go on adding lines to it. This is an important point. *Ed* at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a "w" command. (Writing out the text onto a file from time to time as it is being created is a good idea, since if the system crashes or if you make some horrible mistake, you will lose all the text in the buffer but any text that was written onto a file is relatively safe.)

Leaving ed – the Quit command "q"

To terminate a session with *ed,* save the text you're working on by writing it onto a file using the "w" command, and then type the command

```
q
```

which stands for *quit.* The system will respond with "%". At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting.

Exercise 1:

Enter *ed* and create some text using

```
a
. . . text . . .
.
```

Write it out using "w". Then leave *ed* with the "q" command, and print the file, to see that everything worked. (To print a file, say

```
pr filename
```

or

```
cat filename
```

in response to "%". Try both.)

Reading text from a file – the Edit command "e"

A common way to get text into the buffer is to read it from a file in the file system. This is what you do to edit text that you saved with the "w" command in a previous session. The *edit* command "e" fetches the entire contents of a file into the buffer. So if we had saved the three lines "Now is the time", etc., with a "w" command in an earlier session, the *ed* command

```
e junk
```

would fetch the entire contents of the file "junk" into the buffer, and respond

```
68
```

which is the number of characters in "junk". *If anything was already in the buffer, it is deleted first.*

If we use the "e" command to read a file into the buffer, then we need not use a file name after a subsequent "w" command; *ed* remembers the last file name used in an "e" command, and "w" will write on this file. Thus a common way to operate is

```
ed
e file
[editing session]
w
q
```

You can find out at any time what file name *ed* is remembering by typing the *file* command "f". In our case, if we typed

```
f
```

ed would reply

junk

Reading text from a file – the Read command "r"

Sometimes we want to read a file into the buffer without destroying anything that is already there. This is done by the *read* command "r". The command

r junk

will read the file "junk" into the buffer; it adds it to the end of whatever is already in the buffer. So if we do a read after an edit:

e junk
r junk

the buffer will contain *two* copies of the text (six lines).

Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.

Like the "w" and "e" commands, "r" prints the number of characters read in, after the reading operation is complete.

Generally speaking, "r" is much less used than "e".

Exercise 2:

Experiment with the "e" command – try reading and printing various files. You may get an error "?", typically because you spelled the file name wrong. Try alternately reading and appending to see that they work similarly. Verify that

ed filename

is exactly equivalent to

ed
e filename

What does

f filename

do?

Printing the contents of the buffer – the Print command "p"

To *print* or list the contents of the buffer (or parts of it) on the terminal, we use the print command

p

The way this is done is as follows. We specify the lines where we want printing to begin and where we want it to end, separated by a comma, and followed by the letter "p". Thus to print the first two lines of the buffer, for example, (that is, lines 1 through 2) we say

1,2p (starting line=1, ending line=2 p)

Ed will respond with

Now is the time
for all good men

Suppose we want to print *all* the lines in the buffer. We could use "1,3p" as above if we knew there were exactly 3 lines in the buffer. But in general, we don't know how many there are, so what do we use for the ending line number? *Ed* provides a shorthand symbol for "line number of last line in buffer" – the dollar sign "$". Use it this way:

1,$p

This will print *all* the lines in the buffer (line 1 to last line.) If you want to stop the printing before it is finished, push the DEL or Delete key; *ed* will type

?

and wait for the next command.

To print the *last* line of the buffer, we could use

$,$p

but *ed* lets us abbreviate this to

$p

We can print any single line by typing the line number followed by a "p". Thus

1p

produces the response

Now is the time

which is the first line of the buffer.

In fact, *ed* lets us abbreviate even further: we can print any single line by typing *just* the line number – no need to type the letter "p". So if we say

$

ed will print the last line of the buffer for us.

We can also use "$" in combinations like

$–1,$p

which prints the last two lines of the buffer. This helps when we want to see how far we got in typing.

Exercise 3:

As before, create some text using the append command and experiment with the "p" command. You will find, for example, that you can't print line 0 or a line beyond the end of the buffer, and that attempts to print a buffer in reverse order by saying

3,1p

don't work.

The current line – "Dot" or "."

Suppose our buffer still contains the six lines as above, that we have just typed

1,3p

and *ed* has printed the three lines for us. Try typing just

p (no line numbers).

This will print

to come to the aid of their party.

which is the third line of the buffer. In fact it is the last (most recent) line that we have done anything with. (We just printed it!) We can repeat this "p" command without line numbers, and it will continue to print line 3.

The reason is that *ed* maintains a record of the last line that we did anything to (in this case, line 3, which we just printed) so that it can be used instead of an explicit line number. This most recent line is referred to by the shorthand symbol

. (pronounced "dot").

Dot is a line number in the same way that "$" is; it means exactly "the current line", or loosely, "the line we most recently did something to." We can use it in several ways – one possibility is to say

.,$p

This will print all the lines from (including) the current line to the end of the buffer. In our case these are lines 3 through 6.

Some commands change the value of dot, while others do not. The print command sets dot to the number of the last line printed; by our last command, we would have "." = "$" = 6.

Dot is most useful when used in combinations like this one:

.+1 (or equivalently, .+1p)

This means "print the next line" and gives us a handy way to step slowly through a buffer. We can also say

.–1 (or .–1p)

which means "print the line *before* the current line." This enables us to go backwards if we wish. Another useful one is something like

.–3,.–1p

which prints the previous three lines.

Don't forget that all of these change the value of dot. You can find out what dot is at any time by typing

.=

Ed will respond by printing the value of dot.

Let's summarize some things about the "p" command and dot. Essentially "p" can be preceded by 0, 1, or 2 line numbers. If there is no line number given, it prints the "current line", the line that dot refers to. If there is one line number given (with or without the letter "p"), it prints that line (and dot is set there); and if there are two line numbers, it prints all the lines in that range (and sets dot to the last line printed.) If two line numbers are specified the first can't be bigger than the second (see Exercise 2.)

Typing a single newline will cause printing of the next line – it's equivalent to ".+1p". Try it. Try typing "^" – it's equivalent to ".–1p".

Deleting lines: the "d" command

Suppose we want to get rid of the three extra lines in the buffer. This is done by the *delete* command

d

Except that "d" deletes lines instead of printing them, its action is similar to that of "p". The lines to be deleted are specified for "d" exactly as they are for "p":

starting line, ending line d

Thus the command

4,$d

deletes lines 4 through the end. There are now three lines left, as we can check by using

1,$p

And notice that "$" now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to "$".

Exercise 4:

Experiment with "a", "e", "r", "w", "p", and "d" until you are sure that you know what they do, and until you understand how dot, "$", and line numbers are used.

If you are adventurous, try using line numbers with "a", "r", and "w" as well. You will find that "a" will append lines *after* the line number that you specify (rather than after dot); that "r" reads a file in *after* the line number you specify (not necessarily at the end of the buffer); and that "w" will write out exactly the lines you specify, not necessarily the whole buffer. These variations are sometimes handy. For instance you can insert a file at the beginning of a buffer by saying

0r filename

and you can enter lines at the beginning of the buffer by saying

0a
. . . *text* . . .
.

Notice that ".w" is *very* different from

.
w

Modifying text: the Substitute command "s"

We are now ready to try one of the most important of all commands – the substitute command

s

This is the command that is used to change individual words or letters within a line or group of lines. It is

what we use, for example, for correcting spelling mistakes and typing errors.

Suppose that by a typing error, line 1 says

Now is th time

– the "e" has been left off "the". We can use "s" to fix this up as follows:

1s/th/the/

This says: "in line 1, substitute for the characters 'th' the characters 'the'." To verify that it works (*ed* will not print the result automatically) we say

p

and get

Now is the time

which is what we wanted. Notice that dot must have been set to the line where the substitution took place, since the "p" command printed that line. Dot is always set this way with the "s" command.

The general way to use the substitute command is

starting–line, ending–line s/*change this*/*to this*/

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in *all* the lines between starting line and ending line. Only the first occurrence on each line is changed, however. If you want to change *every* occurrence, see Exercise 5. The rules for line numbers are the same as those for "p", except that dot is set to the last line changed. (But there is a trap for the unwary: if no substitution took place, dot is *not* changed. This causes an error "?" as a warning.)

Thus we can say

1,$s/speling/spelling/

and correct the first spelling mistake on each line in the text. (This is useful for people who are consistent misspellers!)

If no line numbers are given, the "s" command assumes we mean "make the substitution on line dot", so it changes things only on the current line. This leads to the very common sequence

s/something/something else/p

which makes some correction on the current line, and then prints it, to make sure it worked out right. If it didn't, we can try again. (Notice that we put a print command on the same line as the substitute. With few exceptions, "p" can follow any command; no other multi-command lines are legal.)

It's also legal to say

s/ . . . //

which means "change the first string of characters to *nothing*", i.e., remove them. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if we had

Nowxx is the time

we can say

s/xx//p

to get

Now is the time

Notice that "//" here means "no characters", not a blank. There *is* a difference! (See below for another meaning of "//".)

Exercise 5:

Experiment with the substitute command. See what happens if you substitute for some word on a line with several occurrences of that word. For example, do this:

a
the other side of the coin
.
s/the/on the/p

You will get

on the other side of the coin

A substitute command changes only the first occurrence of the first string. You can change all occurrences by adding a "g" (for "global") to the "s" command, like this:

s/ . . . / . . . /gp

Try other characters instead of slashes to delimit the two sets of characters in the "s" command – anything should work except blanks or tabs.

(If you get funny results using any of the characters

^ . $ [* \

read the section on "Special Characters".)

Context searching – "/ . . . /"

With the substitute command mastered, we can move on to another highly important idea of *ed* – context searching.

Suppose we have our original three line text in the buffer:

Now is the time
for all good men
to come to the aid of their party.

Suppose we want to find the line that contains "their" so we can change it to "the". Now with only three lines in the buffer, it's pretty easy to keep track of what line the word "their" is on. But if the buffer contained several hundred lines, and we'd been making changes, deleting and rearranging lines, and so on, we would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context on it.

The way we say ‘‘search for a line that contains this particular string of characters’’ is to type

/*string of characters we want to find*/

For example, the *ed* line

/their/

is a context search which is sufficient to find the desired line – it will locate the next occurrence of the characters between slashes (‘‘their’’). It also sets dot to that line and prints the line for verification:

to come to the aid of their party.

‘‘Next occurrence’’ means that *ed* starts looking for the string at line ‘‘.+1’’, searches to the end of the buffer, then continues at line 1 and searches to line dot. (That is, the search ‘‘wraps around’’ from ‘‘$’’ to 1.) It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters can’t be found in any line, *ed* types the error message

?

Otherwise it prints the line it found.

We can do both the search for the desired line *and* a substitution all at once, like this:

/their/s/their/the/p

which will yield

to come to the aid of the party.

There were three parts to that last command: context search for the desired line, make the substitution, print the line.

The expression ‘‘/their/’’ is a context search expression. In their simplest form, all context search expressions are like this – a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line, or as line numbers for some other command, like ‘‘s’’. We used them both ways in the examples above.

Suppose the buffer contains the three familiar lines

Now is the time
for all good men
to come to the aid of their party.

Then the *ed* line numbers

/Now/+1
/good/
/party/–1

are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, we could say

/Now/+1s/good/bad/

or

/good/s/good/bad/

or

/party/–1s/good/bad/

The choice is dictated only by convenience. We could print all three lines by, for instance

/Now/,/party/p

or

/Now/,/Now/+2p

or by any number of similar combinations. The first one of these might be better if we don’t know how many lines are involved. (Of course, if there were only three lines in the buffer, we’d use

1,$p

but not if there were several hundred.)

The basic rule is: a context search expression is *the same as* a line number, so it can be used wherever a line number is needed.

Exercise 6:

Experiment with context searching. Try a body of text with several occurrences of the same string of characters, and scan through it using the same context search.

Try using context searches as line numbers for the substitute, print and delete commands. (They can also be used with ‘‘r’’, ‘‘w’’, and ‘‘a’’.)

Try context searching using ‘‘?text?’’ instead of ‘‘/text/’’. This scans lines in the buffer in reverse order rather than normal. This is sometimes useful if you go too far while looking for some string of characters – it’s an easy way to back up.

(If you get funny results with any of the characters

^ . $ [* \

read the section on ‘‘Special Characters’’.)

Ed provides a shorthand for repeating a context search for the same string. For example, the *ed* line number

/string/

will find the next occurrence of ‘‘string’’. It often happens that this is not the desired line, so the search must be repeated. This can be done by typing merely

//

This shorthand stands for ‘‘the most recently used context search expression.’’ It can also be used as the first string of the substitute command, as in

/string1/s//string2/

which will find the next occurrence of ‘‘string1’’ and replace it by ‘‘string2’’. This can save a lot of typing. Similarly

??

means ‘‘scan backwards for the same expression.’’

Change and Insert – "c" and "i"

This section discusses the *change* command

c

which is used to change or replace a group of one or more lines, and the *insert* command

i

which is used for inserting a group of one or more lines.

"Change", written as

c

is used to replace a number of lines with different lines, which are typed in at the terminal. For example, to change lines ".+1" through "$" to something else, type

.+1,$c
. . . type the lines of text you want here . . .
.

The lines you type between the "c" command and the "." will take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines which have errors in them.

If only one line is specified in the "c" command, then just that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of "." to end the input – this works just like the "." in the append command and must appear by itself on a new line. If no line number is given, line dot is replaced. The value of dot is set to the last line you typed in.

"Insert" is similar to append – for instance

/string/i
. . . type the lines to be inserted here . . .
.

will insert the given text *before* the next line that contains "string". The text between "i" and "." is *inserted before* the specified line. If no line number is specified dot is used. Dot is set to the last line inserted.

Exercise 7:

"Change" is rather like a combination of delete followed by insert. Experiment to verify that

start, end d
i
. . . text . . .
.

is almost the same as

start, end c
. . . text . . .
.

These are not *precisely* the same if line "$" gets deleted. Check this out. What is dot?

Experiment with "a" and "i", to see that they are similar, but not the same. You will observe that

line–number a
. . . text . . .
.

appends *after* the given line, while

line–number i
. . . text . . .
.

inserts *before* it. Observe that if no line number is given, "i" inserts before line dot, while "a" appends after line dot.

Moving text around: the "m" command

The move command "m" is used for cutting and pasting – it lets you move a group of lines from one place to another in the buffer. Suppose we want to put the first three lines of the buffer at the end instead. We could do it by saying:

1,3w temp
$r temp
1,3d

(Do you see why?) but we can do it a lot easier with the "m" command:

1,3m$

The general case is

start line, end line m *after this line*

Notice that there is a third line to be specified – the place where the moved stuff gets put. Of course the lines to be moved can be specified by context searches; if we had

First paragraph
. . .
end of first paragraph.
Second paragraph
. . .
end of second paragraph.

we could reverse the two paragraphs like this:

/Second/,/second/m/First/–1

Notice the "–1" – the moved text goes *after* the line mentioned. Dot gets set to the last line moved.

The global commands "g" and "v"

The *global* command "g" is used to execute one or more *ed* commands on all those lines in the buffer that match some specified string. For example

g/peling/p

prints all lines that contain "peling". More usefully,

g/peling/s//pelling/gp

makes the substitution everywhere on the line, then prints each corrected line. Compare this to

1,$s/peling/pelling/gp

which only prints the last line substituted. Another subtle difference is that the ‘‘g’’ command does not give a ‘‘?’’ if ‘‘peling’’ is not found where the ‘‘s’’ command will.

There may be several commands (including ‘‘a’’, ‘‘c’’ ‘‘i’’ ‘‘r’’, ‘‘w’’, but not ‘‘g’’); in that case, every line except the last must end with a backslash ‘‘\’’:

```
g/xxx/.–1s/abc/def/\
.+2s/ghi/jkl/\
.–2,.p
```

makes changes in the lines before and after each line that contains ‘‘xxx’’, then prints all three lines.

The ‘‘v’’ command is the same as ‘‘g’’, except that the commands are executed on every line that does *not* match the string following ‘‘v’’:

```
v/ /d
```

deletes every line that does not contain a blank.

Special Characters

You may have noticed that things just don’t work right when you used some characters like ‘‘.’’, ‘‘*’’, ‘‘$’’, and others in context searches and the substitute command. The reason is rather complex, although the cure is simple. Basically, *ed* treats these characters as special, with special meanings. For instance, *in a context search or the first string of the substitute command only,*

```
/x.y/
```

means ‘‘a line with an x, *any character,* and a y,’’ *not* just ‘‘a line with an x, a period, and a y.’’ A complete list of the special characters that can cause trouble is the following:

```
^  .  $  [  *  \
```

Warning: The backslash character \ is special to *ed.* For safety’s sake, avoid it where possible. If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus

```
s/\\.\*/backslash dot star/
```

will change ‘‘\.*’’ into ‘‘backslash dot star’’.

Here is a hurried synopsis of the other special characters. First, the circumflex ‘‘ ^ ’’ signifies the beginning of a line. Thus

```
/^string/
```

finds ‘‘string’’ only if it is at the beginning of a line: it will find

```
string
```

but not

```
the string...
```

The dollar-sign ‘‘$’’ is just the opposite of the circumflex; it means the end of a line:

```
/string$/
```

will only find an occurrence of ‘‘string’’ that is at the end of some line. This implies, of course, that

```
/^string$/
```

will find only a line that contains just ‘‘string’’, and

```
/^.$/
```

finds a line containing exactly one character.

The character ‘‘.’’, as we mentioned above, matches anything;

```
/x.y/
```

matches any of

```
x+y
x–y
x y
x.y
```

This is useful in conjunction with ‘‘*’’, which is a repetition character; ‘‘a*’’ is a shorthand for ‘‘any number of a’s,’’ so ‘‘.*’’ matches any number of anythings. This is used like this:

```
s/.*/stuff/
```

which changes an entire line, or

```
s/.*,//
```

which deletes all characters in the line up to and including the last comma. (Since ‘‘.*’’ finds the longest possible match, this goes up to the last comma.)

‘‘[’’ is used with ‘‘]’’ to form ‘‘character classes’’; for example,

```
/[1234567890]/
```

matches any single digit _ any one of the characters inside the braces will cause a match.

Finally, the ‘‘&’’ is another shorthand character - it is used only on the right-hand part of a substitute command where it means ‘‘whatever was matched on the left-hand side’’. It is used to save typing. Suppose the current line contained

```
Now is the time
```

and we wanted to put parentheses around it. We could just retype the line, but this is tedious. Or we could say

```
s/^/(/
s/$/)/
```

using our knowledge of ‘‘^’’ and ‘‘$’’. But the easiest way uses the ‘‘&’’:

```
s/.*/(&)/
```

This says ‘‘match the whole line, and replace it by itself surrounded by parens.’’ The ‘‘&’’ can be used several times in a line; consider using

```
s/.*/&?  &!!/
```

to produce

```
Now is the time?  Now is the time!!
```

We don't have to match the whole line, of course: if the buffer contains

the end of the world

we could type

/world/s//& is at hand/

to produce

the end of the world is at hand

Observe this expression carefully, for it illustrates how to take advantage of *ed* to save typing. The string "/world/" found the desired line; the shorthand "//" found the same word in the line; and the "&" saved us from typing it again.

The "&" is a special character only within the replacement text of a substitute command, and has no special meaning elsewhere. We can turn off the special meaning of "&" by preceding it with a "\":

s/ampersand/\&/

will convert the word "ampersand" into the literal symbol "&" in the current line.

Summary of Commands and Line Numbers

The general form of *ed* commands is the command name, perhaps preceded by one or two line numbers, and, in the case of *e, r* and *w,* followed by a file name. Only one command is allowed per line, but a *p* command may follow any other command (except for *e, r, w* and *q).*

a (append) Add lines to the buffer (at line dot, unless a different line is specified). Appending continues until "." is typed on a new line. Dot is set to the last line appended.

c (change) Change the specified lines to the new text which follows. The new lines are terminated by a ".". If no lines are specified, replace line dot. Dot is set to last line changed.

d (delete) Delete the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line, unless "$" is deleted, in which case dot is set to "$".

e (edit) Edit new file. Any previous contents of the buffer are thrown away, so issue a *w* beforehand if you want to save them.

f (file) Print remembered filename. If a name follows *f* the remembered name will be set to it.

g (global) g/---/commands will execute the commands on those lines that contain "---", which can be any context search expression.

i (insert) Insert lines before specified line (or dot) until a "." is typed on a new line. Dot is set to last line inserted.

m (move) Move lines specified to after the line named after *m.* Dot is set to the last line moved.

p (print) Print specified lines. If none specified, print line dot. A single line number is equivalent to "line-number p". A single newline prints ".+1", the next line.

q (quit) Exit from ed. Wipes out all text in buffer!!

r (read) Read a file into buffer (at end unless specified elsewhere.) Dot set to last line read.

s (substitute) s/string1/string2/ will substitute the characters of 'string2' for 'string1' in specified lines. If no line is specified, make substitution in line dot. Dot is set to last line in which a substitution took place, which means that if no substitution took place, dot is not changed. *s* changes only the first occurrence of string1 on a line; to change all of them, type a "g" after the final slash.

v (exclude) v/---/commands executes "commands" on those lines that *do not* contain "---".

w (write) Write out buffer onto a file. Dot is not changed.

.= *(dot value)* Print value of dot. ("=" by itself prints the value of "$".)

! (temporary escape)
Execute this line as a UNIX command.

/-----/ Context search. Search for next line which contains this string of characters. Print it. Dot is set to line where string found. Search starts at ".+1", wraps around from "$" to 1, and continues to dot, if necessary.

?-----? Context search in reverse direction. Start search at ".–1", scan to 1, wrap around to "$".

UNIX for Beginners

Brian W. Kernighan

Bell Laboratories, Murray Hill, N. J.

In many ways, UNIX is the state of the art in computer operating systems. From the user's point of view, it is easy to learn and use, and presents few of the usual impediments to getting the job done.

It is hard, however, for the beginner to know where to start, and how to make the best use of the facilities available. The purpose of this introduction is to point out high spots for new users, so they can get used to the main ideas of UNIX and start making good use of it quickly.

This paper is not an attempt to re-write the *UNIX Programmer's Manual;* often the discussion of something is simply "read section x in the manual." (This implies that you will need a copy of the *UNIX Programmer's Manual.)* Rather it suggests in what order to read the manual, and it collects together things that are stated only indirectly in the manual.

There are five sections:

1. Getting Started: How to log in to a UNIX, how to type, what to do about mistakes in typing, how to log out. Some of this is dependent on which UNIX you log into (phone numbers, for example) and what terminal you use, so this section must necessarily be supplemented by local information.
2. Day-to-day Use: Things you need every day to use UNIX effectively: generally useful commands; the file system.
3. Document Preparation: Preparing manuscripts is one of the most common uses for UNIX. This section contains advice, but not extensive instructions on any of the formatting programs.
4. Writing Programs: UNIX is an excellent vehicle for developing programs. This section talks about some of the tools, but again is not a tutorial in any of the programming languages that UNIX provides.
5. A UNIX Reading List. An annotated bibliography of documents worth reading by new users.

