# Appendices

# Appendix A: The C Shell

Csh is an alternate command language interpreter. It incorporates good features of other shells and a history mechanism. most of the features unique to csh are designed more for the interactive XENIX user, although some features of other shells have been incorporated to make writing shell procedures easier.

XENIX users who have read a general introduction to the system will find a valuable basic explanation of the shell here. Simple terminal interaction with csh is possible after reading just the first section of this document. The second section describes the capabilities you can explore after you have begun to become acquainted with the Cshell. Later sections introduce features which are useful, but not necessary for all users of the shell.

The final section of this chapter lists special characters of the Cshell.

A shell is a command language interpreter. Csh is the name of one particular command interpreter on XENIX. The primary purpose of csh is to translate command lines typed at a terminal into system actions, such as invocations of other programs. Csh is a user program just like any you might write.

This document provides a full description of all features of the shell and is a final reference for all questions.

## A.1 Details on the shell for terminal users

### A.1.1 Shell startup and termination

When you login, the shell is started by the system in your home directory and begins by reading commands from a file .cshrc in this directory. All shells which you may start during your terminal session will read from this file. We will later see what kinds of commands are usefully placed there. For now we need not have this file and the shell does not complain about its absence.

A login shell, executed after you login to the system, will, after it reads commands from .cshrc, read commands from a file .login also in your home directory. This file contains commands which you wish to do each time you login to the XENIX system. A typical .login file might look something like this:

```
set ignoreeof
set mail=(/usr/spool/mail/bill)
echo "${prompt}users" ; users
alias ts \
        'set noglob ; eval `tset -s -m dialup:c100rv4pna \
        -m plugboard:?hp2621nl *`';
ts; stty intr ^C kill ^U crt
set time=15 history=10
if (-e $mail) then
        echo "${prompt}mail"
        mail
endif
```

This above file contains several commands to be executed by XENIX at each login. The first is a *set* command which is interpreted directly by the shell. It sets the shell variable *ignoreeof* which shields the shell from log off if <CONTROL-D> is hit. Instead of <CONTROL-D>, the logout command is used to log off the system. By setting the *mail* variable, the shell is notified that it is to watch for incoming mail and to notifiy the user if new mail arrives.

Next the shell variable *time* is set to "15" causing the shell to automatically print out statistics lines for commands which execute for at least 15 seconds of CPU time. The variable "history" is set to 10 indicating that the shell will remember the last 10 commands types in its *history list*, (described later).

Next, an *alias*, "ts", is created which executes a *tset*(1) command setting up the modes of the terminal. The parameters to *tset* indicate the kinds of terminal normally used when not on a hardwired port. Then "ts" is executed, and the *stty* command is used to change the interrupt character to <CONTROL-C> and the line kill character to <CONTROL-U>.

Finally, if my mailbox file exists, then I run the **mail** program to process my mail.

When the **mail** programs finish, the shell will finish processing my *.login* file and begin reading commands from the terminal, prompting for each with "% ". When I log off (by giving the *logout* command) the shell will print "logout" and execute commands from the file *.logout* if it exists in my home directory. After that, the shell will terminate and XENIX will log me off the system. If the system is not going down, I will receive a new login message. In any case, after the logout message the shell is committed to terminating and will take no further input from my terminal.

### A.1.2 Shell variables

The shell maintains a set of variables. We saw above the variables history and time which had the values 10 and 15. In fact, each shell variable has as value an array of zero or more strings. Shell variables may be assigned values by the set command. It has several forms, the most useful of which was given above and is

```
set name=value
```

Shell variables may be used to store values which are to be used in commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those which the shell itself refers to. By changing the values of these variables one can directly affect the behavior of the shell.

One of the most important variables is the variable path. This variable contains a sequence of directory names where the shell searches for commands. The set command with no arguments shows the value of all variables currently defined (we usually say set) in the shell. The default value for path will be shown by set to be

```
% set
argv    ()
cwd     /usr/bill
home    /usr/bill
path    (. /bin /usr/bin)
prompt  %
shell   /bin/csh
status  0
term    c100rv4pna
user    bill
%
```

This output indicates that the variable path points to the current directory indicated by dot (.) and then /bin, and /usr/bin. Your own local commands may be in dot. Normal XENIX commands live in /bin and /usr/bin.

Often a number of locally developed programs on the system live in the directory /usr/local. If we wish that all shells which we invoke to have access to these new programs we can place the command

```
set path=(. /bin /usr/bin /usr/local)
```

in our file .cshrc in our home directory. Try doing this and then logging out and back in. Then type

```
set
```

again to see that the value assigned to path has changed.

You should be aware that the shell examines each directory that you insert into your path and determines which commands are contained there. Except for the current directory, dot (.), which the shell treats specially, this means that if commands are added to a directory in your search path after you have started the shell, they will not necessarily be found. If you wish to use a command which has been added in this way, you should give the command

```
rehash
```

to the shell, which causes it to recompute its internal table of command locations, so that it will find the newly added command. Since the shell has to look in the current directory . on each command, placing it at the end of the path specification usually works equivalently and reduces overhead.

Other useful built in variables are the variable home which shows your home directory, cwd which contains your current working directory, the variable ignoreeof which can be set in your .login file to tell the shell not to exit when it receives an end-of-file from a terminal (as described above). The variable "ignoreeof" is one of several variables which the shell does not care about the value of, only whether they are set or unset. Thus to set this variable you simply do

```
set ignoreeof
```

and to unset it do

```
unset ignoreeof
```

These give the variable "ignoreeof" no value, but none is desired or required.

Finally, some other built-in shell variables of use are the variables noclobber and mail. The metasyntax

```
>filename
```

which redirects the standard output of a command will overwrite and destroy the previous contents of the named file. In this way you may accidentally overwrite a file which is valuable. If you would prefer that the shell not overwrite files in this way you can

```
set noclobber
```

in your .login file. Then trying to do

```
date > now
```

would cause a diagnostic if "now" existed already. You could type

```
date >! now
```

if you really wanted to overwrite the contents of now. The ">!" is a special metasyntax indicating that overwriting or "clobbering" the file is ok. (The space between the exclamation (!) and the word "now" is critical here, as "!now" would be an invocation of the history mechanism, and have a totally different effect.)

### A.1.3 The Shell's History List

The shell can maintain a history list into which it places the words of previous commands. It is possible to use a notation to reuse commands or words from commands in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

The following figure gives a sample session involving typical usage of the history mechanism of the shell.

```
% cat bug.c
main()

{
        printf("hello);
}
% cc !$
cc bug.c
"bug.c", line 4: newline in string or char constant
"bug.c", line 5: syntax error
% ed !$
ed bug.c
29
4s/);/"&/p
        printf("hello");
w
30
q
% !c
cc bug.c
% a.out
hello% !e
ed bug.c
30
4s/lo/lo\\n/p
        printf("hello\n");
w
32
q
% !c -o bug
cc bug.c -o bug
% size a.out bug
a.out: 2784+364+1028 = 4176b = 0x1050b
bug: 2784+364+1028 = 4176b = 0x1050b
% ls -l !*
ls -l a.out bug
-rwxr-xr-x 1 bill       3932 Dec 19 09:41 a.out
-rwxr-xr-x 1 bill       3932 Dec 19 09:42 bug
% bug
hello
% num bug.c | spp
spp: Command not found.
% ^spp^ssp
num bug.c | ssp
    1   main()
    3   {
    4           printf("hello\n");
    5   }
% !! | lpr
num bug.c | ssp | lpr
%
```

In this example, we have a very simple C program which has a bug (or two) in it in the file bug.c, which we **cat** out on our terminal. We then try to run the C compiler on it, referring to the file again as "!$", meaning the last argument to the previous command. Here the exclamation mark (!) is the history mechanism invocation metacharacter, and the dollar sign ($) stands for the last argument, by analogy to the dollar sign in the editor which stands for the end of the line. The shell echoed the command, as it would have been typed without use of the history mechanism, and then executed it. The compilation yielded error diagnostics, so we now run the editor on the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as "!c", which repeats the last command which started with the letter "c". If there were other commands starting with "c" done recently we could have said "!cc" or even "!cc:p" which would have printed the last command starting with "cc" without executing it.

