UNOS 1429

Bell Laboratories

subject: A UNIX™ Tutorial - Version 4.0
Update - Case 40288-100

date: February 5, 1982

from: K. E. Wendland
IH 55625
6D-507 x2068
55625-820205.01EN

ABSTRACT

This memorandum is a revision and considerable expansion of the document "A UNIX™ Tutorial" (5613-800715.01EN).

Covered are topics from "How to obtain a UNIX user ID" and login/logoff procedures, to a thorough discussion of the text editor, the file and directory structure of UNIX, online/offline printing, "shell" properties and simple "shell" scripts, the user's environment, and most commands (greatly expanded from previous issues) useful to the average user. All information is based on the latest Version 4.0 of UNIX available at Indian Hill.

The notes are meant to help the new user make effective use of the available UNIX facilities; they are written in a tutorial format (introducing simple concepts and expanding upon them, with many examples and homework). The "table of contents" is quite detailed, which also makes this document a good reference for the more experienced user.

Copy (abstract only) to
All Supervision Division 556

Copy to
T. Giammarresi
G. Hamstra
L. A. Nelsen
C. L. Scheiderman

CONTENTS

Bell Laboratories

subject: A UNIX™ Tutorial - Version 4.0
Update - Case 40288-100

date: February 5, 1982

from: K. E. Wendland
IH 55625
6D-507 x2068
55625-820205.01EN

ENGINEER'S NOTES

1. INTRODUCTORY COMMENTS ABOUT UNIX™

1.1 What is UNIX ??

UNIX is an operating system, with a hierarchical file structure, a powerful context editor, high level programming languages, and other useful features. To the user, it is a general-purpose, multi-user, time-sharing, interactive computer system.

1.2 Obtaining a UNIX User ID

If you do not have a valid UNIX user ID, go to room 6M-402 (the *current* location of the UNIX administrator at Indian Hill) and obtain a "COMPUTER SERVICES REQUEST" form. Complete the "User Data," "Charging Information" and "IH" sections, have your supervisor complete the "Authorizer Data" section, and return to room 6M-402. Do *not* fill in the password section, as UNIX initially assumes your payroll account number is your password; the first time you enter the system, you will be prompted for a new personalized password. Also, go to room 6F-136 and obtain all available documentation on the PWB/UNIX time-sharing system.

It should be noted that Indian Hill has been mentioned, which is logical, as this memo was written here; for other locations, administrative procedures may vary, but the bulk of this document is still valid.

1.3 Login Procedures Within BTL

The following general procedures will get you logged on "dumb" terminals, such as a Texas Instruments Silent 700 or Lear Siegler ADM-3:

- Connect terminal modem to a green Dimension data phone and dial the *local* number for *your* UNIX system. There are many general purpose and project UNIX machines at Indian Hill --- check with your supervisor or mentor to determine the correct number to use. The terminal should be powered on, set to lower case (if possible), set to full duplex, and set to 30 characters per second (or higher speed, if your terminal and UNIX system is set up for such operation).

- Once you are logically connected to UNIX, an "on line" or "carrier" indicator will light, and a printout will appear as:

  ```
  login:
  ```

  Type in your "user ID" (or "login name") followed by a carriage return (CR).

- The next request printed out is:

  ```
  Password:
  ```

  and UNIX expects your "personal password" followed by a CR. to be entered; printing of this password will be inhibited.

- After printing one or more lines of informative comments, a prompt

  ```
  $
  ```

  will be printed. UNIX is now ready to accept your commands.

1.4 Login Procedures Outside BTL

You may gain access to UNIX from outside BTL using any telephone that will fit your modem, and by dialing a special number, obtainable from people in your group or your local UNIX counselor (at Indian Hill, UNIX is conected through a Wheaton exchange). If this mode of entry is used, the login dialogue will be identical to the scenario described above except the computer will additionally request:

```
External Security:
```

to which you must respond with the current "external security code word" (printing inhibited) followed by a CR; this code word changes monthly and is available from your supervisor.

1.5 Logoff Procedures

Once you are through with your UNIX session, you can simply break the phone connection to disengage the computer. Alternately, you may type "CONTROL-D" (typing D while depressing the CONTROL Key), which will result in a new LOGIN request; at this point you may login again or simply hang-up.

1.6 Typing Errors

Erase Character (#) To correct typing errors caught *before* a "carriage return," you can use

```
#
```

to erase the last character typed; repetitive use of "#" can erase any number of characters back to the beginning of the line, but not beyond. An example is:

```
my#anxx##
```

which will be interpreted by UNIX as "man" typed correctly. A common error is to "backspace" (simultaneously depressing the "CONTROL" and "H" keys) over an unwanted "space"; use the "#" erase character instead, as both "space" and "backspace" are legal characters in UNIX. Also, it should be noted that "#" will work during the login interval.

Kill Line Character (@) The "at-sign"

```
@
```

will erase the entire line printed thus far, return you to the beginning of a new line, and allow you to retype that line from "scratch". Again, the "@" will work during the login interval.