I. GETTING STARTED

Logging In

Most of the details about logging in are in the manual section called "How to Get Started" (pages *iv-v* in the 5th Edition). Here are a couple of extra warnings.

You must have a UNIX login name, which you can get from whoever administers your system. You also need to know the phone number. UNIX is capable of dealing with a variety of terminals: Terminet 300's; Execuport, TI and similar portables; video terminals; GSI's; and even the venerable Teletype in its various forms. But note: UNIX will not handle IBM 2741 terminals and their derivatives (e.g., some Anderson-Jacobsons, Novar). Furthermore, UNIX is strongly oriented towards devices with *lower case.* If your terminal produces only upper case (e.g., model 33 Teletype), life will be so difficult that you should look for another terminal.

Be sure to set the switches appropriately on your device: speed (if it's variable) to 30 characters per second, lower case, full duplex, even parity, and any others that local wisdom advises. Establish a connection using whatever magic is needed for your terminal. UNIX should type "login:" at you. If it types garbage, you may be at the wrong speed; push the 'break' or 'interrupt' key once. If that fails to produce a login message, consult a guru.

When you get a "login:" message, type your login name *in lower case.* Follow it by a RETURN if the terminal has one. If a password is required, you will be asked for it, and (if possible) printing will be turned off while you type it, again followed by a RETURN. (On M37 Teletypes always use NEWLINE or LINEFEED in place of RETURN).

The culmination of your login efforts is a percent sign "%". The percent sign means that UNIX is ready to accept commands from the terminal. (You may also get a message of the day just before the percent sign or a notification that you have mail.)

Typing Commands

Once you've seen the percent sign, you can type commands, which are requests that UNIX do something. Try typing

```
date
```

followed by RETURN. You should get back something like

Sun Sep 22 10:52:29 EDT 1974

Don't forget the RETURN after the command, or nothing will happen. If you think you're being ignored, type a RETURN; something should happen. We won't show the carriage returns, but they have to be there.

Another command you might try is **who**, which tells you everyone who is currently logged in:

who

gives something like

```
pjp    ttyf   Sep 22 09:40
bwk    ttyg   Sep 22 09:48
mel    ttyh   Sep 22 09:58
```

The time is when the user logged in.

If you make a mistake typing the command name, UNIX will tell you. For example, if you type

whom

you will be told

whom: not found

Strange Terminal Behavior

Sometimes you can get into a state where your terminal acts strangely. For example, each letter may be typed twice, or the RETURN may not cause a line feed. You can often fix this by logging out and logging back in. Or you can read the description of the command **stty** in section I of the manual. This will also tell you how to get intelligent treatment of tab characters (which are much used in UNIX) if your terminal doesn't have tabs. If it does have computer-settable tabs, the command **tabs** will set the stops correctly for you.

Mistakes in Typing

If you make a typing mistake, and see it before the carriage return has been typed, there are two ways to recover. The sharp-character ''#'' erases the last character typed; in fact successive uses of ''#'' erase characters back to the beginning of the line (but not beyond). So if you type badly, you can correct as you go:

dd#atte##e

is the same as ''date''.

The at-sign ''@'' erases all of the characters typed so far on the current input line, so if the line is irretrievably fouled up, type an ''@'' and start over (on the same line!).

What if you must enter a sharp or at-sign as part of the text? If you precede either ''#'' or ''@'' by a backslash ''\'', it loses its erase meaning. This implies that to erase a backslash, you have to type two sharps or two at-signs. The backslash is used extensively in UNIX to indicate that the following character is in some way special.

Readahead

UNIX has full readahead, which means that you can type as fast as you want, whenever you want, even when some command is typing at you. If you type during output, your input characters will appear intermixed with the output characters, but they will be stored away by UNIX and interpreted in the correct order. So you can type two commands one after another without waiting for the first to finish or even begin.

Stopping a Program

You can stop most programs by typing the character ''DEL'' (perhaps called ''delete'' or ''rubout'' on your terminal). There are exceptions, like the text editor, where DEL stops whatever the program is doing but leaves you in that program. You can also just hang up the phone. The ''interrupt'' or ''break'' key found on most terminals has no effect.

Logging Out

The easiest way to log out is to hang up the phone. You can also type

login name–of–new–user

and let someone else use the terminal you were on. It is not sufficient just to turn off the terminal. UNIX has no time-out mechanism, so you'll be there forever unless you hang up.

Mail

When you log in, you may sometimes get the message

You have mail.

UNIX provides a postal system so you can send and receive letters from other users of the system. To read your mail, issue the command

mail

Your mail will be printed, and then you will be asked

Save?

If you do want to save the mail, type *y,* for ''yes''; any other response means ''no''.

How do you send mail to someone else? Suppose it is to go to ‘‘joe’’ (assuming ‘‘joe’’ is someone’s login name). The easiest way is this:

mail joe
now type in the text of the letter
on as many lines as you like ...
after the last line of the letter
type the character “control–d”,
that is, hold down “control” and type
a letter “d”.

And that’s it. The ‘‘control-d’’ sequence, usually called ‘‘EOT’’, is used throughout UNIX to mark the end of input from a terminal, so you might as well get used to it.

There are other ways to send mail — you can send a previously prepared letter, and you can mail to a number of people all at once. For more details see **mail** (I).

The notation **mail** (I) means the command **mail** in section (I) of the *UNIX Programmer’s Manual.*

Writing to other users

At some point in your UNIX career, out of the blue will come a message like

Message from joe...

accompanied by a startling beep. It means that Joe wants to talk to you, but unless you take explicit action you won’t be able to talk back. To respond, type the command

write joe

This establishes a two-way communication path. Now whatever Joe types on his terminal will appear on yours and vice versa. The path is slow, rather like talking to the moon. (If you are in the middle of something, you have to get to a state where you can type a command. Normally, whatever program you are running has to terminate or be terminated. If you’re editing, you can escape temporarily from the editor — read the manual.)

A protocol is needed to keep what you type from getting garbled up with what Joe types. Typically it’s like this:

Joe types ‘‘write smith’’ and waits.
Smith types ‘‘write joe’’ and waits.
Joe now types his message (as many lines as he likes). When he’s ready for a reply, he signals it by typing (o), which stands for ‘‘over’’.
Now Smith types a reply, also terminated by (o).
This cycle repeats until someone gets tired; he then signals his intent to quit with (o+o), for ‘‘over and out’’.
To terminate the conversation, each side must type a ‘‘control–d’’ character alone on a line. (‘‘Delete’’ also works.) When the other person types his ‘‘control–d’’, you will get the message “EOT” on your terminal.

If you write to someone who isn’t logged in, or who doesn’t want to be disturbed, you’ll be told. If the target is logged in but doesn’t answer after a decent interval, simply type ‘‘control-d’’.

On-line Manual

The UNIX Programmer’s Manual is typically kept on-line. If you get stuck on something, and can’t find an expert to assist you, you can print on your terminal some manual section that might help. It’s also useful for getting the most up-to-date information on a command. To print a manual section, type ‘‘man section-name’’. Thus to read up on the **who** command, type

man who

If the section in question isn’t in part I of the manual, you have to give the section number as well, as in

man 6 chess

Of course you’re out of luck if you can’t remember the section name.

II. DAY-TO-DAY USE

Creating Files — The Editor

If we have to type a paper or a letter or a program, how do we get the information stored in the machine? Most of these tasks are done with the UNIX ‘‘text editor’’ **ed**. Since **ed** is thoroughly documented in **ed** (I) and explained in *A Tutorial Introduction to the UNIX Text Editor,* we won’t spend any time here describing how to use it. All we want it for right now is to make some *files*. (A file is just a collection of information stored in the machine, a simplistic but adequate definition.)

To create a file with some text in it, do the following:

ed (invokes the text editor)
a (command to ‘‘ed’’, to add text)
now type in
whatever text you want ...
. (signals the end of adding text)

At this point we could do various editing operations on the text we typed in, such as correcting

spelling mistakes, rearranging paragraphs and the like. Finally, we write the information we have typed into a file with the editor command "w":

```
w junk
```

ed will respond with the number of characters it wrote into the file called "junk".

Suppose we now add a few more lines with "a", terminate them with ".", and write the whole thing out as "temp", using

```
w temp
```

We should now have two files, a smaller one called "junk" and a bigger one (bigger by the extra lines) called "temp". Type a "q" to quit the editor.

What files are out there?

The **ls** (for "list") command lists the names (not contents) of any of the files that UNIX knows about. If we type

```
ls
```

the response will be

```
junk
temp
```

which are indeed our two files. They are sorted into alphabetical order automatically, but other variations are possible. For example, if we add the optional argument "-t",

```
ls –t
```

lists them in the order in which they were last changed, most recent first. The "-l" option gives a "long" listing:

```
ls –l
```

will produce something like

```
–rw–rw–rw–  1 bwk   41 Sep 22 12:56 junk
–rw–rw–rw–  1 bwk   78 Sep 22 12:57 temp
```

The date and time are of the last change to the file. The 41 and 78 are the number of characters (you got the same thing from **ed**). "bwk" is the owner of the file — the person who created it. The "-rw-rw-rw-" tells who has permission to read and write the file, in this case everyone.

Options can be combined: "ls -lt" would give the same thing, but sorted into time order. You can also name the files you're interested in, and **ls** will list the information about them only. More details can be found in **ls** (I).

It is generally true of UNIX programs that "flag" arguments like "-t" precede filename arguments.

Printing Files

Now that you've got a file of text, how do you print it so people can look at it? There are a host of programs that do that, probably more than are needed.

One simple thing is to use the editor, since printing is often done just before making changes anyway. You can say

```
ed junk
1,$p
```

ed will reply with the count of the characters in "junk" and then print all the lines in the file. After you learn how to use the editor, you can be selective about the parts you print.

There are times when it's not feasible to use the editor for printing. For example, there is a limit on how big a file **ed** can handle (about 65,000 characters or 4000 lines). Secondly, it will only print one file at a time, and sometimes you want to print several, one after another. So here are a couple of alternatives.

First is **cat**, the simplest of all the printing programs. **cat** simply copies all the files in a list onto the terminal. So you can say

```
cat junk
```

or, to print two files,

```
cat junk temp
```

The two files are simply concatenated (hence the name "cat") onto the terminal.

pr produces formatted printouts of files. As with **cat**, **pr** prints all the files in a list. The difference is that it produces headings with date, time, page number and file name at the top of each page, and extra lines to skip over the fold in the paper. Thus,

```
pr junk temp
```

will list "junk" neatly, then skip to the top of a new page and list "temp" neatly.

pr will also produce multi-column output:

```
pr –3 junk
```

prints "junk" in 3-column format. You can use any reasonable number in place of "3" and **pr** will do its best.

It should be noted that **pr** is *not* a formatting program in the sense of shuffling lines around and justifying margins. The true formatters are **roff**, **nroff**, and **troff**, which we will get to in the section on document preparation.

There are also programs that print files on a high-speed printer. Look in your manual

under **opr** and **lpr**. Which to use depends on the hardware configuration of your machine.

Shuffling Files About

Now that you have some files in the file system and some experience in printing them, you can try bigger things. For example, you can move a file from one place to another (which amounts to giving a file a new name), like this:

mv junk precious

This means that what used to be ‘‘junk’’ is now ‘‘precious’’. If you do an **ls** command now, you will get

precious
temp

Beware that if you move a file to another one that already exists, the already existing contents are lost forever.

If you want to make a *copy* of a file (that is, to have two versions of something), you can use the **cp** command:

cp precious temp1

makes a duplicate copy of ‘‘precious’’ in ‘‘temp1’’.

Finally, when you get tired of creating and moving files, there is a command to remove files from the file system, called **rm**.

rm temp temp1

will remove all of the files named. You will get a warning message if one of the named files wasn’t there.

Filename, What’s in a

So far we have used filenames without ever saying what’s a legal name, so it’s time for a couple of rules. First, filenames are limited to 14 characters, which is enough to be descriptive. Second, although you can use almost any character in a filename, common sense says you should stick to ones that are visible, and that you should probably avoid characters that might be used with other meanings. We already saw, for example, that in the **ls** command, ‘‘ls -t’’ meant to list in time order. So if you had a file whose name was ‘‘-t’’, you would have a tough time listing it by name. There are a number of other characters which have special meaning either to UNIX as a whole or to numerous commands. To avoid pitfalls, you would probably do well to use only letters, numbers and the period. (Don’t use the period as the first character of a filename, for reasons too complicated to go into.)

On to some more positive suggestions. Suppose you’re typing a large document like a book. Logically this divides into many small pieces, like chapters and perhaps sections. Physically it must be divided too, for **ed** will not handle big files. Thus you should type the document as a number of files. You might have a separate file for each chapter, called

chap1
chap2
etc...

Or, if each chapter were broken into several files, you might have

chap1.1
chap1.2
chap1.3
...
chap2.1
chap2.2
...

You can now tell at a glance where a particular file fits into the whole.

There are advantages to a systematic naming convention which are not obvious to the novice UNIX user. What if you wanted to print the whole book? You could say

pr chap1.1 chap1.2 chap1.3 .....

but you would get tired pretty fast, and would probably even make mistakes. Fortunately, there is a shortcut. You can say

pr chap*

The ‘‘*’’ means ‘‘anything at all’’, so this translates into ‘‘print all files whose names begin with ‘chap’ ’’, listed in alphabetical order. This shorthand notation is not a property of the **pr** command, by the way. It is system-wide, a service of the program that interprets commands (the ‘‘shell’’ **sh** (I)). Using that fact, you can see how to list the files of the book:

ls chap*

produces

chap1.1
chap1.2
chap1.3
...

The ‘‘*’’ is not limited to the last position in a filename — it can be anywhere. Thus

rm *junk*

removes all files that contain ‘‘junk’’ as any part of their name. As a special case, ‘‘*’’ by itself matches every filename, so

```
pr *
```

prints all the files (alphabetical order), and

```
rm *
```

removes *all files*. (You had better be sure that's what you wanted to say!)

The ''*'' is not the only pattern-matching feature available. Suppose you want to print only chapters 1 through 4 and 9 of the book. Then you can say

```
pr chap[12349]*
```

The ''[...]'' means to match any of the characters inside the brackets. You can also do this with

```
pr chap[1–49]*
```

''[a-z]'' matches any character in the range *a* through *z*. There is also a ''?'' character, which matches any single character, so

```
pr ?
```

will print all files which have single-character names.

Of these niceties, ''*'' is probably the most useful, and you should get used to it. The others are frills, but worth knowing.

If you should ever have to turn off the special meaning of ''*'', ''?'', etc., enclose the entire argument in quotes (single or double), as in

```
ls "?"
```

What's in a Filename, Continued

When you first made that file called ''junk'', how did UNIX know that there wasn't another ''junk'' somewhere else, especially since the person in the next office is also reading this tutorial? The reason is that generally each user of UNIX has his own ''directory'', which contains only the files that belong to him. When you create a new file, unless you take special action, the new file is made in your own directory, and is unrelated to any other file of the same name that might exist in someone else's directory.

The set of all files that UNIX knows about are organized into a (usually big) tree, with your files located several branches up into the tree. It is possible for you to ''walk'' around this tree, and to find any file in the system, by starting at the root of the tree and walking along the right set of branches.

To begin, type

```
ls /
```

''/'' is the name of the root of the tree (a convention used by UNIX). You will get a response something like this:

```
bin
dev
etc
lib
tmp
usr
```

This is a collection of the basic directories of files that UNIX knows about. On most systems, ''usr'' is a directory that contains all the normal users of the system, like you. Now try

```
ls /usr
```

This should list a long series of names, among which is your own login name. Finally, try

```
ls /usr/your–name
```

You should get what you get from a plain

```
ls
```

Now try

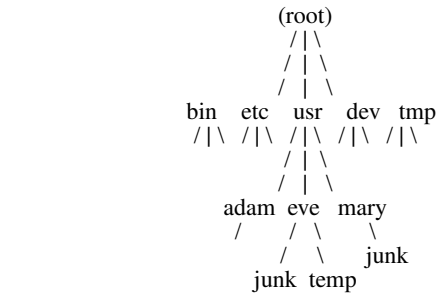
```
cat /usr/your–name/junk
```

(if ''junk'' is still around). The name

```
/usr/your–name/junk
```

is called the ''pathname'' of the file that you normally think of as ''junk''. ''Pathname'' has an obvious meaning: it represents the full name of the path you have to follow through the tree of directories to get to a particular file. It is a universal rule in UNIX that anywhere you can use an ordinary filename, you can use a pathname.

Here is a picture which may make this clearer:



Notice that Mary's ''junk'' is unrelated to Eve's.

This isn't too exciting if all the files of interest are in your own directory, but if you work with someone else or on several projects concurrently, it becomes handy indeed. For example, your friends can print your book by saying

pr /usr/your–name/chap*

Similarly, you can find out what files your neighbor has by saying

ls /usr/neighbor–name

or make your own copy of one of his files by

cp /usr/your–neighbor/his–file yourfile

(If your neighbor doesn't want you poking around in his files, or vice versa, privacy can be arranged. Each file and directory can have read-write-execute permissions for the owner, a group, and everyone else, to control access. See **ls** (I) and **chmod** (I) for details. As a matter of observed fact, most users most of the time find openness of more benefit than privacy.)

As a final experiment with pathnames, try

ls /bin /usr/bin

Do some of the names look familiar? When you run a program, by typing its name after a "%", the system simply looks for a file of that name. It looks first in your directory (where it typically doesn't find it), then in "/bin" and finally in "/usr/bin". There is nothing magic about commands like **cat** or **ls**, except that they have been collected into two places to be easy to find and administer.

What if you work regularly with someone else on common information in his directory? You could just log in as your friend each time you want to, but you can also say "I want to work on his files instead of my own". This is done by changing the directory that you are currently in:

chdir /usr/your–friend

Now when you use a filename in something like **cat** or **pr**, it refers to the file in "your-friend's" directory. Changing directories doesn't affect any permissions associated with a file — if you couldn't access a file from your own directory, changing to another directory won't alter that fact.

If you forget what directory you're in, type

pwd

("print working directory") to find out.

It is often convenient to arrange one's files so that all the files related to one thing are in a directory separate from other projects. For example, when you write your book, you might want to keep all the text in a directory called book. So make one with

mkdir book

then go to it with

chdir book

then start typing chapters. The book is now found in (presumably)

/usr/your–name/book

To delete a directory, see **rmdir** (I).

You can go up one level in the tree of files by saying

chdir ..

".." is the name of the parent of whatever directory you are currently in. For completeness, "." is an alternate name for the directory you are in.

Using Files instead of the Terminal

Most of the commands we have seen so far produce output on the terminal; some, like the editor, also take their input from the terminal. It is universal in UNIX that the terminal can be replaced by a file for either or both of input and output. As one example, you could say

ls

to get a list of files. But you can also say

ls >filelist

to get a list of your files in the file "filelist". ("filelist" will be created if it doesn't already exist, or overwritten if it does.) The symbol ">" is used throughout UNIX to mean "put the output on the following file, rather than on the terminal". Nothing is produced on the terminal. As another example, you could concatenate several files into one by capturing the output of **cat** in a file:

cat f1 f2 f3 >temp

Similarly, the symbol "<" means to take the input for a program from the following file, instead of from the terminal. Thus, you could make up a script of commonly used editing commands and put them into a file called "script". Then you can run the script on a file by saying

ed file <script

Pipes

One of the novel contributions of UNIX is the idea of a *pipe*. A pipe is simply a way to connect the output of one program to the input of another program, so the two run as a sequence of processes — a pipe-line.

For example,

```
pr f g h
```

will print the files "f", "g" and "h", beginning each on a new page. Suppose you want them run together instead. You could say

```
cat f g h >temp
pr temp
rm temp
```

but this is more work than necessary. Clearly what we want is to take the output of **cat** and connect it to the input of **pr**. So let us use a pipe:

```
cat f g h | pr
```

The vertical bar means to take the output from **cat**, which would normally have gone to the terminal, and put it into **pr**, which formats it neatly.

Any program that reads from the terminal can read from a pipe instead; any program that writes on the terminal can drive a pipe. You can have as many elements in a pipeline as you wish.

Many UNIX programs are written so that they will take their input from one or more files if file arguments are given; if no arguments are given they will read from the terminal, and thus can be used in pipelines.

The Shell

We have already mentioned once or twice the mysterious "shell," which is in fact **sh** (I). The shell is the program that interprets what you type as commands and arguments. It also looks after translating "*", etc., into lists of filenames.

The shell has other capabilities too. For example, you can start two programs with one command line by separating the commands with a semicolon; the shell recognizes the semicolon and breaks the line into two commands. Thus

```
date; who
```

does both commands before returning with a "%".

You can also have more than one program running *simultaneously* if you wish. For example, if you are doing something time-consuming, like the editor script of an earlier section, and you don't want to wait around for the results before starting something else, you can say

```
ed file <script &
```

The ampersand at the end of a command line says "start this command running, then take further commands from the terminal immediately." Thus the script will begin, but you can do something else at the same time. Of course, to keep the output from interfering with what you're doing on the terminal, it would be better to have said

```
ed file <script >lines &
```

which would save the output lines in a file called "lines".

When you initiate a command with "&", UNIX replies with a number called the process number, which identifies the command in case you later want to stop it. If you do, you can say

```
kill process–number
```

You might also read **ps** (I).

You can say

```
(command–1; command–2; command–3) &
```

to start these commands in the background, or you can start a background pipeline with

```
command–1 | command–2 &
```

Just as you can tell the editor or some similar program to take its input from a file instead of from the terminal, you can tell the shell to read a file to get commands. (Why not? The shell after all is just a program, albeit a clever one.) For instance, suppose you want to set tabs on your terminal, and find out the date and who's on the system every time you log in. Then you can put the three necessary commands (**tabs**; **date**; **who**) into a file, let's call it "xxx", and then run it with either

```
sh xxx
```

or

```
sh <xxx
```

This says to run the shell with the file "xxx" as input. The effect is as if you had typed the contents of "xxx" on the terminal. (If this is to be a regular thing, you can eliminate the need to type "sh"; see **chmod** (I) and **sh** (I).)

The shell has quite a few other capabilities as well, some of which we'll get to in the section on programming.

III. DOCUMENT PREPARATION

UNIX is extensively used for document preparation. There are three major *formatting* programs, that is, programs which produce a text with justified right margins, automatic page numbering and titling, automatic hyphenation, and the like. The simplest of these formatters is **roff**, which in fact is simple enough that if you type almost any text into a file and "roff" it, you

will get plausibly formatted output. You can do better with a little knowledge, but basically it's easy to learn and use. We'll get back to **roff** shortly.

nroff is similar to **roff** but does much less for you automatically. It will do a great deal more, once you know how to use it.