After this recompilation, we ran the resulting a.out file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra "-o bug" telling the compiler to place the resultant binary in the file bug rather than a.out. In general, the history mechanisms may be used anywhere in the formation of new commands and other characters may be placed before and after the substituted commands.

We then ran the **size** command to see how large the binary program images we have created were, and then an "ls -l" command with the same argument list, denoting the argument list "*". Finally, we ran the program bug to see that its output is indeed correct.

To make a numbered listing of the program, we ran the **num** command on the file bug.c. In order to filter out blank lines in the output of **num** we ran the output through the filter ssp, but misspelled it as "spp". To correct this we used a shell substitute, placing the old text and new text between up arrow (^) characters. This is similar to the substitute command in the editor. Finally, we repeated the same command with "!!", but sent its output to the line printer.

There are other mechanisms available for repeating commands. The history command prints out a number of previous commands with numbers by which they can be referenced. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in the C shell manual pages in the XENIX Programmers Manual.

### A.1.4 Aliases

The shell has an _alias_ mechanism which can be used to make transformations on input commands. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained also using shell command files, but these take place in another instance of the shell and cannot directly affect the current shells environment or involve commands such as _cd_ which must be done in the current shell.

As an example, suppose that there is a new version of the mail program on the system called "newmail" you wish to use, rather than the standard mail program which is called "mail". If you place the shell command

```
alias mail newmail
```

in your ._cshrc_ file, the shell will transform an input line of the form

```
mail bill
```

into a call on "newmail". More generally, suppose we wish the command **ls** to always show sizes of files, that is to always do -s. We can do

```
alias ls ls -s
```

or even

```
alias dir ls -s
```

creating a new command named "dir" which does an "ls -s". If we say

```
dir ~bill
```

then the shell will translate this to

```
ls -s /usr/bill
```

Thus the _alias_ mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases which contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

```
alias cd 'cd \!* ; ls '
```

would do an ls command after each change directory cd command. We enclosed the entire alias definition in single quotes (') to prevent most substitutions from occurring and the semicolon (;) from being recognized as a metacharacter. The exclamation mark (!) is escaped with a backslash (\) to prevent it from being interpreted when the alias command is typed in. The "\!*" here substitutes the entire argument list to the pre-aliasing cd command, without giving an error if there were no arguments. The semicolon (;) separating commands is used here to indicate that one command is to be done and then the next. Similarly the definition

```
alias whois 'grep \!^ /etc/passwd'
```

defines a command which looks up its first argument in the password file.

**Warning:** The shell currently reads the .cshrc file each time it starts up. If you place a large number of commands there, shells will tend to start slowly. You should try to limit the number of aliases you have to a reasonable number... 10 or 15 is reasonable, 50 or 60 will cause a noticeable delay in starting up shells, and make the system seem sluggish when you execute commands from within the editor and other programs.

### A.1.5 More redirection; >> and >&

There are a few more notations useful to the terminal user which have not been introduced yet. In addition to the standard output, commands also have a diagnostic output which is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally desirable to direct the diagnostic output along with the standard output. For instance if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can type

```
command >& file
```

The ">&" here tells the shell to route both the diagnostic output and the standard output into file. Similarly you can give the command

```
command |& lpr
```

to route both standard and diagnostic output through the pipe to the line printer daemon _lpr_. A command form

```
command >&! file
```

exists, and is used when _noclobber_ is set and _file_ already exists.

Finally, it is possible to use the form

```
command >> file
```

to place output at the end of an existing file. If _noclobber_ is set, then an error will result if _file_ does not exist, otherwise the shell will create _file_ if it doesn't exist. A form

```
command >>! file
```

makes it not be an error for file to not exist when _noclobber_ is set.

### A.1.6 Jobs: Background and Foreground

When one or more commands are typed together as a pipeline or as a sequence of commands separated by semicolons, a single _job_ is created by the shell consisting of these commands together as a unit. Single commands without pipes or semicolons create the simplest jobs. Usually, every line typed to the shell creates a job. Some lines that create jobs (one per line) are

```
sort < data
ls -s | sort -n | head -5
mail harold
```

If the ampersand metacharacter (&) is typed at the end of the commands, then the job is started as a _background_ job. This means that the shell does not wait for it to complete but immediately prompts and is ready for another command. The job runs _in the background_ at the same time that normal jobs, called _foreground_ jobs, continue to be read and executed by the shell one at a time. Thus

```
du > usage &
```

would run the _du_ program, which reports on the disk usage of your working directory (as well as any directories below it), put the output into the file _usage_ and return immediately with a prompt for the next command without out waiting for **du** to finish. The du program would continue executing in the background until it finished, even though you can type and execute more commands in the mean time. Background jobs are unaffected by any signals from the keyboard like the <INTERRUPT> or <QUIT> signals mentioned earlier.

The kill command terminates a background job immediately. It may be given process numbers as arguments, as printed by ps.

### A.1.7 Useful Built-In Commands

We now give a few of the useful built-in commands of the shell describing how they are used.

The **alias** command described above is used to assign new aliases and to show the existing aliases. With no arguments it prints the current aliases. It may also be given only one argument such as

```
alias ls
```

to show the current alias for, e.g., **ls**.

The **echo** command prints its arguments. It is often used in shell scripts or as an interactive command to see what filename expansions will produce.

The **history** command will show the contents of the history list. The numbers given with the history events can be used to reference previous events which are difficult to reference using the contextual mechanisms introduced above. There is also a shell variable called prompt. By placing an exclamation mark (!) in its value the shell will there substitute the number of the current command in the history list. You can use this number to refer to this command in a history substitution. Thus you could

```
set prompt='\! % '
```

Note that the exclamation mark (!) had to be escaped here even within backslashes.

The **logout** command can be used to terminate a login shell which has ignoreeof set.

The **rehash** command causes the shell to recompute a table of where commands are located. This is necessary if you add a command to a directory in the current shell's search path and wish the shell to find it, since otherwise the hashing algorithm may tell the shell that the command wasn't in that directory when the hash table was computed.

The **repeat** command can be used to repeat a command several times. Thus to make 5 copies of the file _one_ in the file _five_ you could do

```
repeat 5 cat one >> five
```

The **setenv** command can be used to set variables in the environment. Thus

```
setenv TERM adm3a
```

sets the value of the environment variable TERM to "adm3a". A user program _printenv_ exists which will print out the environment. It might then show:

```
% printenv
HOME=/usr/bill
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
%
```

The **source** command can be used to force the current shell to read commands from a file. Thus

```
source .cshrc
```

can be used after editing in a change to the _.cshrc_ file which you wish to take effect before the next time you login.

The **time** command can be used to cause a command to be timed no matter how much CPU time it takes. Thus

```
% time cp /etc/rc /usr/bill/rc
0.0u 0.1s 0:01 8%
% time wc /etc/rc /usr/bill/rc
     52    178   1347 /etc/rc
     52    178   1347 /usr/bill/rc
    104    356   2694 total
0.1u 0.1s 0:00 13%
%
```

indicates that the _cp_ command used a negligible amount of user time (u) and about 1/10th of a second system time (s); the elapsed time was 1 second (0:01). The word count command, **wc**, on the other hand, used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage "13%" indicates that over the period when it was active the command **wc** used an average of 13 percent of the available CPU cycles of the machine.