1.7 Readahead Capabilities

Once the initial command prompt is received, UNIX allows full readahead. i.e., you may type at any speed regardless of what is printing on the terminal. It should be noted this may cause a strange intermixing of input/output characters to be printed, but UNIX will eventually interpret your typed commands correctly.

1.8 Stopping Terminal Output

Temporarily Stopping Output At times, it may be convenient to stop the "printing" of terminal output *temporarily* (especially when you are trying to keep information from rolling off a CRT type terminal screen, before it can be read). This may be accomplished by typing "*CONTROL-S*" (simultaneously depressing the "CTRL" and "S" keys). To resume printing, simply type another "CONTROL-S"; typing any *other* character will also cause printing to continue, but UNIX will save these characters for the beginning of the next command. Also, a common error is to hit "CONTROL-D" rather than "CONTROL-S." which will promptly log you off the system after the printing of output is completed.

Terminating a Command To *terminate* (permanently) a command in progress and any output it is producing, simply hit the "BREAK" or "DEL(ETE)" key.

1.9 Issuing UNIX Commands

Your response to a "$" command prompt is to issue UNIX commands. These commands are numerous, and many will be discussed in later sections. For now, respond to your first "$" with

```
date
```

followed by a CR. The result of the "date" command is a print-out of the current date and time.

By this time, it should be obvious that all commands and responses, when complete, should be followed by a "carriage return" --- the CR will be implicit in the remainder of these notes.

1.10 File Description

A file is simply a collection of characters (data, text, programs, executable code, etc. ---- a maximum of several million characters) stored in UNIX. The simplest way to form a file is via the TEXT EDITOR, which will be considered in Section 2.

1.11 References

The following references (available from the UNIX library) might be of use:

"A Tutorial Introduction to the UNIX Text Editor" by Kernighan

"Advanced Editing in UNIX" by Kernighan

"UNIX for Beginners" by Kernighan

"UNIX User's Manual" by Dolotta, et.al.

Note the User's Manual is an extremely useful reference source --- use it *only* as a reference, not a tutorial source for learning UNIX.

Also of interest is a program called "learn," which can be used as a source of supplemental "homework" problems. See Section 4.1 for more details.

1.12 Using This Document as a Tutorial

This document is written in "tutorial" format, starting with and thoroughly explaining simple concepts, and expanding upon them. Try to understand these concepts in the order presented, read the associated examples, and then get your "hands on" a terminal to actually test your understanding. Try *not* to "skim" through this memo, as there are "pearls of wisdom" embedded in every section (these are problems that I ran into when I was first exposed to UNIX and had only poor documentation available --- you can benefit from my experience and frustration, by being aware of such problems in advance).

As a final note here, the "Table of Contents" is detailed enough to use this memo as a reference later; command abbreviations appear in *parentheses* following the section descriptions.

2. THE UNIX TEXT EDITOR

2.1 Creating a New File

Entering the Editor (ed) The UNIX TEXT EDITOR allows you to get information stored into a file. Upon first entry type:

```
  ed                [don't forget the CARRIAGE RETURN]
```

in response to a "S" prompt, and you are now in the editor.

Entering Your First Text Lines (a) To enter text, the APPEND command must be given, i.e., type:

```
  a                 [again, don't forget CR]
```

followed by as many lines of text as you desire. Remember "#" and "@" can be used to correct typographical errors. When you have typed in the desired text, type a line with *only* a single "period" on it, i.e.,

```
  .
```

The "period" must be in the *first space* followed *immediately* by a CR --- any variation will *not* allow you to leave the "append" mode and all additional lines typed will be added as garbage to your text, until you properly terminate the appending session as described.

The sequence of commands:

```
  a
  I love telephones
  .
```

will result in the line "I love telephones" being stored in the "editor buffer" (a temporary storage area). The initial "a" and final "." will *not* appear in the buffer, as they are editor commands, not text.

Writing Text onto a New File (w) Once text is accumulated in the editor buffer, it is likely we would like to store it in an external (disk) file for later use. We can use the editor WRITE command as follows:

```
  w file_name    [a "blank" after "w" is necessary]
```

A "file_name" is composed of up to 14 alphanumeric and/or special characters. A "blank", "?", "\", "/", "*", "[", "|", "^", or non-printable characters may/should *not* be used; in fact, to play it safe, use only alphanumeric characters, the "underscore" (_) and "period" (.). The editor responds to a "w" command with a count of characters written onto the file, *if* all goes well; the count includes "blank" and "carriage return" (or "new line") characters.

It should be noted that writing onto a file simply *copies* the contents of the editor buffer to the named file, leaving the contents of the buffer undisturbed. If writing onto an existing file, the original contents are destroyed by overwriting.

Leaving the Editor (q) To leave the editor, simply type the QUIT command

```
  q
```

which will return you to the UNIX command level, indicated by a "S" prompt.