Both **roff** and **nroff** are designed to produce output on terminals, line-printers, and the like. The third formatter, **troff** (pronounced ''tee-roff''), instead drives a Graphic Systems phototypesetter, which produces very high quality output on photographic paper. This paper was printed on the phototypesetter by **troff**.

Because **nroff** and **troff** are relatively hard to learn to use effectively, several ''packages'' of canned formatting requests are available which let you do things like paragraphs, running titles, multi-column output, and so on, with little effort. Regrettably, details vary from system to system.

ROFF

The basic idea of **roff** (and of **nroff** and **troff**, for that matter) is that the text to be formatted contains within it ''formatting commands'' that indicate in detail how the formatted text is to look. For example, there might be commands that specify how long lines are, whether to use single or double spacing, and what running titles to use on each page. In general, you don't have to spell out all of the possible formatting details. Most of them have ''default values'', which you will get if you say nothing at all. For example, unless you take special precautions, you'll get single-spaced output, 65-character lines, justified right margins, and 58 text lines per page when you **roff** a file. This is the reason that **roff** is so simple — most of the decisions have already been made for you.

Some things do have to be done, however. If you want a document broken into paragraphs, you have to tell **roff** where to add the extra blank lines. This is done with the ''.sp'' command:

```
this is the end of one paragraph.
.sp
This begins the next paragraph ...
```

In **roff** (and in **nroff** and **troff**), formatting commands consist of a period followed by two letters, and they must appear at the beginning of a line, all by themselves. The ''.sp'' command tells **roff** to finish printing any of the previous line that might be still unprinted, then print a blank line before continuing. You can have more space if you wish; ''.sp 2'' asks for 2 spaces, and so on.

If you simply want to ensure that subsequent text appears on a fresh output line, you can use the command ''.br'' (for ''break'') instead of ''.sp''.

Most of the other commonly-used **roff** commands are equally simple. For example you can center one or more lines with the ''.ce'' command.

```
.ce
Title of Paper
.sp 2
```

causes the title to be centered, then followed by two blank lines. As with ''.sp'', ''.ce'' can be followed by a number; in that case, that many input lines are centered.

''.ul'' underlines lines, and can also be followed by a number:

```
.ce 2
.ul 2
An Earth–shaking Paper
.sp
John Q. Scientist
```

will center and underline the two text lines. Notice that the ''.sp'' between them is not part of the line count.

You can get multiple-line spacing instead of the default single-spacing with the ''.ls'' command:

```
.ls 2
```

causes double spacing.

If you're typing things like tables, you will not want the automatic filling-up and justification of output lines that is done by default. You can turn this off with the command ''.nf'' (no-fill), and then back on again with ''.fi'' (fill). Thus

```
this section is filled by default.
.nf
here lines will appear just
as you typed them —
no extra spaces, no moving of words.
.fi
Now go back to filling up output lines.
```

You can change the line-length with ''.ll'', and the left margin (the indent) by ''.in''. These are often used together to make offset blocks of text:

```
.ll −10
.in +10
this text will be moved 10
spaces to the right and the
lines will also be shortened 10
characters from the right. The
```

```
"+" and "−" mean to change
the previous value by that
much. Now revert:
.ll +10
.in −10
```

Notice that ".ll +10" adds ten characters to the line length, while ".ll 10" makes the line ten characters *long*.

The ".ti" command indents (in either direction) just like ".in", except for only one line. Thus to make a new paragraph with a 10-character indent, you would say

```
.sp
.ti +10
New paragraph ...
```

You can put running titles on both top and bottom of each page, like this:

```
.he "left top"center top"right top"
.fo "left bottom"center bottom"right bottom"
```

The header or footer is divided into three parts, which are marked off by any character you like. (We used a double quote.) If there's nothing between the markers, that part of the title will be blank. If you use a percent sign anywhere in ".he" or ".fo", the current page number will be inserted. So to get centered page numbers with dashes around them, at the top, use

```
.he ""− % −""
```

You can skip to the top of a new page at any time with the ".bp" command; if ".bp" is followed by a number, that will be the new page number.

The foregoing is probably enough about **roff** for you to go off and format most everyday documents. Read **roff** (I) for more details.

Hints for Preparing Documents

Most documents go through several versions (always more than you expected) before they are finally finished. Accordingly, you should do whatever possible to make the job of changing them easy.

First, when you do the purely mechanical operations of typing, type so subsequent editing will be easy. Start each sentence on a new line. Make lines short, and break lines at natural places, such as after commas and semicolons, rather than randomly. Since most people change documents by rewriting phrases and adding, deleting and rearranging sentences, these precautions simplify any editing you have to do later.

The second aspect of making change easy is not to commit yourself to formatting details too early. For example, if you decide that each paragraph is to have a space and an indent of 10 characters, you might type, before each,

```
.sp
.ti +10
```

But what happens when later you decide that it would have been better to have no space and an indent of only 5 characters? It's tedious indeed to go back and patch this up.

Fortunately, all of the formatters let you delay decisions until the actual moment of running. The secret is to define a new operation (called a *macro),* for each formatting operation you want to do, like making a new paragraph. You can say, in all three formatters,

```
.de PP
.sp
.ti +10
..
```

This *defines* ".PP" as a new **roff** (or **nroff** or **troff**) operation, whose meaning is exactly

```
.sp
.ti +10
```

(The ".." marks the end of the definition.) Whenever ".PP" is encountered in the text, it is as if you had typed the two lines of the definition in place of it.

The beauty of this scheme is that now, if you change your mind about what a paragraph should look like, you can change the formatted output merely by changing the definition of ".PP" and re-running the formatter.

As a rule of thumb, for all but the most trivial jobs, you should type a document in terms of a set of macros like ".PP", and then define them appropriately. As long as you have entered the text in some systematic way, it can always be cleaned up and re-formatted by a judicious combination of editing and macro definitions. The packages of formatting commands that we mentioned earlier are simply collections of macros designed for particular formatting tasks.

One of the main differences between **roff** and the other formatters is that macros in **roff** can only be lines of text and formatting commands. In **nroff** and **troff**, macros may have arguments, so they can have different effects depending on how they are called (in exactly the same way that the ".sp" command has an argument, the number of spaces you want).

Miscellany

In addition to the basic formatters, UNIX provides a host of supporting programs. **eqn**

and **neqn** let you integrate mathematics into the text of a document, in a language that closely resembles the way you would speak it aloud. **spell** and **typo** detect possible spelling mistakes in a document. **grep** looks for lines containing a particular text pattern (rather like the editor's context search does, but on a whole series of files). For example,

```
grep "ing$" chap*
```

will find all lines ending in the letters ''ing'' in the series of files ''chap*''. (It is almost always a good practice to put quotes around the pattern you're searching for, in case it contains characters that have a special meaning for the shell.)

wc counts the words and (optionally) lines in a set of files. **tr** translates characters into other characters; for example it will convert upper to lower case and vice versa. This translates upper into lower:

```
tr "[A–Z]" "[a–z]"
```

diff prints a list of the differences between two files, so you can compare two versions of something automatically (which certainly beats proofreading by hand). **sort** sorts files in a variety of ways; **cref** makes cross-references; **ptx** makes a permuted index (keyword-in-context listing).

Most of these programs are either independently documented (like **eqn** and **neqn**), or are sufficiently simple that the description in the *UNIX Programmer's Manual* is adequate explanation.

IV. PROGRAMMING

UNIX is a marvelously pleasant and productive system for writing programs; productivity seems to be an order of magnitude higher than on other interactive systems.

There will be no attempt made to teach any of the programming languages available on UNIX, but a few words of advice are in order. First, UNIX is written in C, as is most of the applications code. If you are undertaking anything substantial, C is the only reasonable choice. More on that in a moment. But remember that there are quite a few programs already written, some of which have substantial power.

The editor can be made to do things that would normally require special programs on other systems. For example, to list the first and last lines of each of a set of files, say a book, you could laboriously type

```
ed
e chap1.1
1p
$p
e chap1.2
1p
$p
 etc.
```

But instead you can do the job once and for all. Type

```
ls chap* >temp
```

to get the list of filenames into a file. Then edit this file to make the necessary series of editing commands (using the global commands of **ed**), and write it into ''script''. Now the command

```
ed <script
```

will produce the same output as the laborious hand typing.

The pipe mechanism lets you fabricate quite complicated operations out of spare parts already built. For example, the first draft of the **spell** program was (roughly)

```
cat ... (collect the files)
| tr ... (put each word on a new line,
        delete punctuation, etc.)
| sort  (into dictionary order)
| uniq (strip out duplicates)
| comm       (list words found in text but
        not in dictionary)
```

Programming the Shell

An option often overlooked by newcomers is that the shell is itself a programming language, and since UNIX already has a host of building-block programs, you can sometimes avoid writing a special purpose program merely by piecing together some of the building blocks with shell command files.

As an unlikely example, suppose you want to count the number of users on the machine every hour. You could type

```
date
who | wc –l
```

every hour, and write down the numbers, but that is rather primitive. The next step is probably to say

```
(date; who | wc –l) >>users
```

which uses ''>>'' to *append* to the end of the file ''users''. (We haven't mentioned ''>>'' before — it's another service of the shell.) Now all you have to do is to put a loop around this, and ensure that it's done every hour. Thus, place the following commands into a file, say ''count'':

```
: loop
(date; who | wc –l) >>users
sleep  3600
goto loop
```

The command **:** is followed by a space and a label, which you can then **goto**. Notice that it's quite legal to branch backwards. Now if you issue the command

```
sh  count  &
```

the users will be counted every hour, and you can go on with other things. (You will have to use **kill** to stop counting.)

If you would like "every hour" to be a parameter, you can arrange for that too:

```
: loop
(date; who | wc – l) >>users
sleep  $1
goto loop
```

"$1" means the first argument when this procedure is invoked. If you say

```
sh count 60
```

it will count every minute. A shell program can have up to nine arguments, "$1" through "$9".

The other aspect of programming is conditional testing. The **if** command can test conditions and execute commands accordingly. As a simple example, suppose you want to add to your login sequence something to print your mail if you have some. Thus, knowing that mail is stored in a file called 'mailbox', you could say

```
if –r mailbox  mail
```

This says "if the file 'mailbox' is readable, execute the **mail** command."

As another example, you could arrange that the "count" procedure count every hour by default, but allow an optional argument to specify a different time. Simply replace the "sleep $1" line by

```
if $1x = x sleep 3600
if $1x != x sleep $1
```

The construction

```
if $1x = x
```

tests whether "$1", the first argument, was present or absent.

More complicated conditions can be tested: you can find out the status of an executed command, and you can combine conditions with 'and', 'or', 'not' and parentheses — see **if** (I). You should also read **shift** (I) which describes how to manipulate arguments to shell command files.

Programming in C

As we said, C is the language of choice: everything in UNIX is tuned to it. It is also a remarkably easy language to use once you get started. Sections II and III of the manual describe the system interfaces, that is, how you do I/O and similar functions.

You can write quite significant C programs with the level of I/O and system interface described in *Programming in C: A Tutorial,* if you use existing programs and pipes to help. For example, rather than learning how to open and close files you can (at least temporarily) write a program that reads from its standard input, and use **cat** to concatentate several files into it. This may not be adequate for the long run, but for the early stages it's just right.

There are a number of supporting programs that go with C. The C debugger, **cdb**, is marginally useful for digging through the dead bodies of C programs. **db**, the assembly language debugger, is actually more useful most of the time, but you have to know more about the machine and system to use it well. The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

You can instrument C programs and thus find out where they spend their time and what parts are worth optimising. Compile the routines with the "-p" option; after the test run use **prof** to print an execution profile. The command **time** will give you the gross run-time statistics of a program, but it's not super accurate or reproducible.

C programs that don't depend too much on special features of UNIX can be moved to the Honeywell 6070 and IBM 370 systems with modest effort. Read *The GCOS C Library* by M. E. Lesk and B. A. Barres for details.

Miscellany

If you *have* to use Fortran, you might consider **ratfor**, which gives you the decent control structures and free-form input that characterize C, yet lets you write code that is still portable to other environments. Bear in mind that UNIX Fortran tends to produce large and relatively slow-running programs. Furthermore, supporting software like **db**, **prof**, etc., are all virtually useless with Fortran programs.

If you want to use assembly language (all heavens forfend!), try the implementation language LIL, which gives you many of the advantages of a high-level language, like decent control flow structures, but still lets you get close to the machine if you really want to.

If your application requires you to translate a language into a set of actions or another language, you are in effect building a compiler, though probably a small one. In that case, you should be using the **yacc** compiler-compiler, which helps you develop a compiler quickly.

V. UNIX READING LIST

General:

UNIX Programmer's Manual (Ken Thompson, Dennis Ritchie, and a cast of thousands). Lists commands, system routines and interfaces, file formats, and some of the maintenance procedures. You can't live without this, although you will probably only read section I.

The UNIX Time-sharing System (Ken Thompson, Dennis Ritchie). CACM, July 1974. An overview of the system, for people interested in operating systems. Worth reading by anyone who programs. Contains a remarkable number of one-sentence observations on how to do things right.

Document Preparation:

A Tutorial Introduction to the UNIX Text Editor. (Brian Kernighan). Bell Laboratories internal memorandum. Weak on the more esoteric uses of the editor, but still probably the easiest way to learn **ed**.

Typing Documents on UNIX. (Mike Lesk). Bell Laboratories internal memorandum. A macro package to isolate the novice from the vagaries of the formatting programs. If this specific package isn't available on your system, something similar probably is. This one works with both **nroff** and **troff**.

Programming:

Programming in C: A Tutorial (Brian Kernighan). Bell Laboratories internal memorandum. The easiest way to start learning C, but it's no help at all with the interface to the system beyond the simplest IO. Should be read in conjunction with

C Reference Manual (Dennis Ritchie). Bell Laboratories internal memorandum. An excellent reference, but a bit heavy going for the beginner, especially one who has never used a language like C.

Others:

D. M. Ritchie, UNIX Assembler Reference Manual.

B. W. Kernighan and L. L. Cherry, A System for Typesetting Mathematics, Computing Science Tech. Rep. 17.

M. E. Lesk and B. A. Barres, The GCOS C Library. Bell Laboratories internal memorandum.

K. Thompson and D. M. Ritchie, Setting Up UNIX.

M. D. McIlroy, UNIX Summary.

D. M. Ritchie, The UNIX I/O System.

A. D. Hall, The M6 Macro Processor, Computing Science Tech. Rep. 2.

J. F. Ossanna, NROFF User's Manual — Second Edition, Bell Laboratories internal memorandum.

D. M. Ritchie and K. Thompson, Regenerating System Software.

B. W. Kernighan, Ratfor—A Rational Fortran, Bell Laboratories internal memorandum.

M. D. McIlroy, Synthetic English Speech by Rule, Computing Science Tech. Rep. 14.

M. D. McIlroy, Bell Laboratories internal memorandum.

J. F. Ossanna, TROFF Users' Manual, Bell Laboratories internal memorandum.

B. W. Kernighan, TROFF Made Trivial, Bell Laboratories internal memorandum.

R. H. Morris and L. L. Cherry, Computer Detection of Typographical Errors, Computing Science Tech. Rep. 18.

S. C. Johnson, YACC (Yet Another Compiler-Compiler), Bell Laboratories internal memorandum.

P. J. Plauger, Programming in LIL: A Tutorial, Bell Laboratories internal memorandum.

Index

RATFOR — A Rational Fortran

B. W. Kernighan

Bell Laboratories
Murray Hill, N. J. 07974

Fortran programs are hard to read, write and debug. To make program development easier, RATFOR provides a set of decent control structures:

statement grouping
completely general **if - else** statements
while, **for** and **do** for looping
break and **next** for controlling loop exits

and some "syntactic sugar":

free form input (e.g., multiple statements/line)
unobtrusive comment convention
translation of >, >=, etc., into .GT., .GE., etc.
"define" statement for symbolic parameters
"include" statement for including source files

RATFOR not only makes programming in Fortran more enjoyable, but also allows structured programming, in the sense of coding without GOTO's. RATFOR programs tend to be markedly easier to write, read, and make correct than their equivalents in standard Fortran.

RATFOR is a preprocessor, translating the input into standard Fortran constructions. RATFOR programs may readily be written so the generated Fortran is portable; program transferability is easy to achieve. RATFOR is written in itself, so it is also portable.

The grammar of RATFOR is as follows:

```
prog    : stat
        | prog  stat
stat    : if( condition ) stat
        | if( condition ) stat else stat
        | while( condition ) stat
        | for( initialization; condition; increment ) stat
        | do do-part stat
        | break
        | next
        | digits  stat
        | { prog }
        | anything unrecognizable
```

In the grammar above, condition can be any legal Fortran condition like "'A .EQ. B'," i.e., anything that could appear in a legal Fortran logical IF statement. stat is, of course, any Fortran or RATFOR statement, or any collection of these enclosed in braces.

The while statement performs a loop while some specified condition is true. The test is performed at the *beginning* of the loop, so it is possible to do a while zero times, which can't be done with a Fortran DO.

The for statement is a somewhat generalized while statement that allows an initialization and an incrementing step as well as a termination condition on a loop. initialization is any single *Fortran* statement, which gets done once before the loop begins. increment is any single *Fortran* statement, which gets done at the end of each pass through the loop, before the test.

for and while are useful for backward loops, chaining along lists, loops that must be done zero times, and similar things which are hard to express with a DO statement, and obscure to write out directly.

The do statement is like a Fortran DO, except that no label or CONTINUE is needed. The **do-part** that follows the do keyword has to be something that can legally go into a Fortran DO statement.

A break causes an immediate exit from a for, while or do to the following statement. The next statement causes an immediate transfer to the increment part of a for or do, and the test part of a while. Both break and next refer to the innermost enclosing for, while or do.

Statements can be placed anywhere on a line; long statements are continued automatically. Multiple statements may appear on one line, if they are separated by semicolons. No semi-colon is needed at the end of a line, if RATFOR can guess whether the statement ends there. Lines ending with any characters obviously a continuation, like plus or comma, are assumed to be continued on the next line. Any statement that begins with an all-numeric field is assumed to be a Fortran label, and placed in columns 1-5. PP A '#' character in a line marks the beginning of a comment; the comment is terminated by the end of a line.

Text enclosed in matching single or double quotes is converted to nH... by RATFOR, but is otherwise unaltered. Characters like '>', '>=', and '&' are translated into their longer Fortran equivalents '.GT.', '.GE', and '.AND', except within quotes.

Any string of alphanumeric characters can be defined as a name; thereafter, whenever that name occurs in the input (delimited by non-alphanumerics) it is replaced by the rest of the definition line (comments are stripped off).

An entire source file may be included by saying ''include filename'' at the appropriate place.

YACC − Yet Another Compiler-Compiler

Stephen C. Johnson

ABSTRACT

Computer program input generally has some structure; in fact, every computer program which does input can be thought of as defining an ‘‘input language’’ which it accepts. The input languages may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, standard input facilities are restricted, difficult to use and change, and do not completely check their inputs for validity.

Yacc provides a general tool for controlling the input to a computer program. The Yacc user describes the structures of his input, together with code which is to be invoked when each such structure is recognized. Yacc turns such a specification into a subroutine which may be invoked to handle 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, if he wishes, 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 specification.

Yacc is written in C[7], and runs under UNIX. The subroutine which is output may be in C or in Ratfor[4], at the user´s choice; Ratfor permits translation of the output subroutine into portable Fortran[5]. The class of specifications accepted is a very general one, called LALR(1) grammars with disambiguating rules. The theory behind Yacc has been described elsewhere[1,2,3].

Yacc was originally designed to help produce the ‘‘front end’’ of compilers; in addition to this use, it has been successfully used in many application programs, including a phototypesetter language, a document retrieval system, a Fortran debugging system, and the Ratfor compiler.

January 6, 1998

YACC − Yet Another Compiler-Compiler

Stephen C. Johnson

Section 0: Introduction

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 which describe the input structure, code which is to be invoked when these structures are recognized, and a low-level routine to do the basic input. Yacc then produces a subroutine to do the input procedure; this subroutine, called a *parser,* calls the user-supplied low-level input routine (called the *lexical analyzer)* to pick up the basic items (called *tokens)* from the input stream. These tokens are organized according to the input structure rules, called *grammar rules;* when one of these rules has been recognized, then the user code supplied for this rule, called an *action,* is invoked; actions have the ability to return values and make use of the values of other actions.

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 quoted by 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.

As we mentioned above, an important part of the input process is carried out by the lexical analyzer. This user routine reads the true input stream, recognizing those structures which are more conveniently or more efficiently recognized directly, and communicates these recognized tokens to the parser. For historical reasons, the name of a structure recognized by the lexical analyzer is called a *terminal symbol* name, while the name of a structure recognized by the parser is called a *nonterminal symbol* name. To avoid the obvious confusion of terminology, we shall usually refer to terminal symbol names as *token names*.

There is considerable leeway in deciding whether to recognize structures by the lexical analyzer or by a grammar rule. Thus, in the above example it would be possible to have other rules of the form

```
month_name : ´J´ ´a´ ´n´ ;
month_name : ´F´ ´e´ ´b´ ;

    . . .

month_name : ´D´ ´e´ ´c´ ;
```

Here, the lexical analyzer would only need to recognize individual letters, and month_name would be a nonterminal symbol. Rules of this sort tend to be a bit wasteful of time and space, and may even restrict the power of the input process (although they are easy to write). For a more efficient input process, the lexical analyzer itself might 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 considered tokens.

As an example of the flexibility of the grammar rule approach, we might add to the above specifications the rule

date : month ´/´ day ´/´ year ;

and thus optionally allow the form

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 a very small chance of disrupting existing input.

Frequently, the input being read does not conform to the specifications due to errors in the input. The parsers produced by Yacc have the very desirable property that they will detect these input errors at the earliest place at which this can be done 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 facilities, entered as part of the input specifications, frequently permit 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 probably represent true design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. The class of specifications which Yacc can handle compares very favorably with other systems of this type; 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 basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. In Section 4, we discuss the diagnostics produced when Yacc is unable to produce a parser from the given specifications. This section also describes a simple, frequently useful mechanism for handling operator precedences. Section 5 discusses error detection and recovery. Sections 6C and 6R discuss the operating environment and special features of the subroutines which Yacc produces in C and Ratfor, respectively. Section 7 gives some hints which may lead to better designed, more efficient, and clearer specifications. Finally, Section 8 has a brief summary. Appendix A has a brief example, and Appendix B tells how to run Yacc on the UNIX operating system. Appendix C has a brief description of mechanisms and syntax which are no longer actively supported, but which are provided for historical continuity with older versions of Yacc.