The **unalias** and **unset** commands can be used to remove aliases and variable definitions from the shell, and **unsetenv** removes variables from the environment.

This concludes the basic discussion of the shell for terminal users. There are more features of the shell to be discussed here, and all features of the shell are discussed in its manual pages. One useful feature which is discussed later is the **foreach** built-in command which can be used to run the same command sequence with a number of different arguments.

## A.2 Shell Control Structures and Command Scripts

It is possible to place commands in files and to cause shells to be invoked to read and execute commands from these files, which are called _shell scripts_. We here detail those features of the shell useful to the writers of such scripts.

It is important to first note what shell scripts are _not_ useful for. There is a program called _make_ which is very useful for maintaining a group of related files or performing sets of operations on related files. For instance a large program consisting of one or more files can have its dependencies described in a _makefile_ which contains definitions of the commands used to create these different files when changes occur. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily, and most appropriately placed in this _makefile_. This format is superior and preferable to maintaining a group of shell procedures to maintain these files.

Similarly when working on a document a _makefile_ may be created which defines how different versions of the document are to be created and which options of _nroff_ or _troff_ are appropriate.

### A.2.1 Invocation and the argv variable

A _csh_ command script may be interpreted by saying

```
% csh script ...
```

where _script_ is the name of the file containing a group of *csh* commands and "..." is replaced by a sequence of arguments. The shell places these arguments in the variable *argv* and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other shell variables.

If you make the file *script* executable by doing

```
chmod 755 script
```

and place a shell comment at the beginning of the shell script (i.e. begin the file with a pound sign (#)) then a /*bin*/*csh* will automatically be invoked to execute *script* when you type

```
script
```

If the file does not begin with a pound sign (#) then the standard shell /*bin*/*sh* will be used to execute it. This allows you to convert your older shell scripts to use *csh* at your convenience.

### A.2.2 Variable substitution

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism know as *variable substitution* is done on these words. Keyed by the dollar sign ($), this substitution replaces the names of variables by their values. Thus

```
echo $argv
```

when placed in a command script would cause the current value of the variable *argv* to be echoed to the output of the shell script. It is an error for *argv* to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
$?name
```

expands to 1 if name is *set* or to 0 if name is not *set*. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable _name_. Thus

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable which has several values. Thus

```
$argv[1]
```

gives the first component of _argv_ or in the example above "a". Similarly

```
$argv[$#argv]
```

would give "c", and

```
$argv[1-2]
```

would give:

```
a b
```

Other notations useful in shell scripts are

$_n_

where _n_ is an integer as a shorthand for

$argv[_n_]

the _nth_ parameter and

```
$*
```

which is a shorthand for

```
$argv
```

The form

```
$$
```

expands to the process number of the current shell. Since this process number is unique in the system it can be used in generation of unique temporary file names. The form

```
$<
```

is quite special and is replaced by the next line of input read from the shell's standard input (not the script it is reading). This is useful for writing shell scripts that are interactive, reading commands from the terminal, or even writing a shell script that acts as a filter, reading lines from its input file. Thus the sequence

```
echo 'yes or no?\c'
set a=($<)
```

would write out the prompt "yes or no?" without a newline and then read the answer into the variable _a._ In this case "$#a" would be 0 if either a blank line or <CONTROL-D> was typed.

One minor difference between "$_n_" and "$argv[_n_]" should be noted here. The form "$argv[_n_]" will yield an error if _n_ is not in the range "1-$#argv" while "$n" will never yield an out of range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form "n-"; if there are less than "n" components of the given variable then no words are substituted. A range of the form "m-n" likewise returns an empty vector without giving an error when "m" exceeds the number of elements of the given variable, provided the subscript "n" is in range.

### A.2.3 Expressions

In order for interesting shell scripts to be constructed it must be possible to evaluate expressions in the shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations "==" and "!=" compare strings and the operators "&&" and "||" implement the boolean AND and OR operations. The special operators "=~" and "!~" are similar to "==" and "!=" except that the string on the right side can have pattern matching characters (like *, ? or [ and ]) and the test is whether the string on the left matches the pattern on the right.

The shell also allows file enquiries of the form

```
-? filename
```

where question mark (?) is replaced by a number of single characters. For instance the expression primitive

```
-e filename
```

tell whether the file filename exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or has non-zero length.

It is possible to test whether a command terminates normally, by a primitive of the form

```
{ command }
```

which returns true, i.e. 1 if the command succeeds exiting normally with exit status 0, or 0 if the command terminates abnormally or with exit status non-zero. If more detailed information about the execution status of a command is required, it can be executed and the variable "$status" examined in the next command. Since "$status" is set by every command, it is very transient. It can be saved if it is inconvenient to use it only in the single immediately following command.

For a full list of expression components available see the manual section for the shell.

### A.2.4 Sample shell script

A sample shell script which makes use of the expression mechanism of the shell and some of its control structure follows:

```
% cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

        if ($i !~ *.c) continue  # not a .c file so do nothing

        if (! -r ~/backup/$i:t) then
                echo $i:t not in backup... not cp\'ed
                continue
        endif

        cmp -s $i ~/backup/$i:t # to set $status

        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```

This script makes use of the **foreach** command, which causes the shell to execute the commands between the **foreach** and the matching **end** for each of the values given between parentheses with the named variable, in this case "i" set to successive values in the list. Within this loop we may use the command **break** to stop executing the loop and **continue** to prematurely terminate one iteration and begin the next. After the **foreach** loop the iteration variable (_i_ in this case) has the value at the last iteration.

We set the variable _noglob_ here to prevent filename expansion of the members of _argv_. This is a good idea, in general, if the arguments to a shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a "$" variable expansion, but this is harder and less reliable.

The other control construct used here is a statement of the form

```
if ( expression ) then
        command
        ...
endif
```

The placement of the keywords here is _not_ flexible due to the current implementation of the shell. The following two formats are not acceptable to the shell:

```
if (expression) # Won't work!
then
        command
        ...
endif
```

and

```
if (expression) then command endif # Won't work
```

The shell does have another form of the if statement of the form

```
if ( expression ) command
```

which can be written

```
if ( expression ) \
        command
```

Here we have escaped the newline for the sake of appearance. The command must not involve "|", "&" or ";" and must not be another control command. The second form requires the final backslash (\) to **immediately** precede the end-of-line.

The more general **if** statements above also admit a sequence of else-if pairs followed by a single **else** and an **endif**, e.g.:

```
if ( expression ) then
        commands
else if (expression ) then
        commands
...

else
        commands
endif
```

Another important mechanism used in shell scripts is the colon (:) modifier. We can use the modifier ":r" here to extract the root of a filename or `:e' to extract the extension. Thus if the variable i has the value /mnt/foo.bar then

```
% echo $i $i:r $i:e
/mnt/foo.bar /mnt/foo bar
%
```

shows how the ":r" modifier strips off the trailing ".bar" and the the ":e" modifier leaves only the "bar". Other modifiers will take off the last component of a pathname leaving the head ":h" or all but the last component of a pathname leaving the tail ":t". These modifiers are fully described in the csh(1S) manual pages in the XENIX Reference manual. It is also possible to use the command substitution mechanism described in the next major section to perform modifications on strings to then reenter the shells environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the colon (:) modification mechanism. (It is also important to note that the current implementation of the shell limits the number of colon modifiers on a "$" substitution to 1. Thus

```
% echo $i $i:h:t
/a/b/c /a/b:t
%
```

does not do what one would expect.)