At this point the contents of the editor buffer vanishes, which is why you should write it out onto a file before quitting. If you have modified the contents of the editor buffer in any way, and did *not* use the write command before quitting, the editor will respond with a "?" and give you another chance; another "q" command will allow you to successfully "quit" the editor. In fact, any unrecognizable or illogical editor command will result in a "?" response and an invalidation of the typed line.

2.2 Editing Existing Files

Bringing Files into the Editor Buffer (ed, e) When editing existing files, rules for the "ed," "a" and "w" commands can be enhanced. For the purpose of exposition, assume a file "junk" contains a single line: "I love telephones". To transfer the contents of "junk" into the editor buffer, any of the following commands may be used:

```
  ed junk
```

or

```
  ed
  e junk
```

or

```
  e junk
```

The editor will respond with a character count of the text entered into its buffer (in this case, 18). Note that "ed" is referred to as the EDITOR command, and "e" is called simply EDIT.

If the *file* you've requested does *not* exist, UNIX will notify you with a message like "?junk". You have still successfully entered the editor, just as if "junk" did exist, but the *buffer* is *empty*.

Overwriting an Existing File (w) The editor remembers the name of the file it is working on (see the FILENAME command below). Therefore, the command:

```
  w                [followed directly by a CR]
```

in this case, is equivalent to:

```
w junk
```

Filename Associated with an Editor Buffer (f) The editor remembers the filename associated with the *last* "e" or "ed" command -- if you forget, use the FILENAME command:

```
f
```

which will yield a response "junk" (in this case).

If you want to change the name associated with the *editor buffer*, say to "garbage," you may type:

```
f garbage
```

to which UNIX will "echo" the new filename. The contents of the "editor buffer" remain unchanged, but the "filename" the editor remembers is now altered. This command can also be used to associate a filename with the buffer, if the "ed" command (without a filename argument) was used to invoke the editor.

Adding Text to the End of the Buffer (a) The APPEND command, used *immediately* after an existing file is brought into the editor, will add text to the *end* of the current buffer. For example:

```
ed junk
18                  [character count of file read into buffer]
a
--- especially Bell telephones
.
w                   [implicit overwrite of file "junk"]
49                  [character count of buffer written]
q
```

will result in a two line file "junk" now containing:

```
I love telephones
--- especially Bell telephones
```

Repeated Edits (e) If at any time you have finished working with an editor file, you may issue a new "e" command without "quitting" the editor. Consider:

```
e junk    [transfers 18 character file "junk" into
18                                         editor buffer]
{ editing session }
w         [writes 40 character editor buffer contents
40                                       into file "junk"]
e nuts    [transfers 23 character file "nuts" into
23                                         editor buffer]
{ editing session }
w         [writes 37 character editor buffer contents
37                                       into file "nuts"]
q
```

Note, when the command "e nuts" is typed, the old contents of the editor buffer (from file "junk") is destroyed and replaced by the contents of file "nuts"; if the contents of an existing editor buffer has been modified in any way, and you do *not* write it onto a file before invoking a new "e" command, UNIX will respond with "?," giving you another chance.

Suppressing Diagnostics and Character Counts As a final note, if you enter the editor via a command similar to:

```
ed - junk
```

the "-" option indicates that *character counts* due to the "ed," "e," "w" or "r" commands, as well as "?" diagnostics from the "e" or "q" commands, will be *suppressed*.

Unexpected Exit from the Editor For some reason, transmission between your terminal and UNIX may be severed, while you are in an editing session. Therefore, it is a good idea to use the "write" command *liberally* while modifying text — if you didn't and the system goes down, don't panic yet, as UNIX *may* have saved you. *Many* times (not always), the editor buffer will be saved *automatically* in a file "ed.hup," created in your working directory. So when you log in again, check "ed.hup" first, and you may be surprised that your old editor buffer has survived intact.

Appending a File's Contents to the Buffer (r) The final command of interest here is the READ command. This command reads the contents of a specified file, and *adds* it to the *end* of the existing contents of the editor buffer. Consider:

```
e junk    [transfers 18 character file "junk" into editor
18                                                buffer]
r nuts    [adds 23 character file "nuts" to the end of the
23                      existing contents of editor buffer]
w         [writes combined 41 character editor buffer onto
41                                           file "junk"]
q
```

Note that an "r" command does *not* change the file name currently associated with the editor; an "f" command, issued anywhere in the above program, would yield a response "junk," the name of the *last* file entered via an "e" or "ed" command. Now that you are a little more sophisticated, I can tell you that the above statement is not *always* true. If you enter the editor via an "ed" command, with *no* filename argument, the editor will remember the first filename it sees thereafter, which may be associated with a READ or explicit WRITE command. Once a name is associated with the editor buffer, however, it can *only* be changed by an explicit "f" or "e" command.