Section 1: Basic Specifications

As we noted above, names refer to either tokens or nonterminal symbols. Yacc requires those names which will be used as token names to be declared as such. In addition, for reasons which will be discussed in Section 3, it is usually 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 per-cent ‘‘%’’ 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 or operator 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. Notice that 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. Notice that Yacc considers that 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'   represents newline
'\r'   represents return
'\''   represents single quote "'"
'\\'   represents backslash "\"
'\t'   represents tab
'\b'   represents backspace
'\xxx' represents "xxx" in octal
```

For a number of technical reasons, the nul character ('\0' or 000) 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 easy to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :   ;
```

As we mentioned above, names which represent tokens must be declared as such. The simplest way of doing this is to write

```
%token   name1 name2 . . .
```

in the declarations section. (See Sections 3 and 4 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. If, by the end of the rules section, some nonterminal symbol has not appeared on the left of any rule, then an error message is produced and Yacc halts.

The left hand side of the *first* grammar rule in the grammar rules section has special importance; it is taken to be the controlling nonterminal symbol for the entire input process; in technical language it is called

the *start symbol.* In effect, the parser is designed to recognize the start symbol; thus, this symbol generally represents the largest, most general structure described by the grammar rules.

The end of the input 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 subroutine returns to its caller when the endmarker is seen; we say that 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. Frequently, the endmarker token represents some reasonably obvious I/O status, such as ‘‘end-of-file’’ or ‘‘end-of-record’’.

Section 2: Actions

To each grammar rule, the user may associate an action to be performed each time the rule is recognized in the input process. This action may return a value, and may obtain the values returned by previous actions in the grammar rule. In addition, the lexical analyzer can return values for tokens, if desired.

When invoking Yacc, the user specifies a programming language; currently, Ratfor and C are supported. An action is an arbitrary statement in this language, and as such can do input and output, call subprograms, and alter external vectors and variables (recall that a ‘‘statement’’ in both C and Ratfor can be compound and do many distinct tasks). An action is specified by an equal sign ‘‘=’’ at the end of a grammar rule, followed 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 in C. A grammar rule with an action need not end with a semicolon; in fact, it is an error to have a semicolon before the equal sign.

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 integer value. For example, an action which 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 (integer) 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, we might have the rule

```
expression: ´(´ expression ´)´ ;
```

We wish the value returned by this rule to be the value of the expression in parentheses. Then we write

```
expression: ´(´ expression ´)´    = { $$ = $2 ; }
```

As a default, the value of a rule is the value of the first element in it ($1). This is true even if there is no explicit action given for the rule. Thus, grammar rules of the form

```
A: B ;
```

frequently need not have an explict action.

Notice that, although the values of actions are integers, these integers may in fact contain pointers (in C) or indices into an array (in Ratfor); in this way, actions can return and reference more complex data structures.

Sometimes, we wish to get control before a rule is fully parsed, as well as at the end of the rule. There is no explicit mechanism in Yacc to allow this; the same effect can be obtained, however, by introducing a new symbol which matches the empty string, and inserting an action for this symbol. For example, we might have a rule describing an ‘‘if’’ statement:

```
statement: IF ´(´ expr ´)´ THEN statement
```

Suppose that we wish to get control after seeing the right parenthesis in order to output some code. We might accomplish this by the rules:

```
statement:  IF ´(´ expr ´)´ actn THEN statement
        = { call action1 }

actn:   /* matches the empty string */
        = { call action2 }
```

Thus, the new nonterminal symbol actn matches no input, but serves only to call action2 after the right parenthesis is seen.

Frequently, it is more natural in such cases to break the rule into parts where the action is needed. Thus, the above example might also have been written

```
statement:  ifpart THEN statement
        = { call action1 }

ifpart:     IF ´(´ expr ´)´
        = { call action2 }
```

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 which build and maintain the tree structure desired. For example, suppose we have 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 a pointer to the newly created node. Then we can cause a parse tree to be built by supplying actions such as:

```
expr: expr ´+´ expr
        = { $$ = node( ´+´, $1, $3 ); }
```

in our specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in two places in the Yacc specification: in the declarations section, and at the head of the rules sections, before the first grammar rule. In each case, the declarations and definitions are enclosed in the marks ‘‘%{’’ and ‘‘%}’’. Declarations and definitions placed in the declarations section have global scope, and are thus known to the action statements and the lexical analyzer. Declarations and definitions placed at the head of the rules section have scope local to the action statements. Thus, in the above example, we might have included

```
%{ int variable 0; %}
```

in the declarations section, or, perhaps,

```
%{ static int variable; %}
```

at the head of the rules section. If we were writing Ratfor actions, we might want to include some COMMON statements at the beginning of the rules section, to allow for easy communication between the actions and other routines. For both C and Ratfor, Yacc has used only external names beginning in ‘‘yy’’; the user should avoid such names.

Section 3: Lexical Analysis

The user must supply a lexical analyzer which reads the input stream and communicates tokens (with values, if desired) to the parser. The lexical analyzer is an integer valued function called yylex, in both C and Ratfor. The function returns an integer which represents the type of the token. The value to be associated in the parser with that token is assigned to the integer variable yylval. Thus, a lexical analyzer written in C should begin

```
yylex ( ) {
        extern int yylval;
        . . .
```

while a lexical analyzer written in Ratfor should begin

```
integer function yylex(yylval)
        integer yylval
        . . .
```

Clearly, the parser and the lexical analyzer must agree on the type numbers in order for communication between them to take place. These numbers may be chosen by Yacc, or chosen by the user. In either case, the ‘‘define’’ mechanisms of C and Ratfor are 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 specification. The relevant portion of the lexical analyzer (in C) might look like:

```
yylex( ) {
        extern int yylval;
        int c;
        . . .
        c = getchar( );
        . . .
        if( c >= ´0´ && c <= ´9´ ) {
                yylval = c−´0´;
                return(DIGIT);
        }
        . . .
```

The relevant portion of the Ratfor lexical analyzer might look like:

```
integer function yylex(yylval)
        integer yylval, digits(10), c
        . . .
        data digits(1) / "0" /;
        data digits(2) / "1" /;
        . . .
        data digits(10) / "9" /;
        . . .
#   set c to the next input character
        . . .
        do i = 1, 10 {
                if(c .EQ. digits(i)) {
                        yylval = i−1
                        yylex = DIGIT
                        return
                }
        }
        . . .
```

In both cases, the intent is to return a token type 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, the identifier DIGIT will be redefined to be equal to the type number associated with the token name DIGIT.

This mechanism leads to clear and easily modified lexical analyzers; the only pitfall is that it makes it important to avoid using any names in the grammar which are reserved or significant in the chosen language; thus, in both C and Ratfor, the use of token names of ‘‘if’’ or ‘‘yylex’’ 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 (see Section 5).

As mentioned above, the type numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default type number for a literal character is the numerical value of the character, considered as a 1 byte integer. Other token names are assigned type numbers starting at 257. It is a difficult, machine dependent operation to determine the numerical value of an input character in Ratfor (or Fortran). Thus, the Ratfor user of Yacc will probably wish to set his own type numbers, or not use any literals in his specification.

To assign a type 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 type number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all type numbers be distinct.

There is one exception to this situation. For sticky historical reasons, the endmarker must have type number 0. Note that this is not unattractive in C, since the nul character is returned upon end of file; in Ratfor, it makes no sense. This type number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 as a type number upon reaching the end of their input.

Section 4: Ambiguity, Conflicts, and Precedence

A set of grammar rules is *ambiguous* if there is some input string which 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 we have input of the form

expr − expr − expr

the rule would permit us to treat this input either as

(expr − expr) − expr

or as

expr − (expr − expr)

(We speak of the first as *left association* of operators, and the second as *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 which it has seen:

expr − expr

matches the right side of the grammar rule above. One valid thing for the parser to do is to *reduce* the input it has seen by applying this rule; after applying the rule, it would have reduced the input it had already seen to expr (the left side of the rule). It could then read the final part of the input:

− expr

and again reduce by the rule. We see that 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 technical term is *shifting)* the input until it had seen

expr − expr − expr

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

expr − expr

Now it can reduce by the rule again; 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. We refer to this as 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.*

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 which describes which choice to make in a given situation is called a *disambiguating rule.*

Yacc has two disambiguating rules which are invoked by default, in the absence of any user directives to the contrary:

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 the proper use of reduce/reduce conflicts is still a black art, and is properly considered an advanced topic.

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. In these cases, the application of

disambiguating rules is inappropriate, and leads to a parser which is in error. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts which were 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 systems like Yacc have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural to do, 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 ;
```

Here, we consider IF and ELSE to be tokens, cond to be a nonterminal symbol describing conditional (logical) expressions, and stat to be a nonterminal symbol describing statements. In the following, we shall refer to these two rules as the *simple-if* rule and the *if-else* rule, respectively.

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 which have 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, we may *shift* the ELSE and read S2, and then reduce the right hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

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 − we have a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

Notice that 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, which is assigned a nonnegative integer. The number of states in the parser is typically two to five times the number of grammar rules.

When Yacc is invoked with the verbose (−v) option (see Appendix B), it produces a file of user output which includes a description of the states in the parser. For example, the output corresponding to the above example might be:

```
23: shift/reduce Conflict (Shift 45, Reduce 18) on ELSE

State 23

    stat : IF ( cond ) stat_
    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 state title follows, and a brief description of the grammar rules which are active in this state. The underline "_" describes the portions of the grammar rules which have been seen. Thus in the example, in state 23 we have seen input which corresponds to

IF (cond) stat

and the two grammar rules shown are active at this time. The actions possible are, if the input symbol is ELSE, we may shift into state 45. In this state, we should find as part of the description a line of the form

stat : IF (cond) stat ELSE_stat

because in this state we will have read and shifted the ELSE. 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, we should reduce by grammar rule 18, which is presumably

stat : IF ´(´ cond ´)´ stat

Notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In most states, there will be only one reduce action possible in the state, and this will always 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, a reference such as [1] might be consulted; the services of a local guru might also be appropriate.

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the area 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 which 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 which 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, which 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. An interesting situation arises when we have a unary operator and a binary operator which have the same symbolic representation, but different precedences. An example is unary and binary ´−´; frequently, unary minus is given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. We can indicate this situation by use of another keyword, %prec, to change 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 token name or literal. Thus, to make unary minus have the same precedence as multiplication, we might write:

```
%left ´+´ ´−´
%left ´*´ ´/´

%%

expr :
        expr ´+´ expr |
        expr ´−´ expr |
        expr ´*´ expr |
        expr ´/´ expr |
        ´−´ expr %prec ´*´ |
        NAME ;
```

Notice that the precedences which are described by %left, %right, and %nonassoc are independent of the declarations of token names by %token. A symbol can be declared by %token, and, later in the declarations section, be given a precedence and associativity by one of the above methods. It is true, however, that names which are given a precedence or associativity are also declared to be token names, and so in general do not need to be declared by %token, although it does not hurt to do so.

As we mentioned above, 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 which 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. Notice that 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, or both, has no precedence and associativity associated with it, 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.

There are a number of points worth making about this use of disambiguation. There is no reporting of conflicts which are resolved by this mechanism, and these conflicts are not counted in the number of shift/reduce and reduce/reduce conflicts found in the grammar. This means that occasionally mistakes in the specification of precedences 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. Frequently, not enough operators or precedences have been specified; this leads to a number of messages about shift/reduce or reduce/reduce conflicts. The cure is usually to specify more precedences, or use the %prec mechanism, or both. It is generally good to examine the verbose output file to ensure that the conflicts which are being reported can be validly resolved by precedence.

Section 5: 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 putting out any further output.

It is generally not acceptable to stop all processing when an error is found; we wish to continue scanning the input to find any further syntax errors. This leads to the problem of getting the parser ‘‘restarted’’ after an error. The general class of algorithms to do this involves reading ahead and 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 attempts to find the last time in the input when the special token ‘‘error’’ is permitted. The parser then behaves as though it saw the token name ‘‘error’’ as an input token, and attempts to parse according to the rule encountered. The token at which the error was detected remains the next input token after this error token is processed. If no special error rules have been specified, the processing effectively halts when an error is detected.

In order to prevent a cascade of error messages, the parser assumes that, after detecting an error, it 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 error message is given, and the input token is quietly deleted.

As a common example, the user might include a rule of the form

```
statement :  error ;
```

in his specification. This would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. (Notice, however, that it may be difficult or impossible to tell the end of a statement, depending on the other grammar rules). 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.

The user may supply actions after these special grammar rules, just as after the other grammar rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

The above form of grammar rule is very general, but somewhat difficult to control. Somewhat easier to deal with are rules of the form

```
statement :  error ´;´ ;
```

Here, when there is an error, the parser will again attempt to skip over the statement, but in this case will do so by skipping to the next ‘‘;’’. All tokens after the error and before the next ‘‘;’’ give syntax errors, and are discarded. When the ‘‘;’’ is seen, this rule will be reduced, and any ‘‘cleanup’’ action associated with it will be performed.

Still another form of error rule arises in interactive applications, where we may wish to prompt the user who has incorrectly input a line, and allow him to reenter the line. In C we might write:

```
inputline:     error ´\n´ prompt inputline
        = { $$ = $4; };

prompt:        /* matches no input */
        = {  printf( "Reenter last line: " ); };
```

There is one difficulty with this approach; the parser must correctly process three input tokens before it is prepared to admit that it has correctly resynchronized after the error. Thus, if the reentered line contains errors in the first two tokens, the parser will simply delete the offending tokens, and give no message; this is clearly unacceptable. For this reason, there is a mechanism in both C and Ratfor which can be used to force the parser to believe that resynchronization has taken place. One need only include a statement of the form

```
yyerrok ;
```

in his action after such a grammar rule, and the desired effect will take place; this name will be expanded, using the ‘‘# define’’ mechanism of C or the ‘‘define’’ mechanism of Ratfor, into an appropriate code sequence. For example, in the situation discussed above where we want to prompt the user to produce input, we probably want to consider that the original error has been recovered when we have thrown away the previous line, including the newline. In this case, we can reset the error state before putting out the prompt message. The grammar rule for the nonterminal symbol prompt becomes:

```
prompt:     /* matches no input */
        = {
                yyerrok;
                printf( "Reenter last line: " );
        } ;
```

There is another special feature which the user may wish to use in error recovery. As mentioned above, the token seen immediately after the ‘‘error’’ symbol is the input token at which the error was discovered. Sometimes, this is seen to be 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 user wishes a way of clearing the previous input token held in the parser. One need only include a statement of the form

```
yyclearin ;
```

in his action; again, this expands, in both C and Ratfor, to the appropriate code sequence. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, which 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; we wish to throw away the old, illegal token, and reset the error state. We might do this by the sequence:

```
statement :  error
        = {
                resynch( );
                yyerrok ;
                yyclearin ;
        } ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors, and have the virtue that the user can get ‘‘handles’’ by which he can deal with the error actions required by the lexical and output portions of the system.

Section 6C: The C Language Yacc Environment

The default mode of operation in Yacc is to write actions and the lexical analyzer in C. This has a number of advantages; primarily, it is easier to write character handling routines, such as the lexical analyzer, in a language which supports character-by-character I/O, and has shifting and masking operators.

When the user inputs a specification to Yacc, the output is a file of C programs, called ‘‘y.tab.c’’. These are then compiled, and loaded with a library; the library has default versions of a number of useful routines. This section discusses these routines, and how the user can write his own routines if desired. The name of the Yacc library is system dependent; see Appendix B.

The subroutine 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 (type number 0), and the parser accepts. In this case, yyparse returns the value 0.

Three of the routines on the Yacc library are concerned with the ‘‘external’’ environment of yyparse. There is a default ‘‘main’’ program, a default ‘‘initialization’’ routine, and a default ‘‘accept’’ routine, respectively. They are so simple that they will be given here in their entirety:

```
main( argc, argv )
int argc;
char *argv[ ]
{
        yyinit( argc, argv );
        if( yyparse( ) )
                return;
        yyaccpt( );
}

yyinit( ) { }

yyaccpt( ) { }
```

By supplying his own versions of yyinit and/or yyaccpt, the user can get control either before the parser is called (to set options, open input files, etc.) or after the accept action has been done (to close files, call the next pass of the compiler, etc.). Note that yyinit is called with the two ‘‘command line’’ arguments which have been passed into the main program. If neither of these routines is redefined, the default situation simply looks like a call to the parser, followed by the termination of the program. Of course, in many cases the user will wish to supply his own main program; for example, this is necessary if the parser is to be called more than once.

The other major routine on the library is called ‘‘yyerror’’; its main purpose is to write out a message when a syntax error is detected. It has a number of hooks and handles which attempt to make this error message general and easy to understand. This routine is somewhat more complex, but still approachable:

```
extern int yyline;  /* input line number */

yyerror(s)
char *s;
{
        extern int yychar;
        extern char *yysterm[ ];

        printf("\n%s", s );
        if( yyline )
                printf(", line %d,", yyline );
        printf(" on input: ");
        if( yychar >= 0400 )
                printf("%s\n", yysterm[yychar−0400] );
        else switch ( yychar ) {
        case ´\t´: printf( "\\t\n" ); return;
        case ´\n´: printf( "\\n\n" ); return;
        case ´\0´: printf( "$end\n" ); return;
        default: printf( "%c\n" , yychar ); return;
        }
}
```

The argument to yyerror is a string containing an error message; most usually, it is ‘‘syntax error’’. yyerror also uses the external variables yyline, yychar, and yysterm. yyline is a line number which, if set by the user to a nonzero number, will be printed out as part of the error message. yychar is a variable which contains the type number of the current token. yysterm has the names, supplied by the user, for all the tokens which have names. Thus, the routine spends most of its time trying to print out a reasonable name for the input token. The biggest problem with the routine as given is that, on Unix, the error message does not go out on the error file (file 2). This is hard to arrange in such a way that it works with both the portable I/O library and the system I/O library; if a way can be worked out, the routine will be changed to do this.

Beware: This routine will not work if any token names have been given redefined type numbers. In this case, the user must supply his own yyerror routine. Hopefully, this ‘‘feature’’ will disappear soon.

Finally, there is another feature which the C user of Yacc might wish to use. The integer variable yydebug is normally set to 0. If it is set to 1, 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.

Section 6R: The Ratfor Language Yacc Environment

For reasons of portability or compatibility with existing software, it may be desired to use Yacc to generate parsers in Ratfor, or, by extension, in portable Fortran. The user is likely to work considerably harder doing this than he might if he were to use C.

When the user inputs a specification to Yacc, and specifies the Ratfor option (see Appendix B), the output is a file of Ratfor programs called ‘‘y.tab.r’’. These programs are then compiled, and provide the desired subroutine.

The subroutine produced by Yacc which does the input process is an integer function called ‘‘yypars’’. When it is called, it in turn repeatedly calls ‘‘yylex’’, the lexical analyzer supplied by the user (see Section 3). Eventually, either an error is detected, in which case (if no error recovery is possible) yypars returns the value 1, or the lexical analyzer returns the endmarker (type number 0), and the parser accepts. In this case, yypars returns 0.

Unlike the C program situation (see Section 6C) there is no library of Ratfor routines which must be used in the loading process. As a side effect of this, *the user must supply a main program which calls yypars*. A suggested Ratfor main program is

```
integer yypars
n = yypars(0)
if( n .EQ. 0 ) {
        . . . here if the program accepted
} else {
        . . . here if there were unrecoverable errors
}
end
```

Notice that there is no easy way for the user to get control when an error is detected, since the Fortran language provides only a very crude character string capability.

There is another feature which the Ratfor user might wish to use. The argument to yypars is normally 0. If it is set to 1, 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. During the input process, the value of this debug flag is kept in a common variable yydebu, which is available to the actions and may be set and reset at will.

Statement labels 1 through 1000 are reserved for the parser, and may not appear in actions; note that, because Ratfor has a more modern control structure than Fortran, it is rarely necessary to use statement labels at all; the most frequent use of labels in Ratfor is in formatted I/O.

Because Fortran has no standard character set and not even a standard character width, it is difficult to produce a lexical analyzer in portable Fortran The usual solution is to provide a routine which does a table search to get the internal type number for each input character, with the understanding that such a routine can be recoded to run far faster for any particular machine.

Finally, we must warn the user that the Ratfor feature of Yacc has been operational for a much shorter time than the other portions of the system. If past experience is any guide, the Ratfor support will develop and become more powerful and better human engineered in response to user complaints and requirements. Thus, the potential Ratfor user might do well to contact the author to discuss his own particular needs.

Section 7. 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; the reader seeing Yacc for the first time may well find that this entire section could be omitted.

Input Style

It is difficult to input rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan, and are officially endorsed by the author.

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. Indent rule bodies by one tab stop, and action bodies by two tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (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.

Common Actions

When several grammar rules have the same action, the user might well wish to provide only one code sequence. A simple, general mechanism is, of course, to use subroutine calls. It is also possible to put a label on the first statement of an action, and let other actions be simply a goto to this label. Thus, if the user had a routine which built trees, he might wish to have only one call to it, as follows:

```
expr :
        expr ´+´ expr =
        {  binary:
                $$ = btree( $1, $2, $3 );
        } |
        expr ´−´ expr =
        {
                goto binary;
        } |
        expr ´*´ expr =
        {
                goto binary;
        } ;
```

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

```
sequence :
        item |
        sequence item ;
```

Notice that, 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.

If the user were to write these rules right recursively, such as

```
sequence :
        item |
        item sequence ;
```

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.

The user should also consider whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
sequence :
        | /* empty */
        sequence 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. Experience suggests that permitting empty sequences leads to increased generality, which frequently is not evident at the time the rule is first written. There are cases, however, when the Yacc algorithm can fail when such a change is made. In effect, conflicts might arise when Yacc is asked to decide which empty sequence it has seen, when it hasn´t seen enough to know! Nevertheless, this principle is still worth following wherever possible.

Lexical Tie-ins

Frequently, there are lexical decisions which depend on the presence of various constructions in the specification. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling these situations is to create a global flag which is examined by the lexical analyzer, and set by actions. For example, consider a situation where we have a program which consists of 0 or more declarations, followed by 0 or more statements. We declare a flag called ‘‘dflag’’, which is 1 during declarations, and 0 during statements. We may do this as follows:

```
%{
        int dflag ;
%}
%%
program :
        decls  stats ;

decls :
        = /* empty */
        {
                dflag = 1;
        } |
        decls declaration ;

stats :
        = /* empty */
        {
                dflag = 0;
        } |
        stats statement ;

        . . .  other rules . . .
```

The flag dflag is now set to zero when reading statements, and 1 when reading declarations, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. Frequently, however, this single token exception does not affect the lexical scan required.

Clearly, this kind of ‘‘backdoor’’ approach can be elaborated on to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Bundling

Bundling is a technique for collecting together various character strings so that they can be output at some later time. It is derived from a feature of the same name in the compiler/compiler TMG [6].

Bundling has two components − a nice user interface, and a clever implementation trick. They will be discussed in that order.

The user interface consists of two routines, ‘‘bundle’’ and ‘‘bprint’’.

```
bundle( a1, a2, . . ., an )
```

accepts a variable number of arguments which are either character strings or bundles, and returns a bundle, whose value will be the concatenation of the values of a1, . . ., an.