Finally, we note that the pound sign character (#) lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a pound sign are discarded by the shell. This character can be quoted using "'" or "\" to place it in an argument word.

### A.2.5 Other control structures

The shell also has control structures **while** and **switch** similar to those of C. These take the forms

```
while ( expression )
        commands
end
```

and

```
switch ( word )

case str1:
        commands
        breaksw

  ...

case strn:
        commands
        breaksw

default:
        commands
        breaksw

endsw
```

For details see the manual section for *csh*(1S). C programmers should note that we use **breaksw** to exit from a **switch** while **break** exits a **while** or **foreach** loop. A common mistake to make in cshell scripts is to use **break** rather than **breaksw** in switches.

Finally, cshell allows a **goto** statement, with labels looking like they do in C, i.e.:

```
loop:
        commands
        goto loop
```

### A.2.6 Supplying input to commands

Commands run from shell scripts receive by default the standard input of the shell which is running the script. This is different from previous shells running under XENIX. It allows shell scripts to fully participate in pipelines, but mandates extra notation for commands which are to take inline data.

Thus we need a metanotation for supplying inline data to commands in shell scripts. As an example, consider this script which runs the editor to delete leading blanks from the lines in each argument file

```
% cat deblank
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/^[ ]*//
w
q
'EOF'
end
%
```

The notation "<< 'EOF'" means that the standard input for the ed command is to come from the text in the shell script file up to the next line consisting of exactly "'EOF'". The fact that the EOF is enclosed in single quotes ('), i.e. quoted, causes the shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the "<<" which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form "1,$" in our editor script we needed to insure that this dollar sign was not variable substituted. We could also have insured this by preceding the dollar sign ($) with a backslash (\), i.e.:

```
1,\$s/^[ ]*//
```

but quoting the EOF terminator is a more reliable way of achieving the same thing.

### A.2.7 Catching interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. We can then do

```
onintr label
```

where label is a label in our program. If an interrupt is received the shell will do a "goto label" and we can remove the temporary files and then do an exit command (which is built in to the shell) to exit from the shell script. If we wish to exit with a non-zero status we can do

```
exit(1)
```

e.g. to exit with status 1.

### A.2.8 Other Features

There are other features of the shell useful to writers of shell procedures. The verbose and echo options and the related **-v** and **-x** command line options can be used to help trace the actions of the shell. The **-n** option causes the shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that csh will not execute shell scripts which do not begin with the pound sign character (#), that is shell scripts that do not begin with a comment. Similarly, the /bin/sh on your system may well defer to csh to interpret shell scripts which begin with the pound sign (#). This allows shell scripts for both shells to live in harmony.

There is also another quotation mechanism using the quotation mark ("), which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as the single quote (') does.

## A.3 Loops At The Terminal

It is occasionally useful to use the **foreach** control structure at the terminal to aid in performing a number of similar commands. For instance, if there were three shells in use on a particular system, /bin/sh, /bin/nsh, and /bin/csh, you could count the number of persons using each shell by using the following commands:

```
% grep -c csh$ /etc/passwd
5
% grep -c nsh$ /etc/passwd
3
% grep -c -v sh$ /etc/passwd
20
%
```

Since these commands are very similar we can use **foreach** to do this more easily.

```
% foreach i ('sh$' 'csh$' '-v sh$')
? grep -c $i /etc/passwd
? end
5
3
20
%
```

Note here that the shell prompts for input with "? " when reading the body of the loop.

Very useful with loops are variables which contain lists of filenames or other words. You can, for example, do

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
%
```

The set command here gave the variable a a list of all the filenames in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within back quote characters (`) is converted by the shell to a list of words. You can also place the quoted string within double quote characters (") to take each (non-empty) line as a component of the variable. This prevents the lines from being split into words at blanks and tabs. A modifier ":x" exists which can be used later to expand each component of the variable into another variable by splitting the original variable into separate words at embedded blanks and tabs.

## A.4 Braces { ... } in argument expansion

Another form of filename expansion, alluded to before involves the characters, "{" and "}". These characters specify that the contained strings, separated by commas (,) are to be consecutively substituted into the containing characters and the results expanded left to right. Thus

```
A{str1,str2,...strn}B
```

expands to

```
Astr1B Astr2B ... AstrnB
```

This expansion occurs before the other filename expansions, and may be applied recursively (i.e. nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be

```
mkdir ~/{hdrs,retrofit,csh}
```

to make subdirectories _hdrs_, _retrofit_ and _csh_ in your home directory. This mechanism is most useful when the common prefix is longer than in this example, i.e.

```
chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

## A.5 Command substitution

A command enclosed in back quotes (`) is replaced, just before filenames are expanded, by the output from that command. Thus, it is possible to do

```
set pwd=`pwd`
```

to save the current directory in the variable _pwd_ or to do

```
vi `grep -l TRACE *.c`
```

to run the editor **vi** supplying as arguments those files whose names end in ".c" which have the string "TRACE" in them. Command expansion also occurs in input redirected with "<<" and within quotations ("). Refer to _csh_(1S) in the _XENIX Reference_ manual for more information.

## A.6 Other Details Not Covered Here

In particular circumstances it may be necessary to know the exact nature and order of different substitutions performed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed fully in its manual section.

The shell has a number of command line option flags mostly of use in writing XENIX programs and debugging shell scripts. See _csh_(1S) in the _XENIX Reference Manual_ for a list of these options.

## A.7 Special Characters

The following table lists the special characters of _csh_ and the XENIX system. A number of these characters also have special meaning in expressions. See the _csh_ manual section for a complete list.

### Syntactic metacharacters

| | |
|---|---|
| ; | Separates commands to be executed sequentially |
| \| | Separates commands in a pipeline |
| ( ) | Brackets expressions and variable values |
| & | Follows commands to be executed without waiting for completion |

### Filename metacharacters

| | |
|---|---|
| / | Separates components of a file's pathname |
| ? | Expansion character matching any single character |
| * | Expansion character matching any sequence of characters |
| [ ] | Expansion sequence matching any single character from a set of characters |
| ~ | Used at the beginning of a filename to indicate home directories |
| { } | Used to specify groups of arguments with common parts |

### Quotation metacharacters

| | |
|---|---|
| \ | Prevents meta-meaning of following single character |
| ' | Prevents meta-meaning of a group of characters |
| " | Like ', but allows variable and command expansion |

### Input/output metacharacters

| | |
|---|---|
| < | Indicates redirected input |
| > | Indicates redirected output |

### Expansion/Substitution metacharacters

| | |
|---|---|
| $ | Indicates variable substitution |
| ! | Indicates history substitution |
| : | Precedes substitution modifiers |
| ^ | Used in special forms of history substitution |
| ` | Indicates command substitution |

### Other metacharacters

| | |
|---|---|
| # | Begins scratch file names; indicates shell comments |
| - | Prefixes option (flag) arguments to commands |
| % | Prefixes job name specifications |

## APPENDIX B: M4 - A Macro Processor

M4 is the name of the XENIX macro processor. Macro processors are used to define and to process specially defined strings of characters (called macros). By defining a set of macros to be processed by M4, a programming language can be enhanced to make it:

1. More structured
2. More readable
3. More appropriate for a particular application

The #define statement in C and the analogous define in Ratfor are examples of the basic facility provided by any macro processor -- replacement of text by other text.