2.3 Printing Text

Basic Printing (p) The PRINT command "p" will print the contents of the editor buffer (or specific parts of it) on the terminal. We specify the line numbers where printing is to *begin* and to *end*, separated by a "comma," immediately followed by "p". Assume we have a file "stocks" containing the following lines

```
AT & T              [line #1]
General Motors      [line #2]
Georgia Pacific     [line #3]
Ford                [line #4]
```

Consider the following commands:

```
e stocks
41
2,4p                       [print lines 2 to 4]
General Motors             [line #2 -- starting line]
Georgia Pacific            [line #3]
Ford                       [line #4 -- ending line]
q
```

Note that 0 (zero) cannot be used as a starting line number, that the ending line number must be greater than or equal to the starting line number, and that the ending line number must be less than or equal to the last line in the buffer; a breach of these rules will cause a "?" to be printed on the terminal, indicating an irrational command.

The editor command

```
4,4p
```

will print only line #4 of the text and has an equivalent abbreviated form:

```
4p
```

End of Buffer Symbol ($) It should also be noted that in many typing sessions, you will lose track of the size of your buffer, i.e., the number of the last line. To alleviate this problem, a special symbol "$" can be used to designate the last line in the buffer. For our file "stocks"

```
$p
```

will print line #4, the *last line* in the *buffer*.

```
  1,$p
```

will print the entire buffer contents.

As a further form of abbreviation, when printing a single line, the terminating "p" may also be deleted, i.e., the following editor commands are equivalent for the file "stocks": "4,4p", "4p", "4", "$,$p", "$p", "$".

Terminating a Printout It should be noted that many times a long print sequence is started, and for some reason you wish to stop it; depressing the "DEL(ETE)" or "BREAK" key will cause printing to cease and return control to editor. This action will *not* cause you to leave the editor, but only the printing sequence will be stopped and UNIX will reply with a "?".

Similarly, you may stop output *temporarily* using the "CONTROL-S" key, as described previously in Section 1.8.

Current Line Symbol (.) The editor also provides the concept of "current line"; the "most recent" line that we have done *anything* with is symbolically denoted by "." (referred to as "dot"). If we issue a print command

```
  .p
```

or, in an abbreviated form,

```
  p
```

the current line will be printed. Consider a more complicated sequence of print commands, based on our file "stocks":

```
  e stocks
  41
  1,2p               [causes line #1 and #2 to be printed]
  AT & T
  General Motors     [sets "." to line #2 at this point]
  p                  [prints current line, i.e., line #2]
  General Motors
  .,$p               [prints current line #2 to end of buffer]
  General Motors
  Georgia Pacific
  Ford
  q                  [upon quitting the editor, "." = "$" = "4"]
```

As shown above, the "." can be used as a line number in any print command.

It should be pointed out that when you first try to bring a file into the editor, "dot" will be set to the last line read into the buffer (the last line UNIX has done anything with).

Printing Line Numbers (=) In complex editing situations, you may lose track of the current line number. The command

```
.=
```

will result in a printout of the value of "dot". Similarly, "$=" will yield the number of the last line in the buffer, but this command will *not* change the value of "." to "$" (note that the command "=" *alone* defaults to "$=").

Relative Line Number Addressing The editor also allows printing of lines relative to the current ("dot") line. Consider the following commands:

```
.+1p           [prints "next line"]
.-1p           [prints "previous line"]
.+3p           [prints "third line after current line"]
```

In all of the above cases of *single* line printing, the "." and "p" may be optionally deleted without effect.

Multiple line printing may also incorporate this relative format. such as:

```
.-1,.+1p
```

will print the "previous, current and next lines". For multiline printing, the "." again is unnecessary in either argument, but the suffix "p" must be present. Also, if the "." is retained, the "+" may be deleted in forward relative addresses, i.e. ".+3p", "+3p", ".3p", ".+3", "+3" and ".3" are all equivalent.

The "$" (end of buffer) may be used instead of the "." in similar expressions, such as:

```
$-1p           [prints the "next to last" line in buffer]
$-3,$p         [prints the "last four" lines in buffer]
```

Also, there are abbreviations of relative printing commands, such as:

```
-
```

which is equivalent to ".-1p", "--" which is equivalent to ".-2p", "---" which is equivalent to ".-3p", and so forth. Similarly, the command:

```
+
```

is equivalent to ".+1p", "++" is equivalent to ".+2p", "+++" is equivalent to ".+3p", etc. The "+" or "-" (single or multiple occurrences) may be appended to any line number, to indicate an *increment* or *decrement* to that address, such as:

```
$--
```

will print "two lines from the end of the buffer".

Finally, a simple *carriage return* will print the next line (it is equivalent to ".+1p").

A warning in all relative printing statements is that: (1) the value of "." can change only *after* a command has been executed, (2) lines referred to must be valid lines in the buffer, i.e., in the range "1" to "S," inclusive, and (3) in multiple line printing statements, the beginning line number must always be less than or equal to the ending line number. To ensure your understanding, assume a file "number" exists, with contents:

```
one
two
three
four
five
six
```

and consider the following editor dialogue:

```
e number
28
6p                 [prints line #6, "dot" set to "6"]
six
-3                 [prints line ".-3" = line #3, "dot" set to "3"]
three
++                 [prints line ".+2" = line #5, "dot" set to "5"]
five
p                  [prints current line, "dot" unchanged]
five
(carriage return)  [prints next line, "dot" set to "6"]
six
+                  [attempts to print line #7 (non-existent)]
?                  [invalid command, "dot" unchanged at "6"]
.-3,-p             [prints lines ".-3" = #3 to ".-1" = #5]
three
four
five
q                  ["dot" set to "5" when quitting editor]
```

Special Meaning of , and ; as Line Ranges As a final note, the line range "1,S" may be abbreviated by the single character "comma" (,). An example is

```
,p
```

which will print the entire editor buffer for you. The range ".,S" can be replaced by the single "semicolon" (;) character.