```
bprint( b )
```

accepts a bundle as argument and outputs its value.

For example, suppose that we wish to read arithmetic expressions, and output function calls to routines called ‘‘add’’, ‘‘sub’’, ‘‘mul’’, ‘‘div’’, and ‘‘assign’’. Thus, we wish to translate

```
a = b − c*d
```

into

```
assign(a,sub(b,mul(c,d)))
```

A Yacc specification file which does this is given in Appendix D; this includes an implementation of the bundle and bprint routines. A rule and action of the form

```
expr:
        expr ´+´ expr =
        {
                $$ = bundle( "add(", $1, ",", $3, ")" );
        }
```

causes the returned value of expr to be come a bundle, whose value is the character string containing the desired function call. Each NAME token has a value which is a pointer to the actual name which has been read. Finally, when the entire input line has been read and the value has been bundled, the value is written out and the bundles and names are cleared, in preparation for the next input line.

Bundles are implemented as arrays of pointers, terminated by a zero pointer. Each pointer either points to a bundle or to a character string. There is an array, called *bundle space,* which contains all the bundles.

The implementation trick is to check the values of the pointers in bundles − if the pointer points into bundle space, it is assumed to point to a bundle; otherwise it is assumed to point to a character string.

The treatment of functions with a variable number of arguments, like bundle, is likely to differ from one implementation of C to another.

In general, one may wish to have a simple storage allocator which allocates and frees bundles, in order to handle situations where it is not appropriate to completely clear all of bundle space at one time.

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, since it is difficult to pass the required information to the lexical analyzer which tells 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, and one will probably be supported eventually. Until this day comes, I suggest that the keywords be *reserved;* that is, be forbidden for use as variable names. There are powerful stylistic reasons for preferring this, anyway (he said weakly . . .).

Non-integer Values

Frequently, the user wishes to have values which are bigger than integers; again, this is an area where Yacc does not make the job as easy as it might, and some additional support is likely. Nevertheless, at the cost of writing a storage manager, the user can return pointers or indices to blocks of storage big enough to contain the full values desired.

Previous Work

There have been many previous applications of Yacc. The user who is contemplating a big application might well find that others have developed relevant techniques, or even portions of grammars. Yacc specifications appear to be easier to change than the equivalent computer programs, so that the ‘‘prior art’’ is more relevant here as well.

Section 8: User Experience, Summary, and Acknowledgements

Yacc has been used in the construction of a C compiler for the Honeywell 6000, a system for typesetting mathematical equations, a low level implementation language for the PDP 11, APL and Basic compilers to run under the UNIX system, and a number of other applications.

To summarize, Yacc can be used to construct parsers; these parsers can interact in a fairly flexible way with the lexical analysis and output phases of a larger system. The system also provides an indication of ambiguities in the specification, and allows disambiguating rules to be supplied to resolve these ambiguities.

Because the output of Yacc is largely tables, the system is relatively language independent. In the presence of reasonable applications, Yacc could be modified or adapted to produce subroutines for other machines and languages. In addition, we continue to seek better algorithms to improve the lexical analysis and code generation phases of compilers produced using Yacc.

This document would be incomplete if I did not give credit to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for ‘‘one more feature’’. Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of Yacc. Al Aho also deserves recognition for bringing the mountain to Mohammed, and other favors.

References

1 Aho, A.V. and Johnson, S.C., ‘‘LR Parsing’’, Computing Surveys, Vol 6, No 2, June 1974, pp. 99-124.

2 Aho, A.V., Johnson, S.C., and Ullman, J.D., ‘‘Deterministic Parsing of Ambiguous Grammars’’, Proceedings of the A.C.M. Symposium on Principles of Programming Languages, October 1973, pp. 1-21; to appear in CACM.

3 Aho, A.V. and Ullman, J.D., Theory of Parsing, Translation, and Compiling. Volume 1 (1972) and Volume 2 (1973), Prentice-Hall, Englewood Cliffs, N.J.

4 Kernighan, B. W., Ratfor, a Rational Fortran

5 Ryder, B. B., ‘‘The PFORT Verifier,’’ Software−Practice and Experience, Vol 4 (1974), pp 359-377.

6 McIlroy, M. D., A Manual for the TMG Compiler-writing Language

7 Ritchie, D. M., C Reference Manual

Appendix A: 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 is an assignment at the top level, the value is not printed; otherwise it is. As in C, an integer which 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 the way that precedences and ambiguities are used, as well as showing how simple error recovery operates. 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; frequently, this job is better done by the lexical analyzer.

```
%token DIGIT LETTER/* these are token names */
%left ´|´                       /* declarations of operator precedences */
%left ´&´
%left ´+´ ´−´
%left ´*´ ´/´ ´%´
%left UMINUS                    /* supplies precedence for unary minus */
%{                              /* declarations used by the actions */
        int base;
        int regs[26];
%}

%%   /* beginning of rules section */

list :          /* list is the start symbol */
        |       /* 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 =
        {
                $$ = $1 ;
                base = 10 ;
                if( $1 == 0 )
                        base = 8 ;
        } |
        number DIGIT =
        {
                $$ = base * $1 + $2 ;
        } ;

%%              /* start of programs */

yylex( ) /* lexaical 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( )) == ´ ´ )
                ;
        if( c >= ´a´ && c <= ´z´ ) {
                yylval = c − ´a´ ;
                return( LETTER ) ;
        }
        if( c >= ´0´ && c <= ´9´ ) {
                yylval = c − ´0´ ;
                return( DIGIT ) ;
        }
        return( c ) ;
}
```

Appendix B: Use of Yacc on Unix

Suppose that the Yacc specification is on a file called yfile. If the actions are in C, Yacc is invoked by

```
yacc yfile
```

The output appears on file y.tab.c To compile the parser and load it with the Yacc library, use the command

```
cc y.tab.c −ly
```

If Yacc is invoked with the option −v:

```
yacc −v yfile
```

a verbose description of the parser is produced on file y.output. The C user should consult section 6C for more information about the run time environment.

If the actions are in Ratfor, the user should invoke Yacc with the option −r:

```
yacc −r yfile
```

The Ratfor output appears on file y.tab.r It may be compiled by

```
rc −2 y.tab.r
```

Note that when Yacc is used to produce Ratfor programs, there is no need to load these programs with any library.

If the −v action is also invoked:

```
yacc −rv yfile
```

a verbose description of the parser is produced on file y.output. The Ratfor user should consult section 6R for more information about the run time environment.

Appendix C: Old Features Supported but not Encouraged

This appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may be delimited by double quotes "\"" as well as single 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 which 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. The curly braces "{" and "}" around an action are optional if the action consists of a single C statement. (They are always required in Ratfor).

Appendix D: An Example of Bundling

The following program is an example of the technique of bundling; this example is discussed in Section 7.

```
/* warnings:
1. This works on Unix; the handling of functions with a variable number of arguments is different on
   different systems.
2. A number of checks for array bounds have been left out to avoid obscuring the basic ideas, but
   should be there in a practical program.
    */


%token NAME

%right ´=´
%left ´+´ ´−´
%left ´*´ ´/´

%%

lines :
        = /* empty */
        {
                bclear( ) ;
        } |
        lines expr ´\n´ =
        {
                bprint( $2 ) ;
                printf( "\n" ) ;
                bclear( ) ;
        } |
        lines error ´\n´ =
        {
                bclear( ) ;
                yyerrok;
        } ;

expr :
        expr ´+´ expr =
        {
                $$ = bundle( "add(", $1, ",", $3, ")" );
        } |
        expr ´−´ expr =
        {
                $$ = bundle( "sub(", $1, ",", $3, ")" );
        } |
        expr ´*´ expr =
        {
                $$ = bundle( "mul(", $1, ",", $3, ")" );
        } |
        expr ´/´ expr =
```

```
        {
                $$ = bundle( "div(", $1, ",", $3, ")" );
        } |
        ´(´ expr ´)´ =
        {
                $$ = $2;
        } |
        NAME ´=´ expr =
                $$ = bundle( "assign(", $1, ",", $3, ")" );
        } |
        NAME ;

%%

#define        nsize 200
char   names[nsize], *nptr { names };

#define        bsize 500
int    bspace[bsize], *bptr { bspace };

yylex( )
{
        int c;

        c = getchar( );
        while( c == ´ ´ )
                c = getchar( );
        if( c>=´a´ && c<=´z´ ) {
                yylval = nptr;
                for( ; c>=´a´ && c<=´z´; c=getchar( ) )
                        *nptr++ = c;
                ungetc( c );
                *nptr++ = ´\0´;
                return( NAME );
        }
        return( c );
}

bclear( )
{


        nptr = names;
        bptr = bspace;
}

bundle( a1,a2,a3,a4,a5 )
{
        int i, j, *p, *obp;

        p = &a1;
        i = nargs( );
        obp = bptr;
```

```
        for( j=0; j<i; ++j )
                *bptr++ = *p++;
        *bptr++ = 0;
        return( obp );
}

bprint( p )
int *p;
{

        if( p>=bspace && p< &bspace[bsize] ) /* bundle */
                while( *p != 0 )
                        bprint( *p++ );
        else printf( "%s",  p );
}
```

-

The Unix I/O System

Dennis M. Ritchie
Bell Telephone Laboratories

This paper gives an overview of the workings of the Unix I/O system. It was written with an eye toward providing guidance to writers of device driver routines, and is oriented more toward describing the environment and nature of device drivers than the implementation of that part of the file system which deals with ordinary files.

It is assumed that the reader has a good knowledge of the overall structure of the file system as discussed in the paper "The Unix Time-sharing System." Moreover the present document is intended to be used in conjunction with a copy of the system code, since it is basically an exegesis of that code.

Device Classes

There are two classes of device: *block* and *character.* The block interface is suitable for devices like disks, tapes, and DECtape which work, or can work, with addressible 512-byte blocks. Ordinary magnetic tape just barely fits in this category, since by use of forward backward spacing any block can be read, even though blocks can be written only at the end of the tape. Block devices can at least potentially contain a mounted file system. The interface to block devices is very highly structured; the drivers for these devices share a great many routines as well as a pool of buffers.

Character-type devices have a much more straightforward interface, although more work must be done by the driver itself.

Devices of both types are named by a *major* and a *minor* device number. These numbers are generally stored as a word with the minor device number as the low byte and the major device number as the high byte. The major device number selects which driver will deal with the device; the minor device number is not used by the rest of the system but is passed to the driver at appropriate times. Typically the minor number selects a subdevice attached to a given controller, or one of several similar hardware interfaces.

The major device numbers for block and character devices are used as indices in separate tables; they both start at 0 and therefore overlap.

Overview of I/O

The purpose of the *open* and *creat* system calls is to set up entries in three separate system tables. The first of these is the *u_ofile* table, which is stored in the system's per-process data area *u.* This table is indexed by the file descriptor returned by the *open* or *creat,* and is accessed during a *read, write,* or other operation on the open file. An entry contains only a pointer to the corresponding entry of the *file* table, which is a per-system data base. There is one entry in the *file* table for each instance of *open* or *creat.* This table is per-system because the same instance of an open file must be shared among the several processes which can result from *forks* after the file is opened. A *file* table entry contains flags which indicate whether the file was open for reading or writing or is a pipe, and a count which is used to decide when all processes using the entry have terminated or closed the file (so the entry can be abandoned). There is also a 32-bit file offset which is used to indicate where in the file the next read or write will take place. Finally, there is a pointer to the entry for the file in the *inode* table, which contains a copy of the file's i-node. Notice that an entry in the *file* table corresponds precisely to an instance of *open* or *creat;* if the same file is opened several times, it will have several entries in this table. However, there is at most one entry in the *inode* table for a given file. Also, a file may enter the *inode* table not only because it is open, but also because it is the current directory of some process or because it is a special file containing a currently-mounted file system.

An entry in the *inode* table differs somewhat from the corresponding i-node as stored on the disk; the modified and accessed times are not stored, and the entry is augmented by a flag word containing information about the entry, a count used to determine when it may be allowed to disappear, and the device and i-number whence the entry came.

During the processing of an *open* or *creat* call for a special file, the system always calls the device's *open* routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, etc.). However, the *close* routine is called only when the last process closes a file, that is, when the i-node table entry is being deallocated. Thus it is not feasible for a device to maintain, or depend on, a count of its users, although it is quite possible to implement an exclusive-use device which cannot be reopened until it has been closed.

When a *read* or *write* takes place, the user's arguments and the *file* table entry are used to set up the variables *u.u_base, u.u_count,* and *u.u_offset* which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate read or write routine is called; it is responsible for transferring data and updating the count and current location appropriately as discussed below. Otherwise, the current location is used to calculate a logical block number in the file. If the file is an ordinary file the logical block number must be mapped (possibly using an indirect block) to a physical block number; a block-type special file need not be mapped. In any event, the resulting physical block number is used, as discussed below, to read or write the appropriate device.

Character device drivers

The *cdevsw* table specifies the interface routines present for character devices. Each device provides five routines: open, close, read, write, and special-function. Any of these may be missing. If a call on the routine should be ignored, (e.g. *open* on non-exclusive devices which require no setup) the *cdevsw* entry can be given as *nulldev;* if it should be considered an error, (e.g. *write* on read-only devices) *nodev* is used.

The *open* routine is called each time the file is opened with the full device number as argument. The second argument is a flag which is non-zero only if the device is to be written upon.

The *close* routine is called only when the file is closed for the last time, that is when the very last process in which the file is open closes it. This means it is not possible for the driver to maintain its own count of its users. The first argument is the device number; the second is a flag which is non-zero if the file was open for writing in the process which performs the final *close.*

When *write* is called, it is supplied the device as argument. The per-user variable *u.u_count* has been set to the number of characters indicated by the user; for character devices, this number may be 0 initially. *u.u_base* is the address supplied by the user from which to start taking characters. The system may call the routine internally, so the flag *u.u_segflg* is supplied which indicates, if *on,* that *u.u_base* refers to the system address space instead of the user's.

The *write* routine should copy up to *u.u_count* characters from the user's buffer to the device, decrementing *u.u_count* for each character passed. For most drivers, which work one character at a time, the routine

cpass( )

is used to pick up characters from the user's buffer. Successive calls on it return the characters to be written until *u.u_count* goes to 0 or an error occurs, when it returns −1. *Cpass* takes care of interrogating *u.u_segflg* and updating *u.u_count.*

Write routines which want to transfer a probably large number of characters into an internal buffer may also use the routine

iomove(buffer, offset, count, flag)

which is faster when many characters must be moved. *Iomove* transfers up to *count* characters into the *buffer* starting *offset* bytes from the start of the buffer; *flag* should be *B_WRITE* (which is 0) in the write case. Caution: the caller is responsible for making sure the count is not too large and is non-zero. As an efficiency note, *iomove* is much slower if any of *buffer+offset, count* or *u.u_base* is odd.

The device's *read* routine is called under conditions similar to *write,* except that *u.u_count* is guaranteed to be non-zero. To return characters to the user, the routine

passc(c)

is available; it takes care of housekeeping like *cpass* and returns −1 as the last character specified by *u.u_count* is returned to the user; before that time, 0 is returned. *Iomove* is also usable as with *write;* the flag should be *B_READ* but the same cautions apply.

The ‘‘special-functions’’ routine is invoked by the *stty* and *gtty* system calls as follows:

*(*p) (dev, v)*

where *p* is a pointer to the device's routine, *dev* is the device number, and *v* is a vector. In the *gtty* case, the device is supposed to place up to 3 words of status information into the vector; this will be returned to the caller. In the *stty* case, *v* is 0; the device should take up to 3 words of control information from the array *u.u_arg[0...2].*

Finally, each device should have appropriate interrupt-time routines. When an interrupt occurs, it is turned into a C-compatible call on the devices's interrupt routine. The interrupt-catching mechanism makes the low-order four bits of the ''new PS'' word in the trap vector for the interrupt available to the interrupt handler. This is conventionally used by drivers which deal with multiple similar devices to encode the minor device number. After the interrupt has been processed, a return from the interrupt handler will return from the interrupt itself.

A number of subroutines are available which are useful to character device drivers. Most of these handlers, for example, need a place to buffer characters in the internal interface between their ''top half'' (read/write) and ''bottom half'' (interrupt) routines. For relatively low data-rate devices, the best mechanism is the character queue maintained by the routines *getc* and *putc.* A queue header has the structure