Besides the straightforward replacement of one string of text by another, a macro processor provides:

- Macros with arguments
- Conditional macro expansions
- Arithmetic expressions
- File manipulation facilities
- String processing functions

The basic operation of M4 is to copy its input to its output. As the input is read, , each alphanumeric "token" (that is, string of letters and digits) is checked. If it is the name of a macro, then the name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input it is rescanned by M4. Macros may be called with arguments, in which case the arguments are collected and substituted into the right places in the defining text before M4 rescans the text.

M4 provides a collection of about twenty built-in macros which perform various operations. In addition, the user can define new macros. Built-ins and user-defined macros work exactly the same way, except that some of the built-in macros have side effects on the state of the process.

## B.1 Usage

To invoke M4, type:

```
m4 [files]
```

Each argument file is processed in order. If there are no arguments, or if an argument is a dash (-), the standard input is read at that point. The processed text is written to the standard output.

```
m4 [files] >outputfile
```

## B.2 Defining Macros

The primary built-in function of M4 is *define*, which is used to define new macros. The input

```
define(name, stuff)
```

causes the string *name* to be defined as *stuff*. All subsequent occurrences of *name* will be replaced by *stuff*. *Name* must be alphanumeric and must begin with a letter (the underscore _ counts as a letter). *stuff* is any text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example,

```
define(N, 100)
 ...
if (i > N)
```

defines *N* to be 100, and uses this "symbolic constant" in a later *if* statement.

The left parenthesis must immediately follow the word *define*, to signal that *define* has arguments. If a macro or built-in name is not followed immediately by "(", it is assumed to have no arguments. This is the situation for *N* above; it is actually a macro with no arguments, and thus when it is used there need be no (...) following it.

You should also notice that a macro name is only recognized as such if it appears surrounded by non-alphanumerics. For example, in

```
define(N, 100)
 ...
if (NNN > 100)
```

the variable NNN is absolutely unrelated to the defined macro N, even though it contains a lot of N's.

Things may be defined in terms of other things. For example,

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

defines both M and N to be 100.

What happens if N is redefined? Or, to say it another way, is M defined as N or as 100? In M4, the latter is true -- M is 100, so even if N subsequently changes, M does not.

This behavior arises because M4 expands macro names into their defining text as soon as it possibly can. Here, that means that when the string N is seen as the arguments of define are being collected, it is immediately replaced by 100; it's just as if you had said

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

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

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

Now M is defined to be the string N, so when you ask for M later, you will always get the value of N at that time (because the M will be replaced by N which, in turn, will be replaced by 100).

## B.3 Quoting

The more general solution is to delay the expansion of the arguments of define by quoting them. Any text surrounded by the single quotes ` and ' is not expanded immediately, but has the quotes stripped off. If you say

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

the quotes around the N are stripped off as the argument is being collected, but they have served their purpose, and M is defined as the string N, not 100. The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros. If you want the word _define_ to appear in the output, you have to quote it in the input, as in

```
`define' = 1;
```

As another instance of the same thing, which is a bit more surprising, consider redefining _N_:

```
define(N, 100)
  ...
define(N, 200)
```

Perhaps regrettably, the _N_ in the second definition is evaluated as soon as it's seen; that is, it is replaced by 100, so it's as if you had written

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

This statement is ignored by M4, since you can only define things that look like names, but it obviously doesn't have the effect you wanted. To really redefine _N_, you must delay the evaluation by quoting:

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

In M4, it is often wise to quote the first argument of a macro.

If the forward and backward quote characters (` and ') are not convenient for some reason, the quote characters can be changed with the built-in _changequote_. For example:

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

makes the new quote characters the left and right brackets. You can restore the original characters with just

```
changequote
```

There are two additional built-ins related to _define_. _undefine_ removes the definition of some macro or built-in:

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

removes the definition of _N_. Built-ins can be removed with _undefine_, as in

```
undefine(`define')
```

but once you remove one, you can never get it back.

The built-in ifdef provides a way to determine if a macro is currently defined. For instance, pretend that either the word xenix or unix is defined according to a particular implementation of a program. To perform operations according to which system you have you might say:

```
ifdef(`xenix', `define(system,1)' )
ifdef(`unix', `define(system,2)' )
```

Don't forget the quotes in the above example.

Ifdef actually permits three arguments: if the name is undefined, the value of ifdef is then the third argument, as in

```
ifdef(`xenix', on XENIX, not on XENIX)
```

## B.4 Arguments

So far we have discussed the simplest form of macro processing -- replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its define) any occurrence of $n will be replaced by the nth argument when the macro is actually used. Thus, the macro bump, defined as

```
define(bump, $1 = $1 + 1)
```

generates code to increment its argument by 1:

```
bump(x)
```

is

```
x = x + 1
```

A macro can have as many arguments as you want, but only the first nine are accessible, through $1 to $9. (The macro name itself is $0, although that is less commonly used.) Arguments that are not supplied are replaced by null strings, so we can define a macro cat which simply concatenates its arguments, like this:

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

Thus

```
cat(x, y, z)
```

is equivalent to

```
xyz
```

$*4* through $*9* are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus:

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

defines *a* to be *b*   *c*.

Arguments are separated by commas, but parentheses are counted properly, so a comma "protected" by parentheses does not terminate an argument. That is, in

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

there are only two arguments; the second is literally (*b*,*c*). And of course a bare comma or parenthesis can be inserted by quoting it.

## B.5 Arithmetic Built-ins

M4 provides two built-in functions for doing arithmetic on integers. The simplest is *incr*, which increments its numeric argument by 1. Thus, to handle the common programming situation where you want a variable to be defined as "one more than N", write

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

Then *N1* is defined as one more than the current value of *N*.

The more general mechanism for arithmetic is a built-in called *eval*, which is capable of arbitrary arithmetic on integers. It provides the following operators (in decreasing order of precedence):

```
unary + and -
** or ^ (exponentiation)
*  /  % (modulus)
+  -
==  !=  <  <=  >  >=
!       (not)
& or && (logical and)
| or || (logical or)
```

Parentheses may be used to group operations where needed. All the operands of an expression given to *eval* must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in *eval* is implementation dependent.

As a simple example, suppose we want *M* to be *2****N*+*1*. Then

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

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

## B.6 File Manipulation

You can include a new file in the input at any time by the built-in function *include*:

```
include(filename)
```

inserts the contents of *filename* in place of the *include* command. The contents of the file is often a set of definitions. The value of *include* (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in *include* cannot be accessed. To get some control over this situation, the alternate form *sinclude* can be used; *sinclude* ("silent include") says nothing and continues if it can't access the file.