These range abbreviations for line numbers, as well as all relative addressing seen thus far, may be used wherever applicable, *not* only in PRINT commands.

Listing Lines (l) There is another printing command called LIST ("l" ---- that's lower case "L," not the number "1") which follows all the rules of the "p" command. For normal text, the "l" and "p" commands will yield identical results, but the "list" command has the capability of indicating unprintable characters. Examples are: "tab" lists as ">" (entered into text via "CONTROL-I"), "backspace" lists as "<" (entered into text via "CONTROL-H"), and other control characters will print as a "backslash" followed by a string of digits, such as "\07"; note that the "tab" will appear as ">" and the "backspace" as "<" on *CRT* type terminals. If a line is over 72 characters in length, the "list" command will print it on multiple lines with each line, except the last, terminated with a "backslash" (\) to indicate continuation.

If any invisible characters (other than "backslash" at the end of a line, "tab" or "backspace") appear in a listing, the odds are that your finger slipped in typing, as you almost never want them. Some of these special characters can cause havoc in a large program, and the "p" command will be of *no* use in trying to locate them.

2.4 Deleting, Moving and Adding Lines

Writing Selected Lines Only (w) With the introduction of various techniques of line numbering, it may be wise to return to the WRITE command momentarily. When using the "w" command, the *entire* editor buffer contents need *not* be written onto a file; you may choose specific lines, if you wish, such as:

```
  S-9,Sw
```

will write the last 10 lines of the buffer onto the last file indicated by the last "ed," "e" or "f" command, and

```
  -1w good
```

will write the line previous to the current "dot" line into file "good".

As a final note, the value of "dot" will *not* be affected by any WRITE command. Also, should you attempt to "quit" the editor after a "selective write," UNIX may respond with a "?," if you've modified the buffer contents and have failed to "write" the *entire* buffer oto a file. As mentioned previously, another "q" will allow you to exit the editor, if you really want to.

Read Command Flexibility (r) The READ command:

```
  4r junk
```

will read the contents of the specified file (in this case, "junk") and add it to the editor buffer *after* the indicated line number (in this case, line #4). If the line number is omitted, "S" (or the last line in the editor buffer) is the default (as was illustrated in Section 2.2). Also, "0" is a valid line number, if you wish to insert the file text at the beginning of the editor buffer. After the READ command has completed execution, "dot" is set to the last line read into the editor buffer.

Append Command Flexibility (a) Now let's revisit the APPEND command. Consider:

```
4a                          [adds text after line #4]
{ text to be added }
.                           [quits "append" mode]
```

which adds text *after* line #4. The integer "4" was illustrative: actually, any line number in the range "1" to "$," "." or valid relative addressing (as described for the PRINT command) may be used. In addition "0a" is allowed to add text to the beginning of the editor buffer. It should be noted that at the end of the appending session, "dot" will be set to the last line of text added, and all line numbers of original text *following* the appended lines will be changed.

A final comment is that "a" alone defaults to ".a". This is useful when first entering the editor via "e file_name", which sets the "dot" to the last line read from the file; an "a" (without line number) will now append text to the *end* of the editor buffer (as was shown in Section 2.2).

Inserting Lines of Text (i) A similar command is INSERT ("i"), which adds text *before* the indicated line number, such as:

```
1i                          [inserts text before line #1]
{ text to be inserted }
.                           [quits insert mode]
```

which will add lines before line #1 or at the beginning of the editor buffer. Valid line numbers are "1" through "$," "." or valid relative addresses. As a default, "i" is equivalent to ".i". Again, at the end of a session, "dot" is set to the last line inserted.

Deleting Lines of Text (d) Another useful command is DELETE ("d"), which may be used for *single* or *multiple* line deletions, such as:

```
4,$d        [deletes line #4 through the end of the buffer]
-1d         [deletes "next line" (after "dot" line)]
d           [deletes "current line" -- equivalent to ".d"]
;d          [deletes from "current line" to "end of buffer"]
,d          [deletes entire buffer]
```

In all cases of deletions, "dot" is set to the line *after* the last line deleted, unless you deleted lines from the end of the buffer, in which case "." is set to "$". The range of allowed line numbers is again "1" to "$," "." or valid relative addresses (including the special "," and ";" line range abbreviations, as illustrated).