```
struct {
        int     c_cc;   /* character count */
        char    *c_cf;  /* first character */
        char    *c_cl;  /* last character */
} queue;
```

A character is placed on the end of a queue by

putc(c, &queue)

where *c* is the character and *queue* is the queue header. The routine returns −1 if there is no space to put the character, 0 otherwise. The first character on the queue may be retrieved by

getc(&queue)

which returns either the (non-negative) character or −1 if the queue is empty.

Notice that the space for characters in queues is shared among all devices in the system and in the standard system there are only some 600 character slots available. Thus device handlers, especially write routines, must take care to avoid gobbling up excessive numbers of characters.

The other major help available to device handlers is the sleep-wakeup mechanism. The call

sleep(event, priority)

causes the process to wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run and the call will return when there is no process with higher *priority.*

The call

wakeup(event)

indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary quantity agreed upon by the sleeper and the waker-up. By convention, it is the address of some data area used by the driver, which guarantees that events are unique.

Processes sleeping on an event should not assume that the event has really happened; they should check that the conditions which caused them to sleep no longer hold.

Priorities can range from 127 to −127; a higher numerical value indicates a less-favored scheduling situation. A process sleeping at negative priority cannot be terminated for any reason, although it is conceivable that it may be swapped out. Thus it is a bad idea to sleep with negative priority on an event which might never occur. On the other hand, calls to *sleep* with non-negative priority may never return if the process is terminated by some signal in the meantime. Incidentally, it is a gross error to call *sleep* in a routine called at interrupt time, since the process which is running is almost certainly not the process which should go to sleep. Likewise, none of the variables in the user area ''*u.*'' should be touched, let alone changed, by an interrupt routine.

If a device driver wishes to wait for some event for which it is inconvenient or impossible to supply a *wakeup,* (for example, a device going on-line, which does not generally cause an interrupt), the call

sleep(&lbolt, priority)

may be given. *Lbolt* is an external cell whose address is awakened once every 4 seconds by the clock interrupt routine.

The routines

spl4( ), spl5( ), spl6( ), spl7( )

are available to set the processor priority level as indicated to avoid inconvenient interrupts from the device.

If a device needs to know about real-time intervals, then

timeout(func, arg, interval)

will be useful. This routine arranges that after *interval* sixtieths of a second, the *func* will be called with *arg* as argument, in the style

*(*func)(arg)*

Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to terminate an attempt to read the 201 Dataphone *dp* if there is no response within a specified number of seconds. Notice that the number of sixtieths of a second is limited to 32767, since it must appear to be positive, and that only a bounded number of timeouts can be going on at once. Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general.

An example

The driver for the paper-tape reader/punch is worth examining as a fairly simple example of many of the techniques used in writing character device handlers. The *pc11* structure contains a state (used for the reader), an input queue, and an output queue. A structure, rather than three individual variables, was used to cut down on the number of external symbols which might be confused with symbols in other routines.

When the file is opened for reading, the *open* routine checks to see if its state is not *CLOSED;* if so an error is returned since it is considered a bad idea to let several people read one tape simultaneously. The state is set to *WAITING,* the interrupt is enabled, and a character is requested. The reason for this gambit is that there is no direct way to determine if there is any tape in the reader or if the reader is on-line. In these situations an interrupt will occur immediately and an error indicated. As will be seen, the interrupt routine ignores errors if the state is *WAITING,* but if a good character comes in while in the *WAITING* state the interrupt routine sets the state to *READING.* Thus *open* loops until the state changes, meanwhile sleeping on the "lightning bolt" *lbolt.* If it did not sleep at all, it would prevent any other process from running until the reader came on-line; if it depended on the interrupt routine to wake it up, the effect would be the same, since the error interrupt is almost instantaneous.

The open-write case is much simpler; the punch is enabled and a 100-character leader is punched using *pcleader.*

The *close* routine is also simple; if the reader was open, any uncollected characters are flushed, the interrupt is turned off, and the state is set to *CLOSED.* In the write case a 100-character trailer is punched. The routine has a bug in that if both the reader and punch are open *close* will be called only once, so that either the leftover characters are flushed or the trailer is punched, but not both. It is hard to see how to fix this problem except by making the reader and punch separate devices.

The *pcread* routine tries to pick up characters from the input queue and passes them back until the user's read call is satisfied. If there are no characters it checks whether the state has gone to *EOF,* which means that the interrupt routine detected an error in the *READ* state (assumed to indicate the end of the tape). If so, *pcread* returns; either during this call or the next one no characters will be passed back, indicating an end-of-file. If the state is still *READING* the routine enables another character by fiddling the device's reader control register, provided it is not active, and goes to sleep.

When a reader interrupt occurs and the state is *WAITING,* and the device's error bit is set, the interrupt is ignored; if there is no error the state is set to *READING,* as indicated in the discussion of *pcread.* If the state is *READING* and there is an error, the state is set to *EOF;* it is assumed that the error represents the end of the tape. If there is no error, the character is picked up and stored in the input queue. Then, provided the number of characters already in the queue is less than the high-water mark *PCIHWAT,* the reader is enabled again to read another character. This strategy keeps the tape moving without flooding the input queue with unread characters. Finally, the top half is awakened.

Looking again at *pcread,* notice that the priority level is raised by *spl4( )* to prevent interrupts during the loop. This is done because of the possibility that the input queue is empty, and just after the EOF test is made an error interrupt occurs because the tape runs out. Then *sleep* will be called with no possibility of a *wakeup.* In general the processor priority should be raised when a routine is about to sleep awaiting some condition where the presence of the condition, and the consequent *wakeup,* is indicated by an interrupt. The danger is that the interrupt might occur between the test for the condition and the call to *sleep,* so that the *wakeup* apparently never happens.

At the same time it is a bad idea to raise the processor priority level for an extended period of time, since devices with real-time requirements may be shut out so long as to cause an error. The *pcread* routine is perhaps overzealous in this respect, although since most devices have a priority level higher than 4 this difficulty is not very important.

The *pcwrite* routine simply gets characters from the user and passes them to *pcoutput,* which is separated out so that *pcleader* can call it also. *Pcoutput* checks for errors (like out-of-tape) and if none are present makes sure that the number of characters in the output queue does not exceed *PCOHWAT;* if it does, *sleep* is called. Then the character is placed on the output queue. There is a small bug here in that *pcoutput* does not check that the character was successfully put on the queue (all character-queue space might be empty); perhaps in this case it might be a good idea to sleep on the lightning-bolt until things quiet down. Finally *pcstart* is called, which checks to see if the punch is currently busy, and if not starts the punching of the first character on the output queue.

When punch interrupts occur, *pcpint* is called; it starts the punching of the next character on the output queue, and if the number of characters remaining on the queue is less than the low-water mark *PCOLWAT* it wakes up the write routine, which is presumably waiting for the queue to empty.

The Block-device Interface

Handling of block devices is mediated by a collection of routines which manage a set of buffers containing the images of blocks of data on the various devices. The most important purpose of these routines is to assure that several processes which access the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block. A secondary but still important purpose is to increase the efficiency of the system by keeping in-core copies of blocks which are being accessed frequently. The main data base for this mechanism is the table of buffers *buf.* Each buffer header contains a pair of pointers *(b_forw, b_back)* which maintain a doubly-linked list of the buffers associated with a particular block device, and a pair of pointers *(av_forw, av_back)* which generally maintain a doubly-linked list of blocks which are ‘‘free,’’ that is, eligible to be reallocated for another transaction. Buffers which have I/O in progress or are busy for other purposes do not appear in this list. The buffer header also contains the device and block number to which the buffer refers, and a pointer to the actual storage associated with the buffer. There is a word count which is the negative of the number of words to be transferred to or from the buffer; there is also an error byte and a residual word count used to communicate information from an I/O routine to its caller. Finally, there is a flag word with bits indicating the status of the buffer. These flags will be discussed below.

Six routines constitute the most important part of the interface with the rest of the system. Given a device and block number, both *bread* and *getblk* return a pointer to a buffer header for the block; the difference is that *bread* is guaranteed to return a buffer actually containing the current data for the block, while *getblk* returns a buffer which contains the data in the block only if it is already in core (whether it is or not is indicated by the *B_DONE* bit; see below). In either case the buffer, and the corresponding device block, is made ‘‘busy,’’ so that other processes referring to it are obliged to wait until it becomes free. *Getblk* is used, for example, when a block is about to be totally rewritten, so that its previous contents are not useful; still, no other process can be allowed to refer to the block until the new data is placed into it.

Given a pointer to a buffer, the *brelse* routine makes the buffer again available to other processes. It is called, for example, after data has been extracted following a *bread.* There are three subtly-different write routines, all of which take a buffer pointer as argument, and all of which logically release the buffer for use by others and place it on the free list. *Bwrite* puts the buffer on the appropriate device queue, waits for the write to be done, and sets the user’s error flag if required.

Bawrite places the buffer on the device’s queue, but does not wait for completion, so that errors cannot be reflected directly to the user. *Bdwrite* does not start any I/O operation at all, but merely marks the buffer so that if it happens to be grabbed from the free list to contain data from some other block, the data in it will first be written out.

Bwrite is used when one wants to be sure that I/O takes place correctly, and that errors are reflected to the proper user; it is used, for example, when updating i-nodes. *Bawrite* is useful when more efficiency is desired (because no

wait is required for I/O to finish) but when it is reasonably certain that the write is really required. *Bdwrite* is used when there is doubt that the write is needed at the moment. For example, *bdwrite* is called when the last byte of a *write* system call falls short of the end of a block, on the assumption that another *write* will be given soon which will re-use the same block. On the other hand, as the end of a block is passed, *bawrite* is called, since probably the block will not be accessed again soon and one might as well start the writing process as soon as possible.

In any event, notice that the routines *getblk* and *bread* dedicate the given block exclusively to the use of the caller, and make others wait, while one of *brelse, bwrite, bawrite,* or *bdwrite* must eventually be called to free the block for use by others.

As mentioned, each buffer header contains a flag word which indicates the status of the buffer. Since they provide one important channel for information between the drivers and the block I/O system, it is important to understand these flags. The following names are manifest constants which select the associated flag bits.

B_READ

This bit is set when the buffer is handed to the device strategy routine (see below) to indicate a read operation. The symbol *B_WRITE* is defined as 0 and does not define a flag; it is provided as a mnemonic convenience to callers of routines like *swap* which have a separate argument which indicates read or write.

B_DONE

This bit is set to 0 when a block is handed to the the device strategy routine and is turned on when the operation completes, whether normally as the result of an error. It is also used as part of the return argument of *getblk* to indicate if 1 that the returned buffer actually contains the data in the requested block.

B_ERROR

This bit may be set to 1 when *B_DONE* is set to indicate that an I/O or other error occurred. If it is set the *b_error* byte of the buffer header may contain an error code if it is non-zero. If *b_error* is 0 the nature of the error is not specified. Actually no driver at present sets *b_error;* the latter is provided for a future improvement whereby a more detailed error-reporting scheme may be implemented.

B_BUSY

This bit indicates that the buffer header is not on the free list, i.e. is dedicated to someone's exclusive use. The buffer still remains attached to the list of blocks associated with its device, however. When *getblk* (or *bread,* which calls it) searches the buffer list for a given device and finds the requested block with this bit on, it sleeps until the bit clears.

B_WANTED

This flag is used in conjunction with the *B_BUSY* bit. Before sleeping as described just above, *getblk* sets this flag. Conversely, when the block is freed and the busy bit goes down (in *brelse)* a *wakeup* is given for the block header whenever *B_WANTED* is on. This strategem avoids the overhead of having to call *wakeup* every time a buffer is freed on the chance that someone might want it.

B_ASYNC

This bit is set by *bawrite* to indicate to the appropriate device driver that the buffer should be released when the write has been finished, usually at interrupt time. The difference between *bwrite* and *bawrite* is that the former starts I/O, waits until it is done, and frees the buffer. The latter merely sets this bit and starts I/O. The bit indicates that *relse* should be called for the buffer on completion.

B_DELWRI

This bit is set by *bdwrite* before releasing the buffer. When *getblk,* while searching for a free block, discovers the bit is 1 in a buffer it would otherwise grab, it causes the block to be written out before reusing it.

B_XMEM

This is actually a mask for the pair of bits which contain the high-order two bits of the physical address of the origin

of the buffer; these bits are an extension of the 16 address bits elsewhere in the buffer header.

B_RELOC

This bit is currently unused; it previously indicated that a system-wide relocation constant was to be added to the buffer address. It was needed during a period when addresses of data in the system (including the buffers) were mapped by the relocation hardware to a physical address differing from its virtual address.

Block Device Drivers

The *bdevsw* table contains the names of the interface routines and that of a table for each block device.

Just as for character devices, block device drivers may supply an *open* and a *close* routine called respectively on each open and on the final close of the device. Instead of separate read and write routines, each block device driver has a *strategy* routine which is called with a pointer to a buffer header as argument. As discussed, the buffer header contains a read/write flag, the core address (including extended-memory bits), the block number, a (negative) word count, and the major and minor device number. The rôle of the strategy routine is to carry out the operation as requested by the information in the buffer header. When the transaction is complete the *B_DONE* (and possibly the *B_ERROR)* bits should be set. Then if the *B_ASYNC* bit is set, *brelse* should be called; otherwise, *wakeup.* In cases where the device is capable, under error-free operation, of transferring fewer words than requested, the device's word-count register should be placed in the residual count slot of the buffer header; otherwise, the residual count should be set to 0. This particular mechanism is really for the benefit of the magtape driver; when reading this device records shorter than requested are quite normal, and the user should be told the actual length of the record. [However the mechanism has not been integrated into normal I/O even on magtape and is used only in ''raw'' I/O as discussed below.]

Notice that although the most usual argument to the strategy routines is a genuine buffer header allocated as discussed above, all that is actually required is that the argument be a pointer to a place containing the appropriate information. For example the *swap* routine, which manages movement of core images to and from the swapping device, uses the strategy routine for this device. Care has to be taken that no extraneous bits get turned on in the flag word.

The device's table specified by *bdevsw* has a byte to contain an active flag and an error count, a pair of links which constitute the head of the chain of buffers for the device *(b_forw, b_back),* and a first and last pointer for a device queue. Of these things, all are used solely by the device driver itself except for the buffer-chain pointers. Typically the flag encodes the state of the device, and is used at a minimum to indicate that the device is currently engaged in transferring information and no new command should be issued. The error count is useful for counting retries when errors occur. The device queue is used to remember stacked requests; in the simplest case it may be maintained as a first-in first-out list. Since buffers which have been handed over to the strategy routines are never on the list of free buffers, the pointers in the buffer which maintain the free list *(av_forw, av_back)* are also used to contain the pointers which maintain the device queues.

A couple of routines are provided which are useful to block device drivers.

iodone(bp) ,

arranges that the buffer to which *bp* points be released or awakened, as appropriate, when the strategy module has finished with the buffer, either normally or after an error. (In the latter case the *B_ERROR* bit has presumably been set.)

When the device conforms to some rather loose standards adhered to by certain DEC hardware, the routine

devstart(bp, devloc, devblk, hbcom)

is useful. Here *bp* is the address of the buffer header, *devloc* is the address of the slot in the device registers which accepts a perhaps-encoded device block number, *devblk* is the block number, and *hbcom* is a quantity to be stored in the high byte of the device's command register. It is understood, when using this routine, that the device registers are laid out in the order

command register
word count
core address
block (or track or sector)

where the address of the last corresponds to *devloc.* Moreover, the device should correspond to the RP, RK, and RF devices with respect to its layout of extended-memory bits and structure of read and write commands.

The routine

geterror(bp)

can be used to examine the error bit in a buffer header and arrange that any error indication found therein is reflected to the user. It may be called only in the non-interrupt part of a driver when I/O has completed *(B_DONE* has been set).

An example

The RF disk driver is worth studying as the simplest example of a block I/O device. Its *strategy* routine checks to see if the requested block lies beyond the end of the device; the size of the disk, in this instance, is indicated by the minor device number. If the request is plausible, the buffer is placed at the end of the device queue, and if the disk is not running, *rfstart* is called.

Rfstart merely returns if there is nothing to do, but otherwise sets the device-active flag, loads the address extension register, and calls *devstart* to perform the remaining tasks attendant on beginning a data transfer.

When a completion or error interrupt occurs, *rfintr* is called. If an error is indicated, and if the error count has not exceeded 10, the same transaction is reattempted; otherwise the error bit is set. If there was no error or if 10 failing transfers have been issued the queue is advanced and *rfstart* is called to begin another transaction.

Raw Block-device I/O

A scheme has been set up whereby block device drivers may provide the ability to transfer information directly between the user's core image and the device without the use of buffers and in blocks as large as the caller requests. The method involves setting up a character-type special file corresponding to the raw device and providing *read* and *write* routines which set up what is usually a private, non-shared buffer header with the appropriate information and call the device's strategy routine. If desired, separate *open* and *close* routines may be provided but this is usually unnecessary. A special-function routine might come in handy, especially for magtape.

A great deal of work has to be done to generate the ''appropriate information'' to put in the argument buffer for the strategy module; the worst part is to map relocated user addresses to physical addresses. Most of this work is done by

physio(strat, bp, dev, rw)

whose arguments are the name of the strategy routine *strat,* the buffer pointer *bp,* the device number *dev,* and a read-write flag *rw* whose value is either *B_READ* or *B_WRITE. Physio* makes sure that the user's base address and count are even (because most devices work in words) and that the core area affected is contiguous in physical space; it delays until the buffer is not busy, and makes it busy while the operation is in progress; and it sets up user error return information.

The magtape driver is the only one which as of this writing provides a raw I/O capability; given *physio,* the work involved is trivial, and amounts to passing back to the user information on the length of the record read or written. (There is some funniness because the magtape, uniquely among DEC devices, works in bytes, not words.) Putting in raw I/O for disks should be equally trivial except that the disk address has to be carefully checked to make sure it does not overflow from one logical device to another on which the caller may not have write permission.

On the Security of UNIX

Dennis M. Ritchie

Bell Laboratories, Murray Hill, N. J.

Recently there has been much interest in the security aspects of operating systems and software. At issue is the ability to prevent undesired disclosure of information, destruction of information, and harm to the functioning of the system. This paper discusses the degree of security which can be provided under the UNIX system and offers a number of hints on how to improve security.

The first fact to face is that UNIX was not developed with security, in any realistic sense, in mind; this fact alone guarantees a vast number of holes. (Actually the same statement can be made with respect to most systems.) The area of security in which UNIX is theoretically weakest is in protecting against crashing or at least crippling the operation of the system. The problem here is not mainly in uncritical acceptance of bad parameters to system calls— there may be bugs in this area, but none are known— but rather in the lack of any checks for excessive consumption of resources. Most notably, there is no limit on the amount of disk storage used, either in total space allocated or in the number of files or directories. Here is a particularly ghastly shell sequence guaranteed to stop the system:

```
: loop
mkdir x
chdir x
goto loop
```

Either a panic will occur because all the i-nodes on the device are used up or all the disk blocks will be consumed, thus preventing anyone from writing files on the device.

Processes are another resource on which the only limit is total exhaustion. For example, the sequence

```
command&
command&
command&
```

if continued long enough will use up all the slots in the system's process table and prevent anyone from executing any commands. Alternatively, if the commands use much core, swap space may run out, causing a panic. Incidently, because of the implementation of process termination, the above sequence is effective in stopping the system no matter how short a time it takes each command to terminate. (The process-table slot is not freed until the terminated process is waited for; if no commands without "&" are executed, the Shell never executes a "wait.")

It should be evident that unbounded consumption of disk space, files, swap space, and processes can easily occur accidentally in malfunctioning programs as well as at command level. In fact UNIX is essentially defenseless against this kind of abuse, nor is there any easy fix. The best that can be said is that it is generally fairly easy to detect what has happened when disaster strikes, to identify the user responsible, and take appropriate action. In practice, we have found that difficulties in this area are rather rare, but we have not been faced with malicious users, and enjoy a fairly generous supply of resources which have served to cushion us against accidental overconsumption.

The picture is considerably brighter in the area of protection of information from unauthorized perusal and destruction. Here the degree of security seems (almost) adequate theoretically, and the problems lie more in the necessity for care in the actual use of the system.

Each UNIX file has associated with it eleven bits of protection information together with a user identification number and a user-group identification number (UID and GID). Nine of the protection bits are used to specify independently permission to read, to write, and to execute the file to the user himself, to members of the user's group, and to all other users. Each process generated by or for a user has associated with it an effective UID and a real UID, and an effective and real GID. When an attempt is made to access the file for reading, writing, or execution, the user process's effective UID is compared against the file's UID; if a match is obtained, access is granted provided the read, write, or execute bit respectively for the user himself is present. If the UID for the file and for the process fail to match, but the GID's do match, the group bits are used; if the GID's do not match, the bits for other users are

tested. The last two bits of each file's protection information, called the set-UID and set-GID bits, are used only when the file is executed as a program. If, in this case, the set-UID bit is on for the file, the effective UID for the process is changed to the UID associated with the file; the change persists until the process terminates or until the UID changed again by another execution of a set-UID file. Similarly the effective group ID of a process is changed to the GID associated with a file when that file is executed and has the set-GID bit set. The real UID and GID of a process do not change when any file is executed, but only as the result of a privileged system call.

The basic notion of the set-UID and set-GID bits is that one may write a program which is executable by others and which maintains files accessible to others only by that program. The classical example is the game-playing program which maintains records of the scores of its players. The program itself has to read and write the score file, but no one but the game's sponsor can be allowed unrestricted access to the file lest they manipulate the game to their own advantage. The solution is to turn on the set-UID bit of the game program. When, and only when, it is invoked by players of the game, it may update the score file ordinary programs executed by others cannot access the score.

There are a number of special cases involved in determining access permissions. Since executing a directory as a program is a meaningless operation, the execute-permission bit, for directories, is taken instead to mean permission to search the directory for a given file during the scanning of a path name; thus if a directory has execute permission but no read permission for a given user, he may access files with known names in the directory, but may not read (list) the entire contents of the directory. Write permission on a directory is interpreted to mean that the user may create and delete files in that directory; it is impossible for any user to write directly into any directory.

Another, and from the point of view of security, much more serious special case is that there is a ''super user'' who is able to read any file and write any non-directory. The super-user is also able to change the protection mode and the owner UID and GID of any file and to invoke privileged system calls. It must be recognized that the mere notion of a super-user is a theoretical, and usually practical, blemish on any protection scheme.

The first necessity for a secure system is of course arranging that all files and directories have the proper protection modes. Unfortunately, UNIX software is exceedingly permissive in this regard; essentially all commands create files readable and writable by everyone. This means that more or less continuous attention must be paid to adjusting modes properly. If one wants to keep one's files completely secret, it is possible to remove all permissions from the directory in which they live, which is easy and effective; but if it is desired to give general read permission while preventing writing, things are more complicated. The main problem is that write permission in a directory means precisely that; it has nothing to do with write permission for a file in that directory. Thus a writeable file in a read-only directory may be changed, or even truncated, though not removed. This fact is perfectly logical, though in this case unfortunate. A case can be made for requiring write permission for the directory of a file as well as for the file itself before allowing writing. (This possibility is more complicated than it seems at first; the system has to allow users to change their own directories while forbidding them to change the user-directory directory.)

A situation converse to the above-discussed difficulty is also present— it is possible to delete a file if one has write permission for its directory independently of any permissions for the file. This problem is related more to self-protection than protection from others. It is largely mitigated by the fact that the two major commands which delete named files (mv and rm) ask confirmation before deleting unwritable files.

It follows from this discussion that to maintain both data privacy and data integrity, it is necessary, and largely sufficient, to make one's directory inaccessible to others. The lack of sufficiency could follow from the existence of set-UID programs created by the user and the possibility of total breach of system security in one of the ways discussed below (or one of the ways not discussed below).

Needless to say, the system administrators must be at least as careful as their most demanding user to place the correct protection mode on the files under their control. In particular, it is necessary that special files be protected from writing, and probably reading, by ordinary users when they store sensitive files belonging to other users. It is easy to write programs that examine and change files by accessing the device on which the files live.

On the issue of password security, UNIX is probably better than most systems. Passwords are stored in an encrypted form which, in the absence of serious attention from specialists in the field, appears reasonably secure, provided its limitations are understood. Since both the encryption algorithm and the encrypted passwords are available, exhaustive enumeration of potential passwords is feasible up to a point. As a practical test of the possibilities in this

area, 67 encrypted passwords were collected from 10 UNIX installations. These were tested against all five-letter combinations, all combinations of letters and digits of length four or less, and all words in Webster's Second unabridged dictionary; 60 of the 67 passwords were found. The whole process took about 12 hours of machine time. This experience suggests that passwords should be at least six characters long and randomly chosen from an alphabet which includes digits and special characters.

Of course there also exist feasible non-cryptanalytic ways of finding out passwords. For example: write a program which types out "login: " on the typewriter and copies whatever is typed to a file of your own. Then invoke the command and go away until the victim arrives. (It is this kind of possibility that makes it evident that UNIX was not designed to be secure.)

The set-UID (set-GID) notion must be used carefully if any security is to be maintained. The first thing to keep in mind is that a writable set-UID file can have another program copied onto it. For example, if the super-user *(su)* command is writable, anyone can copy the shell onto it and get a password-free version of *su.* A more subtle problem can come from set-UID programs which are not sufficiently careful of what is fed into them. In some systems, for example, the *mail* command is set-UID and owned by the super-user. The notion is that one should be able to send mail to anyone even if they want to protect their directories from writing. The trouble is that *mail* is rather dumb: anyone can mail someone else's private file to himself. Much more serious, is the following scenario: make a file with a line like one in the password file which allows one to log in as the super-user. Then make a link named ".mail" to the password file in some writeable directory on the same device as the password file (say /tmp). Finally mail the bogus login line to /tmp/.mail; You can then login as the super-user, clean up the incriminating evidence, and have your will.

The fact that users can mount their own disks and tapes as file systems can be another way of gaining super-user status. Once a disk pack is mounted, the system believes what is on it. Thus one can take a blank disk pack, put on it anything desired, and mount it. There are obvious and unfortunate consequences. For example: a mounted disk with garbage on it will crash the system; one of the files on the mounted disk can easily be a password-free version of *su;* other files can be unprotected entries for special files. The only easy fix for this problem is to forbid the use of *mount* to unprivileged users. A partial solution, not so restrictive, would be to have the *mount* command examine the special file for bad data, set-UID programs owned by others, and accessible special files, and balk at unprivileged invokers.

DC − An Interactive Desk Calculator

Robert Morris

Lorinda Cherry

ABSTRACT

DC is an interactive desk calculator program implemented on the UNIX time-sharing system to do 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 UNIX, the size of numbers that can be handled varies from several hundred digits on the smallest systems to several thousand on the largest.

January 6, 1998

DC − An Interactive Desk Calculator

Robert Morris

Lorinda Cherry

DC is an arbitrary precision arithmetic package implemented on the UNIX time-sharing system in the form of an interactive desk calculator. It 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 [1] 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.

SYNOPTIC DESCRIPTION

Here we describe the DC commands that are intended for use by people. 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.

<*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 UNIX command. Control returns to DC when the UNIX 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.

DETAILED DESCRIPTION

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.

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. This is an implementation of the 'buddy system' of allocation described in [2].

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.

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.

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.

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.

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.

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.

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 successive approximations by the rule

The initial guess is found by taking the integer square root of the top two digits.

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 removed to make the scale of the result the same as if the indicated multiplication had been performed.

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.

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.

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.

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***x* pops the top of the stack and stores the result in register **x**. *x* can be any character. **l***x* puts the contents of register **x** on the top of the stack. The **l** command has no effect on the contents of register *x*. The **s** command, however, is destructive.

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.

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.

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,

```
[lip1+ si li10>a]sa
0si lax
```

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.

Miscellaneous Commands

The command **!** interprets the rest of the line as a UNIX command and passes it to UNIX 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.

DESIGN CHOICES

The real reason for the use of a dynamic storage allocator was that a general purpose program could be (and in fact has been) used for a variety of other tasks. The allocator has some value for input and for compiling (i.e. the bracket [...] commands) where it cannot be known in advance how long a string will be. The result was that at a modest cost in execution time, all considerations of string allocation and sizes of strings were removed from the remainder of the program and debugging was made easier. The allocation method used wastes approximately 25% of available space.

The choice of 100 as a base for internal arithmetic seemingly has no compelling advantage. Yet the base cannot exceed 127 because of hardware limitations and at the cost of 5% in space, debugging was

made a great deal easier and decimal output was made much faster.

The reason for a stack-type arithmetic design was to permit all DC commands from addition to subroutine execution to be implemented in essentially the same way. The result was a considerable degree of logical separation of the final program into modules with very little communication between modules.

The rationale for the lack of interaction between the scale and the bases was to provide an understandable means of proceeding after a change of base or scale when numbers had already been entered. An earlier implementation which had global notions of scale and base did not work out well. If the value of **scale** were to be interpreted in the current input or output base, then a change of base or scale in the midst of a computation would cause great confusion in the interpretation of the results. The current scheme has the advantage that the value of the input and output bases are only used for input and output, respectively, and they are ignored in all other operations. The value of scale is not used for any essential purpose by any part of the program and it is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

The design rationale for the choices for the scales of the results of arithmetic were that in no case should any significant digits be thrown away if, on appearances, the user actually wanted them. Thus, if the user wants to add the numbers 1.5 and 3.517, it seemed reasonable to give him the result 5.017 without requiring him to unnecessarily specify his rather obvious requirements for precision.

On the the other hand, multiplication and exponentiation produce results with many more digits than their operands and it seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless the user asked for them by specifying a value for **scale**. Square root can be handled in just the same way as multiplication. The operation of division gives arbitrarily many decimal places and there is simply no way to guess how many places the user wants. In this case only, the user must specify a **scale** to get any decimal places at all.

The scale of remainder was chosen to make it possible to recreate the dividend from the quotient and remainder. This is easy to implement; no digits are thrown away.

References

[1] L. L. Cherry, R. Morris, *BC − An Arbitrary Precision Desk-Calculator Language,*

[2] K. C. Knowlton, *A Fast Storage Allocator,* Comm. ACM **8**, pp. 623-625 (Oct. 1965)

BC − An Arbitrary Precision Desk-Calculator Language

Lorinda Cherry

Robert Morris

ABSTRACT

BC is a language and a compiler for doing arbitrary precision arithmetic on the PDP-11 under the UNIX time-sharing system. 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 five hundred-digit numbers can be multiplied to give a thousand 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 do computation with large integers,
- to do computation accurate to many decimal places,
- conversion of numbers from one base to another base.

January 6, 1998

BC − An Arbitrary Precision Desk-Calculator Language

Lorinda Cherry

Robert Morris

Introduction

BC is a language and a compiler for doing arbitrary precision arithmetic on the UNIX time-sharing system [1]. The compiler was written to make conveniently available a collection of routines (called DC [6]) 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 UNIX.

The syntax of BC has been deliberately selected to agree substantially with the C language [2,3]. Those who are familiar with C will find few surprises in this language.

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 undesirable 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
```

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.