It is also possible to divert the output of M4 to temporary files during processing, and output the collected material upon command. M4 maintains nine of these diversions, numbered 1 through 9. If you say

```
divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as *n*. Diverting to this file is stopped by another *divert* command; in particular, *divert* or *divert*(*0*) resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

```
undivert
```

brings back all diversions in numeric order, and *undivert* with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of *undivert* is *not* the diverted stuff. Furthermore, the diverted material is *not* rescanned for macros.

The built-in *divnum* returns the number of the currently active diversion. This is zero during normal processing.

## B.7 System Command

You can run any program in the local operating system with the *syscmd* built-in. For example,

```
syscmd(date)
```

runs the *date* command. Normally, *syscmd* would be used to create a file for a subsequent *include*.

To facilitate making unique file names, the built-in *maketemp* is provided, with specifications identical to the system function *mktemp*: a string of *XXXXX* in the argument is replaced by the process id of the current process.

## B.8 Conditionals

There is a built-in called *ifelse* which enables you to perform arbitrary conditional testing. In the simplest form,

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

compares the two strings *a* and *b*. If these are identical, *ifelse* returns the string *c*; otherwise it returns *d*. Thus, we might define a macro called *compare* which compares two strings and returns "yes" or "no" if they are the same or different.

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

Note the quotes, which prevent too-early evaluation of *ifelse*.

If the fourth argument is missing, it is treated as empty.

*ifelse* can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input

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

if the string *a* matches the string *b*, the result is *c*. Otherwise, if *d* is the same as *e*, the result is *f*. Otherwise the result is *g*. If the final argument is omitted, the result is null, so

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

is *c* if *a* matches *b*, and null otherwise.

## B.9 String Manipulation

The built-in *len* returns the length of the string that makes up its argument. Thus

```
len(abcdef)
```

is 6, and *len*((*a*,*b*)) is 5.

The built-in *substr* can be used to produce substrings of strings. *substr*(*s*, *i*, *n*) returns the substring of *s* that starts at the *i*th position (origin zero), and is *n* characters long. If *n* is omitted, the rest of the string is returned, so

```
substr(`now is the time', 1)
```

is

```
ow is the time
```

If *i* or *n* are out of range, various sensible things happen.

*index*(*s1*, *s2*) returns the index (position) in *s1* where the string *s2* occurs, or -1 if it doesn't occur. As with *substr*, the origin for strings is 0.

The built-in translit performs character transliteration.

```
translit(s, f, t)
```

modifies s by replacing any character found in f by the corresponding character of t. That is,

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If t is shorter than f, characters which don't have an entry in t are deleted; as a limiting case, if t is not present at all, characters from f are deleted from s. So

```
translit(s, aeiou)
```

deletes vowels from s.

There is also a built-in called dnl which deletes all characters that follow it up to and including the next newline. It is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. For example, if you say

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add dnl to each of these lines, the newlines will disappear.

Another way to achieve this, is

```
divert(-1)
        define(...)
        ...
divert
```

## B.10 Printing

The built-in errprint writes its arguments out on the standard error file. Thus, you can say

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

Dumpdef is a debugging aid which dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Don't forget the quotes.

## B.11 Summary of Built-ins

```
changequote(L, R)
define(name, replacement)
divert(number)
divnum
dnl
dumpdef(`name', `name', ...)
errprint(s, s, ...)
eval(numeric expression)
ifdef(`name', this if true, this if false)
ifelse(a, b, c, d)
include(file)
incr(number)
index(sl, s2)
len(string)
maketemp(...XXXXX...)
sinclude(file)
substr(string, position, number)
syscmd(s)
translit(str, from, to)
undefine(`name')
undivert(number,number,...)
```

# APPENDIX C: C Language Portability

The C language is defined in the appendix to "The C Programming Language", by Kernighan and Ritchie. This definition leaves many details to be decided by individual implementations of the language. It is those incompletely specified features of the language that detract from its portability and that should be studied when attempting to write portable C code.

Most of the issues affecting C portability arise from differences in either target machine hardware or compilers. C was designed to compile to efficient code for the target machine (initially a PDP-11) and so many of the language features not precisely defined are those that reflect a particular machine's hardware characteristics.

This document highlights the various aspects of C that may not be portable across different machines and compilers. It also briefly discusses the portability of a C program in terms of its environment, which is determined by the system calls and library routines it uses during execution, file pathnames it requires, and other items not guaranteed to be constant across different systems.

The C language has been implemented on many different computers with widely different hardware characteristics, varying from small 8-bit microprocessors to large mainframes. This document is largely concerned with the portability of C code in the XENIX programming environment. This is a more restricted problem to consider since all XENIX systems to date run on hardware with the following basic characteristics:

- Ascii character set.

- 8-bit bytes.

- 2 or 4 byte integers.

- Two's Complement Arithmetic.

None of these features is required by the formal definition of the language, nor is it true of all implementations of C. However, the remainder of this document is largely devoted to those systems where these basic assumptions hold.

The C language definition contains no specification of how input and output is performed. This is left to system calls and library routines on individual systems. Within XENIX systems there are a large number of system calls and library routines which can be considered portable. These are described briefly in a later section.

This document is not intended as a C language primer, for which should be used. It is assumed here that the reader is familiar with C, and with the basic architecture of common microprocessors.

## C.1 Source Code Portability

We are concerned here with source code portability, which means that programs can be compiled and run successfully on different machines without alteration.

Programs can be written to achieve this goal using several techniques. The first is to avoid using inherently non-portable language features. Secondly, any non-portable interactions with the environment, such as I/O to non-standard devices should be isolated, and possibly passed as an argument to the program at run time. For example programs should not, in general, contain hard-coded file pathnames except where these are commonly understood to be portable (an example might be /etc/passwd).

Files required at compile time (i.e. include files) may also introduce non-portability if the pathnames are not the same on all machines. However in some cases the use of include files to contain machine parameters can be used to make the source code itself portable.

## C.2 Machine Hardware

As mentioned earlier, most non-portable features of the C language are due either to hardware differences in the target machine or to compiler differences. This section lists the more common hardware differences encountered on XENIX systems and some language features to beware of.

### C.2.1 Byte Length

The length of the **char** data type is not defined in the language, other than that it must be sufficient to hold all members of the machine's character set as positive numbers. Within the scope of this document we will consider only 8-bit bytes, since this is the byte size on all XENIX systems.

### C.2.2 Word Length

The definition of C makes no mention of the size of the basic data types for a given implementation. These generally follow the most natural size for the underlying machine. It is safe to assume that **short** is no longer than **long.** Beyond that no assumptions are portable. For example on the PDP-11 **short** is the same length as **int,** whereas on the VAX **long** is the same length as **int.**

Programs that need to know the size of a particular data type should avoid hard-coded constants where possible. Such information can usually be written in a fairly portable way. For example the maximum positive integer (on a two's complement machine) can be obtained with:

```
#define MAXPOS  ((int)(((unsigned) 0) >> 1))
```

This is usually preferable to something like:

```
#ifdef PDP11
#define MAXPOS 32767
#else
        ...
#endif
```

Likewise to find the number of bytes in an **int** use **sizeof(int)** rather than 2, 4, or some other non-portable constant.

### C.2.3 **Storage Alignment**

The C language defines no particular layout for storage of data items relative to each other, or for storage of elements of structures or unions within the structure or union.

Some CPU's, such as the PDP-11 and M68000 require that data types longer than one byte be aligned on even byte address boundaries. Others, such as the 8086 and VAX-11 have no such hardware restriction. However, even with these machines, most compilers generate code that aligns words, structures, arrays and long words, on even addresses, or even long word addresses. Thus, on the VAX-11, the following code sequence gives '8', even though the VAX hardware can access an **int** (a 4 byte word) on any physical starting address:

```
struct s_tag {
        char c;
        int  i;
};
printf("%d\n",sizeof(struct s_tag));
```

The principal implications of this variation in data storage are twofold: 1) data accessed as non-primitive data types is not portable, and 2) neither is code that makes use of knowledge of the layout on a particular machine.

Thus unions containing structures are non-portable if the union is used to access the same data in different ways. Unions are only likely to be portable if they are used simply to have different data in the same space at different times. For example, if the following union were used to obtain four bytes from a long word, there's no chance of the code being portable:

```
union {
        char c[4];
        long lw;
} u;
```

The sizeof operator should always be used when reading and writing structures:

```
struct s_tag st;

    ...