Changing Lines of Text (c) A related command is CHANGE ("c"), which can be used in *single* or *multiple* line format; this command deletes designated line numbers and replaces them with desired text, such as:

```
4,9c                       [deletes lines #4 through #9]
{ replacement test }
.                          [quits change mode]
```

It should be noted that the number of replacement lines need *not* equal the number of lines deleted, and you may issue the CHANGE command for a *single* line, such as the command "7c", which will delete line #7 and replace them with text which follows. After completion of a CHANGE session, "dot" is set to the last replacement line. The valid range of line numbers is again "1" to "$," "." and relative addresses. As a default, "c" is equivalent to ".c".

Moving Lines of Text (m) You can move lines within the buffer via the MOVE ("m") command, of general form:

```
A,BmC
```

where "A" and "B" represent the beginning and ending line numbers, respectively, of lines to be moved ("B" must be greater or equal to "A," and, if only one line is to be moved, a single line number replaces the couplet "A,B"), and these lines will be placed *after* line "C". The line numbers "A," "B" and "C" may fall in the usual range "1" to "$," "." or relative addresses. In addition, "C" may have value "0" to move text to the beginning of the editor buffer, but "C" may *not* fall within the inclusive range dictated by "A" and "B" (as it makes no sense to attempt this type of move). The "dot" is set to the last moved line, and the original lines of text *are* moved, i.e., they no longer exist in their original location.

Copying Lines of Text (t) A related command COPY ("t") provides the same effects as the MOVE command, *except* the original lines remain intact. Also, if we view the general form "A,BtC," "A," "B" and "C" have ranges of validity as described for the MOVE command, but "C" *may* fall within the inclusive range dictated by "A" and "B" (as it now makes sense to do so).

To check your understanding of commands described in this section, consider manipulating the previous "number" file containing:

```
one
two
three
four
five
six
```

with the following editor session:

```
e number
28
2a          [append after line #2]
help
.           [quits append mode - "dot" set to "3"]
3i          [insert before line #3 - equivalent to ".i"]
I'm
.           [quits insert mode - "dot" set to "3"]
4c          [delete line #4 - replace with new text]
lost
!!!
.           [quits change mode - "dot" set to "5"]
5,.+1d      [deletes lines #5 and #6 - "dot" set to "5"]
1,2m5       [moves lines #1 and #2 after line #5 --- "." = "5"]
7tS-1       [copies line #7 after line "S-1" = "6"]
w           [writes buffer onto file "number"]
35
q           [quits editor]
```

Check your interpretation of the commands with my current contents of file "number":

```
I'm
lost
four
one
two
five
six
six
```

One final warning about the "a," "i" and "c" commands ----- the terminating "." is easily forgotten, and will cause garbage to be erroneously added to the buffer, until you realize your error. At this point "bite the bullet" and make use of the "d" command (carefully).

2.5 Text Searches and Special Characters

Searching Forwards Many times you wish to search for a line(s) containing a particular character string -- this is called "Text Searching". An editor command:

```
/string_of_characters_you_want_to_find/
```

will locate the *next* occurrence of the string of characters between the "slashes" (called a "regular expression," henceforth abbreviated as "RE"). The search starts with the next line after the search command is given (i.e., line .+1), continues to the end of the file (i.e., line S), "wraps around" to line #1 and continues to the current "." line. The *first* line containing a match to the RE is set to "dot," and the line is automatically printed for verification. If many lines in the buffer contain the "regular expression," only the *first* one encountered will be found by the search command. If no match is found, the editor will prompt with "?". The "regular expression" may be composed of any printable or non-printable characters, with some characters ("metacharacters") having special meaning, as delineated below.

Searching Backwards We may also search for lines in a reverse order, i.e., start our search at ".-1", continue backwards to "1", "wrap around" to "$" and continue backwards to "." . This reverse search is instituted by a "regular expression" enclosed by "question marks," as:

```
?string_of_characters_for_reverse_search?
```

Printing with Line Numbers (n) While context searching is the preferred method of locating lines, you may wish to issue the NUMBER command of the form:

```
7,20n
```

which will print the lines indicated ("7,20" may be replaced by any valid line range), each prefixed with its *line number* followed by a "tab".

The command:

```
n
```

prints the value of "dot," followed by the current line itself. Also the command:

```
,n
```

may be of interest; in this case, the contents of the entire editor buffer will be printed, each line prefixed by its line number.

Finally, it should be noted that an "n" may be added as a *suffix* to any command, which could accept a "p" or "l" as a suffix.

Context Addressing In addition to finding lines containing specific character strings, the search command itself may be substituted for a line number (called "context addressing"), when using the READ, APPEND, DELETE, PRINT, CHANGE, INSERT, MOVE, COPY or WRITE commands. The following commands are all valid:

```
/help/d
/new/,/old/+3p
?five?i
```

Semicolon Separator (;) Assume a file of the following form:

```
  ....        ["...." lines do not contain "a" or "b"]
  ....
  ab
  ....
  ....
  bc
  ....
  ....
```

Starting at line #1, we might expect the command

```
  /a/,/b/p
```

to print all lines between "ab" and "bc" inclusive. Actually, only the line "ab" is printed. This is due to the fact that searches for "a" and "b" *both* start at the same point, and "/a/" and "/b/" both find the same line, "ab". Worse, if a line *before* "ab" contained a "b," the entire print command would be in error. The problem is that the "comma" separator for line numbers doesn't set "dot" as each address is processed ---- "dot" is reset only after a command is actually executed.