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.

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.

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.

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
        }
}
```

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. Consult the appendix or the C manuals [2,3] for their exact workings.

x=y=z is the same as	x=(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.

Three Important Things

1. To exit a BC program, type ‘quit’.

2. There is a comment convention identical to that of C and of PL/I. 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 [4].

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.

Acknowledgement

The compiler is written in YACC [5]; its original version was written by S. C. Johnson.

References

[1] K. Thompson and D. M. Ritchie, *UNIX Programmer’s Manual,* Fifth Edition (1974)

[2] D. M. Ritchie, *C Reference Manual,*

[3] B. W. Kernighan, *Programming in C: A Tutorial,*

[4] Robert Morris, *A Library of Reference Standard Mathematical Subroutines,*

[5] S. C. Johnson, *YACC, Yet Another Compiler-Compiler,*

[6] R. Morris and L. L. Cherry, *DC − An Interactive Desk Calculator,*

The Portable C Library (on UNIX) *

M. E. Lesk

1. INTRODUCTION

The C language [1] now exists on three operating systems. A set of library routines common to PDP 11 UNIX, Honeywell 6000 GCOS, and IBM 370 OS has been provided to improve program portability. This memorandum describes the UNIX implementation of the portable routines.

The programs defined here were chosen to follow the standard routines available on UNIX, with alterations to improve transferability to other computer systems. It is expected that future C implementations will try to support the basic library outlined in this document. It provides character stream input and output on multiple files; simple accessing of files by name; and some elementary formatting and translating routines. The remainder of this memorandum lists the portable and non-portable library routines and explains some of the programming aids available.

The I/O routines in the C library fall into several classes. Files are addressed through intermediate numbers called *file-descriptors* which are described in section 2. Several default file-descriptors are provided by the system; other aspects of the system environment are explained in section 3.

Basic character-stream input and output involves the reading or writing of files considered as streams of characters. The C library includes facilities for this, discussed in section 4. Higher-level character stream operations permit translation of internal binary representations of numbers to and from character representations, and formatting or unpacking of character data. These operations are performed with the subprograms in section 5. Binary input and output routines permit data transmission without the cost of translation to or from readable ASCII character representations. Such data transmission should only be directed to files or tapes, and not to printers or terminals. As is usual with such routines, the only simple guarantee that can be made to the programmer seeking portability is that data written by a particular sequence of binary writes, if read by the exactly matching sequence of binary reads, will restore the previous contents of memory. Other reads or writes have system-dependent effects. See section 6 for a discussion of binary input and output. Section 7 describes some further routines in the portable library. These include a storage allocator and some other control and conversion functions.

2. FILE DESCRIPTORS

Except for the standard input and output files, all files must be explicitly opened before any I/O is performed on them. When files are opened for writing, they are created if not already present. They must be closed when finished, although the normal *cexit* routine will take care of that. When opened a disc file or device is associated with a file descriptor, an integer between 0 and 9. This file descriptor is used for further I/O to the file.

Initially you are given three file descriptors by the system: 0, 1, and 2. File 0 is the standard input; it is normally the teletype in time-sharing or input data cards in batch. File 1 is the standard output; it is normally the teletype in time-sharing or the line printer in batch. File 2 is the error file; it is an output file, normally the same as file 1, except that when file 1 is diverted via a command line '>' operator, file 2 remains attached to the original destination, usually the terminal. It is used for error message output. These popular UNIX conventions are considered part of the C library specification. By closing 0 or 1, the default input or output may be re-directed; this can also be done on the command line by *>file* for output or *<file* for input.

---

* This document is an abbreviated form of "The Portable C Library", by M. E. Lesk, describing only the UNIX section of the library.

Associated with the portable library are two external integers, named *cin* and *cout.* These are respectively the numbers of the standard input unit and standard output unit. Initially 0 and 1 are used, but you may redefine them at any time. These cells are used by the routines *getchar, putchar, gets,* and *puts* to select their I-O unit number.

3. THE C ENVIRONMENT

The C language is almost exactly the same on all machines, except for essential machine differences such as word length and number of characters per word. On UNIX ASCII character code is used. Characters range from −128 to +127 in numeric value, there is sign extension when characters are assigned to integers, and right shifts are arithmetic. The ''first'' character in a word is stored in the right half word.

More serious problems of compatibility are caused by the loaders on the different operating systems.

UNIX permits external names to be in upper and lower case, up to seven characters long. There may be multiple external definitions (uninitialized) of the same name.

The C alphabet for identifier names includes the upper and lower case letters, the digits, and the underline. It is not possible for C programs to communicate with FORTRAN programs.

4. BASIC CHARACTER STREAM ROUTINES

These routines transfer streams of characters in and out of C programs. Interpretation of the characters is left to the user. Facilities for interpreting numerical strings are presented in section 5; and routines to transfer binary data to and from files or devices are discussed in section 6. In the following routine descriptions, the optional argument *fd* represents a file-descriptor; if not present, it is taken to be 0 for input and 1 for output. When your program starts, remember that these are associated with the ''standard'' input and output files.

COPEN (filename, type)

Copen initiates activity on a file; if necessary it will create the file too. Up to 10 files may be open at one time. When called as described here, *copen* returns a filedescriptor for a character stream file. Values less than zero returned by *copen* indicate an error trying to open the file. Other calls to *copen* are described in sections 6 and 7.

Arguments :

Filename: a string representing a file name, according to the local operating system conventions. All accept a string of letters and digits as a legal file name, although leading digits are not recommended on GCOS.
Type: a character 'r', 'w', or 'a' meaning read, write, or append. Note that the type is a single character, whereas the file name must be a string.

CGETC (fd)

Cgetc returns the next character from the input unit associated with *fd.* On end of file *cgetc* returns '\0'. To signal end of file from the teletype, type the special symbol appropriate to UNIX: EOT (control-D)

CPUTC (ch , fd)

Cputc writes a character onto the given output unit. *Cputc* returns as its value the character written.

Output for disk files is buffered in 512 character units, irrespective of newlines; teletype output goes character by character

CCLOSE (fd)

Activity on file *fd* is terminated and any output buffers are emptied. You usually don't have to call *cclose; cexit* will do it for you on all open files. However, to write some data on a file and then read it back in, the correct sequence is:

```
fd = copen (''file'', 'w');
write on fd ...
cclose (fd);
fd = copen(''file'', 'r');
read from fd ...
```

CFLUSH (fn)

To get buffer flushing, but retain the ability to write more on the file, you may call this routine.

Normally, output intended for the teletype is not buffered and this call is not needed.

CEXIT ([errcode])

Cexit closes all files and then terminates execution. If a non-zero argument is given, this is assumed to be an error indication or other returned value to be signalled to the operating system.

Cexit **must** be called explicitly; a return from the main program is not adequate.

CEOF (fd)

Ceof returns nonzero when end of file has been reached on input unit *fd.*

GETCHAR ()

Getchar is a special case of *cgetc;* it reads one character from the standard input unit. *Getchar ( )* is defined as *cgetc (cin);* it should not have an argument.

PUTCHAR (ch)

Putchar (ch) is the same as *cputc (ch, cout);* it writes one character on the standard output.

GETS (s)

Gets reads everything up to the next newline into the string pointed to by *s.* If the last character read from this input unit was newline, then *gets* reads the next line, which on GCOS and IBM corresponds exactly to a logical record. The terminating newline is replaced by '\0'. The value of *gets* is *s,* or 0 if end of file.

PUTS (s)

Copies the string *s* onto the standard output unit. The terminating '\0' is replaced by a newline character. The value of *puts* is *s.*

UNGETC (ch , fd)

Ungetc pushes back its character argument to the unit *fd,* which must be open for input. After *ungetc ('a', fd); ungetc ('b', fd);* the next two characters to be read from *fd* will be 'b' and then 'a'. Up to 100 characters may be pushed back on each file. This subroutine permits a program to read past the end of its input, and then restore it for the next routine to read. It is impossible to change an external file with *ungetc;* its purpose is only for internal communications, most particularly *scanf,* which is described in section 5. Note that *scanf* actually requires only one character of ''unget'' capability; thus it is possible that future implementors may change the specification of the *ungetc* routine.

5. HIGH-LEVEL CHARACTER STREAM ROUTINES

These two routines, *printf* for output and *scanf* for input, permit simple translation to and from character representations of numerical quantities. They also allow generation or interpretation of formatted lines.

PRINTF ([fd,] control-string, arg1, arg2, ...)

PRINTF ([−1, output-string,] control-string, arg1, arg2, ...)

Printf converts, formats, and prints its arguments under control of the control string. The control string contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to *printf.*

Each conversion specification is introduced by the character '%'. Following the '%', there may be:

— an optional minus sign '−' which specifies left adjustment of the converted argument in the indicated field;

— an optional digit string specifying a minimum field width; if the converted argument has fewer characters than the field width it will be padded on the left (or right, if the left adjustment indicator has been given) to make up the field width; the padding character is blank normally and zero if the field width was specified with a leading zero (note that this does not imply an octal field width);

— an optional period '.' which serves to separate the field width from the next digit string;

— an optional digit string (the precision) which specifies the maximum number of characters to be printed from a string, or the number of digits to be printed to the right of the decimal point of a floating or double number.

— an optional length modifier 'l' which indicates that the corresponding data item is a *long* rather than an *int.*

— a character which indicates the type of conversion to be applied.

The conversion characters and their meanings are:

d The argument is converted to decimal notation.

o The argument is converted to octal notation.

x The argument is converted to hexadecimal notation.

u The argument is converted to unsigned decimal notation. This is only implemented (or useful) on UNIX.

c The argument is taken to be a single character.

s The argument is taken to be a string and characters from the string are printed until a null character is reached or until the number of characters indicated by the precision specification is exhausted.

e The argument is taken to be a float or double and converted to decimal notation of the form *[−]m.nnnnnnE[−]xx* where the length of the string of *n*'s is specified by the precision. The default precision is 6 and the maximum is 22.

f The argument is taken to be a float or double and converted to decimal notation of the form *[−]mmm.nnnnn* where the length of the string of *n*'s is specified by the precision. The default precision is 6 and the maximum is 22. Note that the precision does not determine the number of significant digits printed in **f** format.

If no recognizable conversion character appears after the '%', that character is printed; thus '%' may be printed by use of the string ''%%''.

As an example of *printf,* the following program fragment

```
int i, j; float x; char *s;
i = 35; j=2; x= 1.732; s = "ritchie";
printf ("%d %f %s\n", i, x, s);
printf ("%o, %4d or %−4d%5.5s\n", i, j, j, s);
```

would print

```
35 1.732000 ritchie
043,      2 or 2    ritch
```

If *fd* is not specified, output is to unit *cout.* It is possible to direct output to a string instead of to a file. This is indicated by −1 as the first argument. The second argument should be a pointer to the string. *Printf* will put a terminating '\0' onto the string.

SCANF ([fd,] control-string, arg1, arg2, ...)

SCANF ([−1, input-string,] control-string, arg1, arg2, ...)

Scanf reads characters, interprets them according to a format, and stores the results in its arguments. It expects as arguments:

1. An optional file-descriptor or input-string, indicating the source of the input characters; if omitted, file *cin* is used;
2. A control string, described below;
3. A set of arguments, *each of which must be a pointer,* indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs or newlines, which are ignored.
2. Ordinary characters (not %) which are expected to match the next non-space character of the input stream (where space characters are defined as blank, tab or newline).
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character ∗, an optional numerical maximum field width, and a conversion character.

A conversion specification is used to direct the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by the ∗ character. An input field is defined as a string of non-space characters; it extends either to the next space character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. Pointers, rather than variable names, are required by the ''call-by-value'' semantics of the C language. The following conversion characters are legal:

% indicates that a single % character is expected in the input stream at this point; no assignment is done.

d indicates that a decimal integer is expected in the input stream; the corresponding argument should be an integer pointer.

o indicates that an octal integer is expected in the input stream; the corresponding argument should be a integer pointer.

x indicates that a hexadecimal integer is expected in the input stream; the corresponding argument should be an integer pointer.

s indicates that a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating '\0', which will be added. The input field is terminated by a space character or a newline.

c indicates that a single character is expected; the corresponding argument should be a character pointer; the next input character is placed at the indicated spot. The normal skip over space characters is suppressed in this case; to read the next non-space character, try *%1s*.

e or **f** indicates that a floating point number is expected in the input stream; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a float. The input format for *floats* is a string of numbers possibly containing a decimal point, followed by an optional exponent field containing an E or e followed by a possibly signed integer.

[indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not circumflex (^), the input field is all

characters until the first character not in the set between the brackets; if the first character after the left bracket is ^, the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters *d, o* and *x* may be preceded by *l* to indicate that a pointer to *long* rather than *int* is expected. Similarly, the conversion characters *e* or *f* may be preceded by *l* to indicate that a pointer to *double* rather than *float* is in the argument list. The character *h* will function similarly in the future to indicate *short* data items.

For example, the call

```
int i; float x; char name[50];
scanf ( "%d%f%s", &i, &x, name);
```

with the input line

```
25   54.32E−1  thompson
```

will assign to *i* the value 25, *x* the value 5.432, and *name* will contain *"thompson\0"*. Or,

```
int i; float x; char name[50];
scanf ("%2d%f%*d%[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to *i,* 789.0 to *x,* skip ''0123'', and place the string ''56\0'' in *name.* The next call to *cgetc* will return 'a'.

Scanf returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, −1 is returned; note that this is different from 0, which means that the next input character does not match what you called for in the control string. *Scanf,* if given a first argument of −1, will scan a string in memory given as the second argument. For example, if you want to read up to four numbers from an input line and find out how many there were, you could try

```
int a[4], amax;
char line[100];
amax = scanf (−1, gets(line), "%d%d%d%d", &a[0], &a[1], &a[2], &a[3]);
```

6. BINARY STREAM ROUTINES

These routines write binary data, not translated to printable characters. They are normally efficient but do not produce files that can be printed or easily interpreted. No special information is added to the records and thus they can be handled by other programming systems *if* you make the departure from portability required to tell the other system how big a C item (integer, float, structure, etc.) really is in machine units.

COPEN (name, direction, "i")

When *copen* is called with a third argument as above, a binary stream filedescriptor is returned. Such a file descriptor is required for the remaining subroutines in this section, and may not be used with the routines in the preceding two sections. The first two arguments operate exactly as described in section 3; further details are given in section 7. An ordinary file descriptor may be used for binary I-O, but binary and character I-O may not be mixed unless *cflush* is called at each switch to binary I-O. The third argument to *copen* is ignored.

*CWRITE (ptr, sizeof(*ptr), nitems, fd)*

Cwrite writes *nitems* of data beginning at *ptr* on file *fd. Cwrite* writes blocks of binary information, not translated to printable form, on a file. It is intended for machine-oriented bulk storage of intermediate data. Any kind of data may be written with this command, but only the corresponding *cread* should be expected to make any sense of it on return. The first argument is a pointer to the beginning of a vector of any kind of data. The second argument tells *cwrite* how big the items are. The third argument specifies the number of the items to be written; the fourth indicates where.

CREAD (ptr, sizeof(∗ptr), nitems, fd)

Cread reads up to *nitems* of data from file *fd* into a buffer beginning at *ptr.* *Cread* returns the number of items read.
The returned number of items will be equal to the number requested by *nitems* except for reading certain devices (e.g. the teletype or magnetic tape) or reading the final bytes of a disk file.

Again, the second argument indicates the size of the data items being read.

CCLOSE (fd)

The same description applies as for character-stream files.

7. OTHER PORTABLE ROUTINES

REW (fd)

Rewinds unit *fd.* Buffers are emptied properly and the file is left open.

SYSTEM (string)

The given *string* is executed as if it were typed at the terminal.

NARGS ( )

A subroutine can call this function to try to find out how many arguments it was called with. Normally, *nargs()* returns the number of arguments plus 3 for every *float* or *double* argument and plus one for every *long* argument. If the new UNIX feature of separated instruction and data space areas is used, *nargs()* doesn't work at all.

CALLOC (n, sizeof(object))

Calloc returns a pointer to new storage, allocated in space obtained from the operating system. The space obtained is well enough aligned for any use, i.e. for a double-precision number. Enough space to store *n* objects of the size indicated by the second argument is provided. The *sizeof* is executed at compile time; it is not in the library. A returned value of −1 indicates failure to obtain space.

*CFREE (ptr, n, sizeof(*ptr))*

Cfree returns to the operating system memory starting at *ptr* and extending for *n* units of the size given by the third argument. The space should have been obtained through *calloc*. On UNIX you can only return the exact amount of space obtained by *calloc;* the second and third arguments are ignored.

FTOA (floating-number, char-string, precision, format)

Ftoa (floating to ASCII conversion) converts floating point numbers to character strings. The *format* argument should be either 'f' or 'e'; 'e' is default. See the explanation of *printf* in section 5 for a description of the result.

ATOF (char-string)

Returns a floating value equal to the value of the ASCII character string argument, interpreted as a decimal floating point number.

TMPNAM (str)

This routine places in the character array expected as its argument a string which is legal to use as a file name and which is guaranteed to be unique among all jobs executing on the computer at the same time. It is thus appropriate for use as a temporary file name, although the user may wish to move it into an appropriate directory. The value of the function is the address of the string.

ABORT (code)

Causes your program to terminate abnormally, which typically results in a dump by the operating system.

INTSS ()

This routine tells you whether you are running in foreground or background. The definition of ‘‘foreground’’ is that the standard input is the terminal.

WDLENG ( )

This returns 16 on UNIX. C users should be aware that the preprocessor normally provides a defined symbol suitable for distinguishing the local system; thus on UNIX the symbol *unix* is defined before starting to compile your program.

A. Hardware

UNIX runs on a DEC PDP11/40*, 11/45 or 11/70 with at least the following equipment:

48K to 124K words of managed memory: parity not used,
disk: RP03, RP04, RK05(preferably 2) or equivalent,
console typewriter,
clock: KW11-L or KW11-P,
extended instruction set KE11-E, on 11/40 only.

The system is normally distributed on 9-track tape or RK05 packs.
The following equipment is strongly recommended:

communications controllers such as DL11, DC11 or DH11,
full duplex 96-character ASCII terminals,
9-track tape, or extra disk for system backup.

The minimum memory and disk space specified is enough to run and maintain UNIX. More will be needed to keep all source on line, or to handle a large number of users, big data bases, diversified complements of devices, or large programs. UNIX does swapping and reentrant sharing of user code to minimize main memory requirements. The resident code of UNIX occupies 20-22K depending on configuration.

An 11/40 is not advisable for heavy floating point work, as UNIX on this hardware uses interpreted 11/45 floating point.

B. Software

All the programs available as UNIX commands are listed. Every command, including all options, is issued as just one line, unless specifically noted as ''interactive''. Interactive programs can be made to run from a prepared script simply by redirecting input.

File processing commands that go from standard input to standard output are noted as usable as filters. The piping facility of the Shell may be used to connect filters directly to the input or output of other programs.

Commercially distributed UNIX normally excludes software listed in Section 5, ''Typesetting.'' Source code is included except as noted.

1 Basic Software

This package includes time-sharing operating system with utilities, machine language assembler and the compiler for the programming language C—enough software to write and run new applications and to maintain or modify UNIX itself.

1.1 Operating System

□ UNIX — The basic resident code on which everything else depends. Supports the system calls, and maintains the file system. A general description of UNIX design philosophy and system facilities appeared in the Communications of the ACM, July, 1974. Further capabilities include:
- Automatically supported reentrant code.
- Separate instruction and data spaces on 11/45 and 11/70.
- ''Group'' access permissions allow cooperative projects, with overlapping memberships.
- Timer-interrupt sampling and interprocess monitoring for debugging and measurement.

□ Manual — Printed manuals for UNIX and all its software, except where other manuals exist.
- UNIX Programmer's Manual.
- The UNIX Time-Sharing System, reprint setting forth design principles.
- UNIX for Beginners.

• The UNIX I-O System.
• On the Security of UNIX.

□ (DEV) All I/O is logically synchronous. Normally, invisible buffering makes all physical record structure transparent and exploits the hardware's ability to do overlapped I/O. Unbuffered physical record I/O is available for unusual applications. Drivers for these devices are available; others can be easily written:
• Asynchronous interfaces: DC11, DH11, DL11. Support for most common ASCII terminals.
• Synchronous interface: DP11.
• Automatic calling unit interface: DN11.
• Line printer: LP11.
• Magnetic tape: TU10 and TU16.
• DECtape*: TC11.
• Paper tape: PC11.
• Fixed head disk: RS11, RS03 and RS04.
• Pack type disk: RP03 and RP04, one or more logical devices per physical device, minimum-latency seek scheduling.
• Cartridge-type disk: RK05, one or more physical devices per logical device.
• Null device.
• Physical memory of PDP11, or mapped memory in resident system.
• Phototypesetter: Graphic Systems System/1 through DR11C.
• Voice synthesizer: VOTRAX* through DC11. Includes TOUCH-TONE® input.

□ BOOT Procedures to get UNIX up on a naked machine.

□ Manual Setting up UNIX.

□ MKCONF Tailor device-dependent system code to hardware configuration. Other changes, such as optimal assignment of directories to devices, inclusion of floating point simulator, or installation of device names in file system can then be made at leisure. (As distributed, UNIX can be brought up directly on any acceptable CPU with any acceptable disk, any sufficient amount of core and either clock.)