write(fd, &st, sizeof(st));
```

This ensures portability of the source code. It does NOT produce a portable data file. Portability of data is discussed in a later section.

Note that the sizeof operator returns the number of bytes an object would occupy in an array. Thus on machines where structures are always aligned to begin on a word boundary in memory, the sizeof operator will include any necessary padding for this in the return value, even if the padding occurs after all useful data in the structure. This occurs whether or not the argument is actually an array element.

### C.2.4 Byte Order in a Word

The variation in byte order in a word between machines affects the portability of data between machines more than the portability of source code. However any program that makes use of knowledge of the internal byte order in a word is not portable. For example, on some PDP-11 systems there is an include file misc.h which contains the following structure declaration:

```
/*
* structure to access an
* integer in bytes
*/
struct {
        char    lobyte;
        char    hibyte;
};
```

With certain less restrictive compilers this could be used to access the high and low order bytes of an integer separately, and in a completely non-portable way. The correct way to do this is to use mask and shift operations to extract the required byte:

```
#define LOBYTE(i) (i & 0xff)
#define HIBYTE(i) ((i >> 8) & 0xff)
```

Note that even this is only applicable to machines with two bytes in an **int**.

One result of the byte ordering problem is that the following code sequence will not always perform as intended:

```
int c = 0;

read(fd, &c, 1);
```

On machines where the low order byte is stored first, the value of _c_ will be the byte value read. On other machines the byte is read into some byte other than the low order one, and the value of _c_ is different.

### C.2.5 Bitfields

Bitfields are not implemented in all C compilers. When they are, a number of restrictions apply:

- No field may be larger than an **int**.
- No field will overlap an **int** boundary. If necessary the compiler will leave gaps and move to the next int boundary.

The C language makes no guarantees about whether fields are assigned left to right, or right to left in an **int**. Thus while bitfields may be useful for storing flags, and other small data items, their use in unions to disect bits from other data is definitely non-portable.

To ensure portability no individual field should exceed 16 bits.

### C.2.6 Pointers

The C language is fairly generous in allowing manipulation of pointers, to the extent that most compilers will not object to non-portable pointer operations. The lint program is particularly useful for detecting questionable pointer assignments and comparisons.

The common non-portable use of pointers is where a pointer to one data type is cast to be a pointer to a different data type. This almost always makes some assumption about the internal byte ordering and layout of the data type, and is therefore non-portable. For example, in the following code, the ordering of the bytes from the **long** in the byte array is not portable:

```
char c[4];
long *lp;

lp = (long *)&c[0];
*lp = 0x12345678L;
```

The lint program will issue warning messages about such uses of pointers. Very occasionally it is necessary and valid to write code like this. An example is when the malloc() library routine is used to allocate memory for something other than type **char**. The routine is declared as type **char *** and so the return value has to be cast to the type to be stored in the allocated memory. If this type is not **char *** then lint will issue a warning concerning illegal type conversion. In addition, the malloc() routine is written to always return a starting address suitable for storing all types of data, but lint does not know this, so it gives a warning about possible data alignment problems too. In the following example, malloc() is used to obtain memory for an array of 50 integers. The code will attract a warning message from lint. There is nothing which can be done about this.

```
extern char *malloc();
int *ip;

ip = (int *)malloc(50);
```

### C.2.7 Address Space

The address space available to a program running under XENIX varies considerably from system to system. On a small PDP-11 there may be only 64k bytes available for program and data combined (although this can be increased - see 23fix(1)). Larger PDP-11's, and some 16 bit microprocessors allow 64k bytes of data, and 64k bytes of program text. Other machines may allow considerably more text, and possibly more data as well.

Large programs, or programs that require large data areas may have portability problems on small machines.

### C.2.8 Character Set

We have said that we are concerned here mainly with the ascii character set. The C language does not require this however. The only requirements are:

- All characters fit in the **char** data type.
- All characters have positive values.

In the ascii character set, all characters have values between zero and 127. Thus they can all be represented in 7 bits, and on an 8 bits per byte machine are all positive regardless of whether **char** is treated as signed or unsigned.

There is a set of macros defined under XENIX in the header file /usr/include/ctype.h which should be used for most tests on character quantities. Not only do they provide some insulation from the internal structure of the character set, their names are more meaningful than the equivalent line of code in most cases, Compare

```
if(isupper(c))
```

to

```
if((c >= 'A') && (c <= 'Z'))
```

With some of the other macros, such as isxdigit() to test for a hex digit, the advantage is even greater. Also, the internal implementation of the macros makes them more efficient than an explicit test with an 'if' statement.

## C.3 Compiler Differences

There are a number of C compilers running under XENIX. On PDP-11 systems there is the so called "Ritchie" compiler. Also on the 11, and on most other systems, there is the Portable C Compiler.

### C.3.1 Signed/Unsigned char, Sign Extension

The current state of the signed versus unsigned char problem is best described as unsatisfactory. The problem is completely explained and discussed in Sign Extension and Portability in C, Hans Spiller, Microsoft 1982, so that material is not repeated here.

The sign extension problem is one of the more serious barriers to writing portable C, and the best solution at present is to write defensive code which does not rely on particular implementation features. The above paper suggests some ways.

### C.3.2 Shift Operations

The left shift operator, << shifts its operand a number of bits left, filling vacated bits with zero. This is a so-called logical shift.

The right shift operator, >> when applied to an unsigned quantity, performs a logical shift operation. When applied to a signed quantity, the vacated bits may be filled with zero (logical shift) or with sign bits (arithmetic shift). The decision is implementation dependent, and code which uses knowledge of a particular implementation is non-portable.

The PDP-11 compilers use arithmetic right shift. Thus to avoid sign extension it is necessary to either shift and mask out the appropriate number of high order bits, or to use a divide operator which will avoid the problem completely:

```
     char c;

     For  c >> 3;    use:    (c >> 3) & 0x1f;
                     or:     c / 8;
```

### C.3.3 Identifier Length

The use of long identifier names will cause portability problems with some compilers. There are three different cases to be aware of:

- C Preprocessor Symbols.
- C Local Symbols.
- C External Symbols.

The loader used may also place a restriction on the number of unique characters in C external symbols.

Symbols unique in the first six characters are unique to most C language processors.

On some non-XENIX C implementations, upper and lower case letters are not distinct in identifiers.

### C.3.4 Register Variables

The number and type of register variables in a function depends on the machine hardware and the compiler. Excess and invalid register declarations are treated as non-register declarations, which should not cause a portability problem. On a PDP-11, up to three register declarations are significant, and they must be of type **int**, **char**, or pointer. ( Page 81). Whilst other machines/compilers may support declarations such as "register unsigned short" this should not be relied upon.

Since the compiler ignores excess register keywords, register type variables should always be declared in their importance of being register type. Then the ones the compiler ignores will be the least important.

### C.3.5 Type Conversion

The C language has some rules for implicit type conversion; tt also allows explicit type conversions by type casting. The most common portability problem arising from implicit type conversion is unexpected sign extension. This is a potential problem whenever something of type **char** is compared with an **int**.

For example

```
char c;

if(c == 0x80)
        ...
```

will never evaluate true on a machine which sign extends since <u>c</u> is sign extended before the comparison with 0x80, an int.

The only safe comparison between **char** type and an **int** is the following:

```
     char c;

     if(c == 'x')
             ...
```

This is reliable since C guarantees all characters to be positive. The use of hard-coded octal constants is subject to sign extension. For example the following program prints ff80 on a PDP-11:

```
     main()
     {
             printf("%x0,'\200');
     }
```

Type conversion also takes place when arguments are passed to functions. Types **char** and **short** become **int.** Once again machines that sign extend **char** can give surprises. For example the following program gives -128 on the PDP-11:

```
     char c = 128;
     printf("%d\n",c);
```

This is because c is converted to **int** before passing on the stack to the function. The function itself has no knowledge of the original type of the argument, and is expecting an int. The correct way to handle this is to code defensively and allow for the possibility of sign extension:

```
char c = 128;
printf("%d\n", c & 0xff);
```

### C.3.6 **Functions With Variable Number of Arguments**

Functions with a variable number of arguments present a particular portability problem if the type of the arguments is variable too. In such cases the code is dependent upon the size of various data types.

In XENIX there is an include file, /usr/include/varargs.h, that contains macros for use in variable argument functions to access the arguments in a portable way:

```
typedef char *va_list;
#define va_dcl int va_alist;
#define va_start(list) list = (char *) &va_alist
#define va_end(list)
#define va_arg(list,mode) ((mode *)(list += sizeof(mode)))[-1]
```

Figure 1. File: /usr/include/varargs.h

The va_end() macro is not currently required. The use of the other macros will be demonstrated by an example of the fprintf() library routine. This has a first argument of type **FILE ***, and a second argument of type **char ***. Subsequent arguments are of unknown type and number at compilation time. They are determined at run time by the contents of the control string, argument 2.

The first few lines of fprintf() to declare the arguments and find the output file and control string address could be:

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

int
fprintf(va_alist)
va_dcl;
{
      va_list ap;      /* pointer to arg list   */
      char *format;
      FILE *fp;

      va_start(ap);    /* initialize arg pointer */
      fp = va_arg(ap, (FILE *));
      format = va_arg(ap, (char *));

                ...
}
```

Note that there is just one argument declared to fprintf(). This argument is declared by the va_dcl macro to be type int, although its actual type is unknown at compile time. The argument pointer, ap, is initialized by va_start() to the address of the first argument. Successive arguments can be picked from the stack so long as their type is known using the va_arg() macro. This has a **type** as its second argument, and this controls what data is removed from the stack, and how far the argument pointer, ap, is incremented. In fprintf(), once the control string is found, the type of subsequent arguments is known and they can be accessed sequentially by repeated calls to va_arg(). For example, arguments of type **double**, **int ***, and short, could be retrieved as follows:

```
        double dint;
        int *ip;
        short s;

        dint = va_arg(ap, double);
        ip = va_arg(ap, (int *));
        s = va_arg(ap, short);
```

The use of these macros makes the code more portable, although it does assume a certain standard method of passing arguments on the stack. In particular no holes must be left by the compiler, and types smaller than **int** (e.g. **char**, and short on long word machines) must be declared as **int.**

### C.3.7 Side Effects, Evaluation Order

The C language makes few guarantees about the order of evaluation of operands in an expression, or arguments to a function call. Thus

```
        func(i++, i++);
```

is extremely non-portable, and even

```
        func(i++);
```

is unwise if _func_() is ever likely to be replaced by a macro, since the macro may use _i_ more than once. There are certain XENIX macros commonly used in user programs; these are all guaranteed to only use their argument once, and so can safely be called with a side-effect argument. The commonest examples are _getc_(), _putc_(), _getchar_(), and _putchar_().

Operands to the following operators are guaranteed to be evaluated left to right:

```
        ,       &&      ||      ?       :
```

Note that the comma operator here is a separator for two C statements. A list of items separated by commas in a declaration list are not guaranteed to be processed left to right. Thus the declaration

```
        register int a, b, c, d;
```

on a PDP-11 where only three register variables may be declared could make any three of the four variables register type, depending on the compiler. The correct declaration is to decide the order of importance of the variables being register type, and then use separate declaration statements, since the order of processing of individual declaration statements is guaranteed to be sequential:

```
        register int a;
        register int b;
        register int c;
        register int d;
```

For the same reason declaration initializations of the following type are unwise:

```
        int  a = 0, b = a;
```

## C.4 Program Environment Differences

Most non-trivial programs make system calls and use library routines for various services. The sections below indicate some of those routines that are not always portable, and those that particularly aid portability.

We are concerned here primarily with portability under the XENIX operating system. Many of the XENIX system calls are specific to that particular operating system environment and are not present on all other operating system implementations of C. Examples of this are getpwent() for accessing entries in the XENIX password file, and getenv() which is specific to the XENIX concept of a process's environment.

Any program containing hard-coded pathnames to files or directories, or user id's, login names, terminal lines or other system dependent parameters is non-portable. These types of constant should be in header files, passed as command line arguments, obtained from the environment, or by using the XENIX default parameter library routines dfopen(), and dfread().

Within XENIX, most system calls and library routines are portable across different implementations and XENIX releases. However, a few routines have changed in their user interface.

### C.4.1 Libraries

The various XENIX library routines are generally portable among XENIX systems; however, note the following:

printf The members of the printf family, printf, fprintf, sprintf, sscanf, and scanf have changed in several small ways during the evolution of XENIX, and some features are not completely portable. The return values from these routines cannot be relied upon to have the same meaning on all systems. Certain of the format conversion characters have changed their meanings, in particular relating to upper/lower case in the output of hexadecimal numbers, and the specification of **long** integers on 16-bit word machines. The reference manual page for printf(3S) contains the correct specification for the routines.

## C.5 Portability of Data

Data files are almost always non-portable across different machine CPU architectures. As mentioned above, structures, unions, and arrays have varying internal layout and padding requirements on different machines. In addition, byte ordering within words and actual word length may differ.

The only way to get close to data file portability is to write and read data files as one dimensional character arrays. This avoids alignment and padding problems if the data is written and read as characters, and interpreted that way. Thus ascii text files can usually be moved between different machine types without too much problem.

## C.6 Lint

For a complete description of lint(1) see the discussion in a following chapter.

Lint is a C program checker which attempts to detect features of a collection of C source files which are non-portable or even incorrect C. One particular advantage over any compiler checking is that lint checks function declaration and usage across source files. Neither compiler nor loader do this.

Lint will generate warning messages about non-portable pointer arithmetic and dubious assignments and type conversions. Passage unscathed through lint is not a guarantee that a program is completely portable.

## C.7 Byte Ordering Summary

The following conventions are used below. 'a0' is the lowest physical addressed byte of the data item. 'a1' has a byte address a0 + 1, etc. 'b0' is the least significant byte of the data item, 'b1' being the next least significant, etc.

Note that any program which actually makes use of the following information is guaranteed to be non-portable!

```
 +---------+           +-------------------+
 | b0 | b1 |           | b2 | b3 | b0 | b1 |
 +---------+           +-------------------+
   a0   a1               a0   a1   a2   a3

 type short                  type long
 ----------                  ---------
```

**Figure 2.** PDP-11 Byte Ordering

```
 +---------+           +-------------------+
 | b0 | b1 |           | b0 | b1 | b2 | b3 |
 +---------+           +-------------------+
   a0   a1               a0   a1   a2   a3

 type short                  type long
 ----------                  ---------
```

**Figure 3.** VAX-11 Byte Ordering

```
 +---------+           +-------------------+
 | b0 | b1 |           | b2 | b3 | b0 | b1 |
 +---------+           +-------------------+
   a0   a1               a0   a1   a2   a3

 type short                  type long
 ----------                  ---------
```

**Figure 4.** 8086 Byte Ordering

```
 +---------+           +-------------------+
 | b1 | b0 |           | b3 | b2 | b1 | b0 |
 +---------+           +-------------------+
   a0   a1               a0   a1   a2   a3

 type short                  type long
 ----------                  ---------
```

**Figure 5.** M68000 Byte Ordering

```
+----------+          +-------------------+
| b1 | b0 |          | b3 | b2 | b1 | b0 |
+----------+          +-------------------+
  a0   a1               a0   a1   a2   a3

type short               type long
----------               ---------
```

**Figure 6.** Z8000 Byte Ordering