In the editor, the "semicolon" can be used just as a "comma" separator of line numbers, but the use of ";" forces "dot" to be reset to the first line number, after it is evaluated. The command:

```
  /a/;/b/p
```

prints the range of lines from "ab" to "bc" inclusive. The line containing "a" is found first, "dot" is changed to that line number, and *then* the search command for "b" is instituted, starting at the next line.

Another interesting use for the "semicolon" is a command like:

```
  23;/a/
```

which would generate a search for the string "a" beginning at line #24. The initial "23" causes "dot" to be set to that line, and the search then proceeds with the next line. You can also use a command like

```
  0;/a/
```

to start a search at the beginning of a file (this is one of the few places where "0" can be used as a legal line number).

2.6 Metacharacters in Regular Expressions

Some characters, called "metacharacters," have special meaning when included in "regular expressions; they are the "period (.)," "left bracket ([)," "asterisk (*)," "backslash/brace (\{)" combination, "circumflex (^)," "dollar sign ($)," "pound sign (#)," "at-sign (@)" and "backslash (\)".

BACKSLASH Metacharacter (\) If you desire to eliminate the special meaning of any character, the "backslash" (called the "escape character") can be used. Simply preceding the "metacharacter" by the "backslash" will accomplish this.

The "pound sign," which is a "global metacharacter," has the property of "erasing" the previous character typed anywhere in a UNIX session; the following line:

```
IX#\#2
```

is equivalent to "I#2", as the first "#" erases the previous character, but the second grouping "\#" removes the special meaning of the "pound sign" and introduces the "#" character into the buffer. Similarly, for the "at-sign," the sequence "\@" will return you to a new line, making you think you've erased the line; in reality, the text will contain the "@" symbol and you are actually typing on the same line.

Also, if you wish to remove the special meaning of the "backslash" character itself, the sequence "\\" will accomplish this. If we wish to search for an unlikely sequence of text "#\", the proper search command is:

```
/\#\\/
```

The "backslash" found at the end of a line has special meaning also; it removes the meaning of the "newline" ("carriage return") character, and you are in effect typing continuously on the same line.

Finally, in the search mode, we would have difficulty trying to match a "slash", such as:

```
/ab/c/
```

would *not* match "ab/c"; the "slash" between "b" and "c" would indicate a complete search command, followed by "c/" (garbage), and a "?" would be printed indicating an error. A proper search could be accomplished by:

```
/ab\/c/
```

since the combination ""\/" is equivalent to searching for the "/" character, once its special meaning is removed.

PERIOD Metacharacter (.) The "period metacharacter" is a RE standing for *any* single character (except "new line"). Thus the search

```
/a.b/
```

finds any line where "a" and "b" are separated by any single character, and would match any of the following strings:

```
a-b
abb
a-b
a.b
a b
```

If you wish to locate the text string "a.b" only, you must use the "backslash" to remove the special meaning of the "." metacharacter. i.e.,

```
/a\.b/
```

The "." is very useful in matching "metacharacters" and is very useful in eliminating non-printable characters, as will be seen in Section 2.7.

DOLLAR SIGN Metacharacter ($) The "dollar sign," when used in an RE represents the "end of the line" (as opposed to "end of buffer" when used in line numbering). Revisiting our original file "number," which is repeared here for convenience:

```
one
two
three
four
five
six
```

could locate the line "two" with the search command:

```
/o$/
```

since only an "o" at the end of a line is sought. The search /o/ could match "one", "two" or "four". The "$" must be used as the *last* character in a "regular expression" to retain its special meaning.

CIRCUMFLEX Metacharacter (^) A related metacharacter, "^" (the "circumflex" or "hat" or "caret" symbol), will restrict your searches to characters appearing at the "beginning of a line". The search command:

```
/^o/
```

would match "one" in the file "number", but not "two" or "four". The "^" must be used as the first character in a "regular expression" to retain its special meaning.

Locating Blank Lines Also the search

```
/^$/
```

may be of interest to you. It is seeking a "blank line," i.e., *no* characters between the *beginning* of the line (represented by the "^" metacharacter) and the *end* of the line (represented by the "$" metacharacter).

Matching a Line Exactly Similarly, the search:

```
/^line of text$/
```

provides special characteristics. It is seeking a line which will match "line of text" (the entire string between the "^" and "$" metacharacters) *exactly*, on a character-for-character basis.

Other uses for the special meaning of "$" and "^" will be discussed later in Section 2.7. Also remember the special meaning of "$" and "^" may be "turned off" by preceding them with a "backslash".