□ Manual Printed manual on setting up UNIX.

1.2 User Access Control

□ LOGIN Sign on as a new user.
• Verify password and establish user's individual and group (project) identity.
• Adapt to characteristics of terminal.
• Establish working directory.
• Announce presence of mail (from MAIL).
• Publish message of the day.
• Start command interpreter or other initial program.

□ PASSWD Change a password.
• User can change his own password.
• Passwords are kept encrypted for better security.

□ NEWGRP Change working group (project). Protects against changes to unauthorized projects.

1.3 File Manipulation

□ CAT Concatenate one or more files onto standard output. Particularly used for unadorned printing, for inserting data into a pipeline, and for buffering output that comes in dribs and drabs.
• Usable as filter.

□ CP — Copy one file to another. Works on any file without distinction as to contents.

□ PR — Print files with title, date, and page number on every page.
- Multicolumn output.
- Parallel column merge of several files.
- Usable as a filter.

□ OPR — Off line print. Spools arbitrary files to the line printer.
- Usable as a filter.

□ SPLIT — Split a large file into more manageable pieces. Is occasionally necessary for editing (ED).

□ ED — Interactive context editor. Can work on single lines, blocks of lines, or all pattern-selected lines in a given range.
- Finds lines by number or pattern.
- Random access to lines.
- Add, delete, change, copy or move lines.
- Permute or split contents of a line.
- Replace one or all instances of a pattern within a line.
- Combine or split files.
- Escape to Shell (UNIX command language) during editing.
- Patterns may include:
  - specified characters,
  - don't care characters,
  - choices among characters,
  - repetitions of above,
  - beginning of line,
  - end of line.
- All operations may be done globally on every pattern-selected line in a given range.

□ Manual — Introductory manual for ED.

□ DD — Physical file format translator, for exchanging data with foreign systems, especially OS/360.

□ STTY — Sets up options for optimal control of a terminal. In so far as they are deducible from the input, these options are set automatically by LOGIN.
- Half vs. full duplex.
- Carriage return+line feed vs. newline.
- Interpretation of tabs.
- Parity.
- Mapping of upper case to lower.
- Raw vs. edited input.
- Delays for tabs, newlines and carriage returns.

1.4 Manipulation of Directories and File Names

□ RM — Remove a file. Only the name goes away if any other names are linked to the file.

□ LN — "Link" another name (alias) to an existing file.

□ MV — Move a file.
- Used for renaming files or directories.

□ CHMOD — Change permissions on one or more files. Executable by files' owner.

□ CHOWN Change owner of one or more files.

□ CHGRP Change group (project) to which a file belongs.

□ MKDIR Make a new directory.

□ RMDIR Remove a directory.

□ CHDIR Change working directory.

□ FIND Prowl the directory hierarchy finding every file that meets specified criteria.
- These criteria are understood:
  - spelling of name matches a given pattern,
  - creation date in given range,
  - date of last use in given range,
  - permissions,
  - owner,
  - characteristics of device files,
  - boolean combinations of above.
- Any directory may be considered to be the root.
- Specified commands may be performed on every file found.

□ DSW Interactively step through a directory, deleting or keeping files.

1.5 Running of Programs

□ SH The Shell, or command language interpreter. Provides "background" and macro capability when run with a file of commands as input.
- Any executable object file is automatically a command.
- Redirect standard input or standard output.
- Operators to compose compound commands:
  - ';' for sequential execution,
  - '|' for functional composition with output of one command taken directly as input to another running simultaneously,
  - '&' for asynchronous operation,
  - parentheses for grouping.
- Substitutable arguments.
- Construction of argument lists from all file names satisfying specified patterns.
- Collects command usage statistics.

□ IF A conditional statement for Shell programs.
- String comparison.
- Querying file accessibility.

□ GOTO A "go-to" statement for Shell programs.

□ WAIT Wait for termination of asynchronously running processes.

□ EXIT Terminate a Shell program. Useful with IF.

□ ECHO Print remainder of command line. Useful for diagnostic or prompting data in Shell programs, or for inserting data into a pipeline.

□ SLEEP Suspend execution for a specified time.

□ NOHUP Run a command immune to hanging up the terminal.

□ NICE Run a command in low (or high) priority.

□ KILL Terminate named processes.

□ CRON A table of actions to be taken at specified times.
- Actions are arbitrary Shell (SH) scripts.
- Times are conjunctions of month, day of month, day of week, hour and minute. Ranges are specifiable for each.

□ TEE Pass data between processes and divert a copy into a file. Used as a filter.

1.6 Status Inquiries

□ LS List the names of one, several, or all files in one or more directories.
- Alphabetic or temporal sorting, up or down.
- Optional information: size, owner, group, date last modified, date last accessed, permissions, i-node number.

□ FILE Tries to determine what kind of information is in a file by consulting the file system index and by reading the file itself.

□ DATE System date routine. Has considerable knowledge of calendric and horological peculiarities.
- Print present date, day of week, local time.
- May set UNIX's idea of date and time.

□ DF Report amount of free space on file system devices.

□ DU Print a summary of total space occupied by all files in a hierarchy.

□ WHO Tell who's on the system.
- List of presently logged in users, ports and times on.
- Optional history of all logins and logouts.

□ PS Report on all active processes attached to a terminal.
- Gives all commands being executed.
- Can also report on other terminals.
- Extended status information available:
  state and scheduling info,
  priority,
  attached terminal,
  what it's waiting for,
  size.

□ TTY Find name of your terminal.

□ PWD Print name of your working directory.

□ PFE Print type of last floating exception.

1.7 Backup and Maintenance

□ MOUNT Attach a device containing a file system to the tree of directories. Protects against nonsense arrangements.

□ UMOUNT — Remove the file system contained on a device from the tree of directories. Protects against removing a busy device.

□ MKFS — Make a new file system on a device.

□ MKNOD — Make an i-node (file system entry) for a special file. Special files are physical devices, virtual devices, physical memory, etc.

□ TP — Manage file archives on magnetic tape or DEC tape.
- Collect files into an archive.
- Update DECtape archive by date.
- Replace or delete DECtape files.
- Table of contents.
- Retrieve from archive.

□ DUMP — Dump the file system stored on a specified device, selectively by date, or indiscriminately.

□ RESTOR — Restore a dumped file system, or selectively retrieve parts thereof.

□ SU — Temporarily become the super user with all the rights and privileges thereof. Requires a password.

□ DCHECK
□ ICHECK
□ NCHECK

Check consistency of file system.
- Gross statistics:
  - number of files,
  - number of directories,
  - number of special files,
  - spaced used,
  - space free.
- Report of duplicate use of space.
- Retrieval of lost space.
- Report of inaccessible files.
- Check consistency of directories.
- List names of all files.

□ CLRI — Peremptorily expunge a file and its space from a file system. Used in putting damaged file systems together again.

□ SYNC — Force all outstanding I/O on the system to completion. Used to shut down gracefully.

1.8 Accounting

These routines use floating point. The timing information on which the reports are based can be manually cleared or shut off completely.

□ AC — Publish cumulative connect time report.
- Connect time by user or by day.
- For all users or for selected users.

□ SA — Publish Shell accounting report. Gives usage information on each command executed.
- Number of times used.
- Total system time, user time and elapsed time.
- Optional averages and percentages.
- Sorting on various fields.

1.9 Inter-user Communication

□ MAIL — Mail a message to one or more users. Also used to read and dispose of incoming mail. The presence of mail is announced by LOGIN.

□ WRITE — Establish direct terminal communication with another user.

□ WALL — Write to all users.

□ MESG — Inhibit receipt of messages from WRITE and WALL.

1.10 Basic Program Development Package

A kit of fundamental programming tools. Some of these utilities are used as integral parts of the higher level languages described below.

□ AR — Archive and library maintainer. Combines several files into one for housekeeping efficiency. Archive files are used by the link editor LD as libraries.
- Create new archive.
- Update archive by date.
- Replace or delete files.
- Table of contents.
- Retrieve from archive.

□ AS — Assembler. Similar to PAL-11, but different in detail.
- Creates object program consisting of
  code, possibly read-only,
  initialized data or read-write code,
  uninitialized data.
- Relocatable object code is directly executable without further transformation.
- Object code normally includes a symbol table.
- Combines source files.
- Local labels.
- Conditional assembly.
- ‘‘Conditional jump’’ instructions become branches or branches plus jumps depending on distance.

□ Manual — Printed manual for the assembly language.

□ Library — The basic run-time library. These routines are used freely by all system software.
- Formatted writing on standard output.
- Time conversions.
- Convert integer and floating numbers to ASCII and vice versa.
- Elementary functions: sin, cos, log, exp, atan, sqrt, gamma.
- Password encryption.
- Quicksort.
- Buffered character-by-character I/O.
- Random number generator.
- Floating point interpreter for 11/40’s and non-floating point machines.

□ (LIBP) — An elaborated I/O library.
- Formatted input and output.
- Ability to put characters back into input streams.

□ Manual — Printed manual for LIBP.

□ DB — Interactive post-mortem debugger. Works on core dump files, such as are produced by all program aborts, on object files, or on any arbitrary file.
- Symbolic addressing of files that have symbol tables.
- Octal, decimal or ASCII output.
- Symbolic disassembly.
- Octal or decimal patching.

□ OD — Dump any file.
- Output options include:
  - octal or decimal by words,
  - octal by bytes,
  - ASCII,
  - opcodes,
  - hexadecimal,
  - any combination of the above.
- Range of dumping is controllable.

□ LD — Link edit. Combine relocatable object files. Insert required routines from specified libraries.
- Resulting code may be sharable.
- Resulting code may have separate instruction and data spaces.

□ NM — Print the namelist (symbol table) of an object program. Provides control over the style and order of names that are printed.

□ SIZE — Report the core requirements of one or more object files.

□ STRIP — Remove the relocation and symbol table information from an object file to save space.

□ TIME — Run a command and report timing information on it.

□ PROF — Construct a profile of time spent per routine from statistics gathered by time-sampling the execution of a program. Uses floating point.
- Subroutine call frequency and average times for C programs.

1.11 The Programming Language "C"

□ CC — Compile and/or link edits programs in the C language. The UNIX operating system, most of the subsystems and C itself are written in C.
- Full general purpose language designed for structured programming.
- Data types:
  - character,
  - integer,
  - float,
  - double,
  - pointers to all types,
  - arrays of all types,
  - structures of all types,
  - functions returning all types.
- Operations intended to give access to full machine facility, including to-memory operations and data-sensitive pointer arithmetic.
- Macro preprocessor for parameterized code and inclusion of standard files.
- All procedures recursive, with parameters by value.
- Natural coercions.

• True compiled object code capitalizing on addressing capability of the PDP11.
• Runtime library gives access to all system facilities.

□ Manuals — Printed manual and tutorial for the C language.

□ CDB — An interactive debugger tailored for use with C.
• Usable in real time or post-mortem.
• The debugger is a completely separate process from the debuggee. No debugging code is loaded with debuggee.
• Prints all kinds of data in natural notation:
character,
integer (octal and decimal),
float,
double,
machine instructions (disassembled).
• Stack trace and fault identification.
• Breakpoint tracing.

2 Other Languages

2.1 FORTRAN

□ FC — Compile and/or link-edit FORTRAN IV programs. Object code is ''threaded''. Relies heavily on floating point.
• Idiosyncracies:
free form, lower-case source code,
no arithmetic statement functions,
unformatted I/O requires record lengths agree,
no BACKSPACE,
no P FORMAT control on input.
• Handles mixed-mode arithmetic, general subscripts and general DO limits.
• 32-bit integer arithmetic.
• Free format numeric input.
• Understands these nonstandard specifications:
LOGICAL*1, *2, *4,
INTEGER*2, *4,
REAL*4, *8,
COMPLEX*8, *16,
IMPLICIT.

□ RC — ''Ratfor'', a preprocessor that adds rational control structure à la C to FORTRAN.
• Else, for, while, repeat...until statements.
• Symbolic constants.
• File insertion.
• Compound statements.
• Can produce genuine FORTRAN to carry away.

□ Manual — Printed manual for Ratfor.

2.2 Other Algorithmic Languages

□ BAS — An interpreter, similar in style to BASIC, that allows immediate execution of unnumbered statements, or deferred execution of numbered statements.
• Statements include:
comment,

dump,
for...next,
goto,
if...else...fi,
list,
print,
prompt,
return,
run,
save.

- All calculations double precision.
- Recursive function defining and calling.
- Builtin functions include log, exp, sin, cos, atn, int, sqr, abs, rnd.
- Escape to ED for complex program editing.
- Usable as a filter.

□ DC — Programmable reverse Polish desk calculator. Has named storage locations as well as conventional stack for holding integers or programs.

- Unlimited precision decimal arithmetic.
- Appropriate treatment of decimal fractions.
- Arbitrary input and output radices, in particular binary, octal, decimal and hexadecimal.
- Operators include:
  + − * /
  remainder, power, square root,
  load, store, duplicate, clear,
  print, enter program text, execute.
- Usable as a filter.

□ BC — A C-like interface to the desk calculator DC.

- All the capabilities of DC with a high-level syntax.
- Arrays and recursive functions.
- Immediate evaluation of expressions and evaluation of functions upon call.
- Arbitrary precision elementary functions: exp, sin, cos, atan, J_n.
- Go-to-less programming.
- Usable as a filter.

□ Manual — Printed manual for BC.

□ SNO — An interpreter very similar to SNOBOL 3.

- Limitations:
  function definitions are static,
  pattern matches are always anchored,
  no built-in functions.
- Usable as a filter.

□ Manual — Reprint of basic article.

2.3 Macroprocessing

□ M6 — A general purpose macroprocessor.

- Stream-oriented, recognizes macros anywhere in text.
- Integer arithmetic.
- Usable as a filter.

□ Manual Printed manual for M6.

2.4 Compiler-compilers

□ TMG A classical top-down compiler-compiler language. Provides a formalism for syntax-directed translation. Produces driving tables to be loaded with a standard interpreter.
• Resulting compilers can have arbitrary tables kept in paged secondary store.
• Integer arithmetic capability.
• Syntactic function capability (similar to ALGOL 68 metaproductions).

□ Manual Printed manual for the TMG compiler-writing system.

□ YACC An LR(1)-based compiler writing system. During execution of resulting parsers, arbitrary C-language, Ratfor or FORTRAN functions may be called to do code generation or semantic actions.
• BNF syntax specifications.
• Handles precedence relations.
• Accepts formally ambiguous grammars with non-BNF resolution rules.
• Optimizes space taken by driving tables.

□ Manual Printed manual for the YACC compiler-writing system.

3 Word Processing

□ ROFF A typesetting program for terminals. Easy for nontechnical people to learn, and good for most ordinary kinds of documents. Input consists of data lines intermixed with control lines, such as

```
.sp 2      insert two lines of space
.ce        center the next line
```

• Justification of either or both margins.
• Automatic hyphenation.
• Generalized running heads and feet, with even-odd page capability, numbering, etc.
• Definable macros for frequently used control sequences (no substitutable arguments).
• All 4 margins and page size dynamically adjustable.
• Hanging indents and one-line indents.
• Absolute and relative parameter settings.
• Optional legal-style numbering of output lines.
• Multiple file capability.

□ CREF Make cross-reference listings of a collection of files. Each symbol is listed together with file, line number, and text of each line in which it occurs.
• Assembler or C language.
• Gathering or suppressing references to selected symbols.
• Last symbol defined may replace line number.
• Various ways to sort output available.
• Selective print of uniquely occurring symbols.

□ INDEX Make cross-reference indexes of English text.
• Handles lists of specific index terms or excluded terms.
• Handles words hyphenated across lines.
• Understands TROFF and NROFF output, so can gather references according to final pagination.
• Output capabilities like CREF.
• Frequency counts.

□ FORM Form letter generator. Remembers any number of forms and stock phrases such as names and addresses. Output usually intended to be ROFFed.
• Anything that is typed in can be remembered for later use.

• Runs interactively, querying only for those items that are not in its memory.
• Any item may call for the inclusion of other items. For example, full name, address, first name, title, etc., may be separately retrieved from one name key.

□ FED — Editor for the memory used by FORM. Extract any item, turn it over to context editor ED for editing, and put it back when done.
• List names of selected items.
• Print contents of selected item.

□ SORT — Sort or merge ASCII files line-by-line.
• Sort up or down.
• Sort lexicographically or on numeric key.
• Multiple keys located by delimiters or by character position.
• May sort upper case together with lower into dictionary order.
• Usable as a filter.

□ UNIQ — Collapse successive duplicate lines in a file into one line.
• Publishes lines that were originally unique, duplicated, or both.
• May give redundancy count for each line.
• Usable as a filter.

□ TR — Do one-to-one character translation according to an arbitrary code.
• May coalesce selected repeated characters.
• May delete selected characters.
• Usable as a filter.

□ DIFF — Report line changes, additions and deletions necessary to bring two files into agreement.
• May produce an editor script to convert one file into another.

□ COMM — Identify common lines in two sorted files. Output in up to 3 columns shows lines present in first file only, present in both, and/or present in second only.

□ CMP — Compare two files and report disagreeing bytes.

□ GREP — Print all lines in a file that satisfy a pattern of the kind used in the editor ED.
• May print all lines that fail to match.
• May print count of hits.
• Usable as a filter.

□ WC — Count the lines and "words" (blank-separated strings) in a file.
• Usable as a filter.

□ TYPO — Find typographical errors. Statistically analyzes all the words in a text, weeds out several thousand familiar ones, and publishes the rest sorted so that the most improbably spelled ones tend to come to the top of the list.

□ GSI — Simulate Model 37 Teletype facilities on GSI-300, DASI and other Diablo-mechanism terminals.
• Gives half-line and reverse platen motions.
• Approximates Greek letters and other special characters by overstriking.
• Usable as a filter.

□ COL — Canonicalize files with reverse line feeds for one-pass printing.
• Usable as a filter.

4 Novelties

Source code for game-playing programs is not distributed.

□ SPEAK — Driver for Vocal Interface's VOTRAX speech synthesizer. Reads input text and utters it.
- Associative memory allows pronunciation rules for whole words or word fragments to be added, changed, deleted or queried.
- Can use different memories for different languages.
- Usable as a filter to make the output of any other program audible.

□ CHESS — This chess-playing program scored 1-2-1 and 3-0-1 in the 1973 and 1974 Computer Chess Championships.

□ BJ — A blackjack dealer.

□ CUBIC — An accomplished player of 4×4×4 tic-tac-toe.

□ MOO — A fascinating number-guessing game.

□ CAL — Prints a calendar of specified month and year.

□ UNITS — Converts amounts between different scales of measurement. Knows hundreds of units. For example, how many km/sec is a parsec/megayear?

□ TTT — A tic-tac-toe program that learns. It never makes the same mistake twice.

□ QUIZ — Tests your knowledge of Shakespeare, Presidents, capitals, etc.

□ WUMP — Hunt the wumpus, thrilling search in a dangerous cave.

5 Typesetting

This software is distributed separately as an enhancement to UNIX.

5.1 Formatters

High programming skill is required to exploit the formatting capabilities of these programs, although unskilled personnel can easily be trained to enter documents according to canned formats. Terminal-oriented and typesetter-oriented formatters are sufficiently compatible that it is usually possible to define interchangeable formats.

□ NROFF — Advanced typesetting for terminals. Style similar to ROFF, but capable of much more elaborate feats of formatting, at a price in ease of use.
- All ROFF capabilities available or definable.
- Completely definable page format keyed to dynamically planted "interrupts" at specified lines.
- Maintains several separately definable typesetting environments (e.g. one for body text, one for footnotes, and one for unusually elaborate headings).
- Arbitrary number of output pools can be combined at will.
- Macros with substitutable arguments, and macros invocable in mid-line.
- Computation and printing of numerical quantities.
- Conditional execution of macros.
- Tabular layout facility.
- Multicolumn output on terminals capable of reverse line feed, or through the postprocessor COL.
- Usable as a filter

□ Manual — Printed manual for NROFF.

□ TROFF — Advanced phototypesetting for the Graphic Systems System/1. Provides facilities like NROFF, augmented as follows. This Summary was typeset by TROFF.
- Vocabulary of several 102-character fonts (4 simultaneously) in 15 sizes.
- Positions expressible in inches, centimeters, ems, points, machine units or arithmetic combinations thereof.
- Access to character-width computation for unusually difficult layout problems.
- Overstrikes, built-up brackets, horizontal and vertical line drawing.
- Dynamic relative or absolute positioning and size selection, globally or at the character level.
- Terminal output for rough sampling of the product, usually needs a wide platen. Not a substitute for NROFF.
- Usable as a filter.

□ Manuals — Printed manual and tutorial for TROFF.

□ EQN — A mathematical typesetting preprocessor for TROFF. Translates easily readable formulas, either in-line or displayed, into detailed typesetting instructions. Formulas are written in a style like this:

sigma sup 2 ~=~ 1 over N sum from i=1 to N (x sub i - x bar) sup 2

which produces this:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N} (x_i - \bar{x})^2$$

- Automatic calculation of size changes for subscripts, sub-subscripts, etc.
- Full vocabulary of Greek letters, such as ‘gamma’, ‘GAMMA’.
- Automatic calculation of large bracket sizes.
- Vertical ‘‘piling’’ of formulae for matrices, conditional alternatives, etc.
- Integrals, sums, etc. with arbitrarily complex limits.
- Diacriticals: dots, double dots, hats, bars.
- Easily learned by nonprogrammers and mathematical typists.
- Usable as a filter.

□ Manual — Printed manual for EQN.

□ NEQN — A mathematical typesetting preprocessor for NROFF. Prepares formulas for display on Model 37 Teletypes with half-line functions and 128-character font.
- For Diablo-mechanism terminals, filter output through GSI.
- Same facilities as EQN within graphical capability of terminal.

□ TBL — A preprocessor for NROFF/TROFF that translates simple descriptions of table layouts and contents into detailed typesetting instructions.
- Computes column widths.
- Handles left- and right-justified columns, centered columns and decimal-point alignment.
- Places column titles.

□ MS — A standardized manuscript layout for use with NROFF/TROFF.
- Page numbers and draft dates.
- Cover sheet and title page.
- Automatically numbered subheads.
- Footnotes.
- Single or double column.
- Paragraphing, display and indentation.
- Numbered equations.

5.2 UNIX Programmer's Manual

□ MAN — Print specified manual section on your terminal.

□ Manual — Machine-readable version of the UNIX Programmer's Manual.
- System overview.
- All commands.
- All system calls.
- All subroutines in assembler, C and FORTRAN libraries.
- All devices and other special files.
- Formats of file system and kinds of files known to system software.
- Boot procedures.

May, 1975

* DEC, PDP and DECtape are registered trademarks of Digital Equipment Corporation. VOTRAX is a registered trademark of Vocal Interface Division, Federal Screw Works.