ASTERISK Metacharacter (*) Next, a character in an RE *followed* by a "*" ("asterisk" or "star") stands for *any* number of *consecutive* occurrences of that character. Be aware that as "any number" means *zero* or more occurrences; a misinterpretation here can lead to many problems. Assume we are looking for a line containing a character string "abbbbbbbbbbc"; certainly a search command.

```
/abbbbbbbbbbc/
```

would accomplish our aims, but the command

```
/ab*c/
```

is far simpler. If the above search encountered a line containing the string "ac" first, a match would have occurred; the string contains an "a" followed by a "c", separated by *no* "b's"; *no* "b's" is a legitimate match for the couplet "b*" in the regular expression. Again, the special meaning of "*" may be "turned off" by preceding it with a "backslash".

BACKSLASH/BRACE Metacharacters (\{...\}) *Any* character in a "regular expression" followed by "\{m\}", "\{m,\}", or "\{m,n\}", where "m" and "n" are non-negative integers less than 256, has special meaning. The grouping "\{m\}" indicates a match of *exactly* "m" occurrences of the character preceding this special sequence in the regular expression; "\{m,\}" matches *at least* "m" occurrences of that character; "\{m,n\}" matches any number of occurrences of that character in the *range* "m" through "n" inclusive.

Some examples are:

```
/aB\{2\}/     [Search for "a" followed by exactly two "B's"]
/40\{3,\}/    [Search for "4" followed by three or more "0's"]
/X\{5,9\}/    [Search for five to nine "X's"]
```

These "metacharacters" are cumbersome to type and use, and are avoided by experienced users, except in special circumstances. There is also another even more cumbersome RE enclosed by the character sequences "\(" and "\)", which has only highly specialized uses; the reader should consult the "ed" manual page to become aware of its properties.

SQUARE BRACKETS Metacharacter ([...]) The last "metacharacter" of interest is the "left square bracket" or "[", which is always coupled with a closing "right bracket". The construction

```
/[12ab]/
```

will cause a search for *any* ofthe *single* characters enclosed in the "square brackets" ("1", "2", "a" or "b"). Similarly,

```
/a[0123456789]*/
```

searches for a line containing a string, "a" followed by zero or more digits.

Matching Lexical Ranges The command

```
/a[0-9]*/
```

accomplishes the same thing; the "minus" may be used to indicate a range of lexically consecutive ASCII characters, such as: [0-9] is equivalent to [0123456789], [a-f] is equivalent to [abcdef], and [W-Z] is equivalent to [WXYZ]. Compound sequences are also allowed, i.e., an RE such as:

```
[A-Za-z6-9]
```

represents a *single* character, i.e., an upper or lower case letter or a digit in the range "6" to "9".

Exception Metacharacter If the "circumflex" is used as the *first* character of the string *within* "square brackets," then any single character, *except* "new line" ("carriage return") and the *remaining* characters in the string, is matched; the "^" at any position other than the first has no special meaning. The RE

```
[^ad-g]
```

represents any character *except* "new line", "a", "d", "e", "f" or "g".

Square Brackets Anomalies Also, the characters "*", "[", "\" and "." have no special meaning when enclosed by "square brackets". Confused --- consider the search:

```
/^[^\^]/
```

which seeks a line which does *not begin* with a "\" or "^".

A final comment is that the "-" loses its special meaning if it is first in a string (after an initial "^", if any) enclosed in "square brackets," or if it is the last character in that string; also, the "right bracket]" will *not* terminate a string if it is the first character following the opening "[" (after an initial "^", if any). Remember, after proper interpretation, the string within "square brackets" represents a single character; verify that

```
  /[][d-w]/
```

will match "]", "[", or any lower case letter in the range "d" through "w": it matches only a *single* character, as there is only *one* set of valid "square brackets".

2.7 Modifying Existing Test

Substitution Command (s) Probably one of the most important commands is SUBSTITUTE. This command ("s") is used to change strings of characters within a line or group of lines. The basic form is:

```
  s/regular_expression/replacement_text/
```

which will replace the "regular expression" denoted (possibly containing "metacharacters," as previously described) by the "replacement text" in the current "dot" line. If we have a current line "I love telephones" and issue the command:

```
  s/e t/e Bell t/p
```

the line is changed to "I love Bell telephones".

The optional appended "p" will cause the corrected line to be printed for your inspection. Note that "p" (or "l" for "list," or "n" for "number") may be appended to many commands to cause printing of the current line after the command has been executed, such as ".dp" will delete the current line and print the new "dot" line.

Metacharacters in REs Revisited Using "regular expression metacharacters" in SUBSTITUTE commands is almost essential to good editing style. Let's consider various common examples using these special characters. To simplify matters, in *all* the examples below, assume we are *always* making substitutions in the line:

```
  I love telephones
```

unless clearly stated otherwise.

The command

```
  s/^/Boy, do /p
```

will print the corrected line "Boy, do I love telephones": the "circumflex" alone indicated the replacement is to be placed at the *beginning* of the line.

Similarly

```
  s/S/!!!/
```

will add "!!!" to the *end* of the line, due to the special meaning of "S" in the regular expression.

The command:

```
  s/t.*/girls/p
```

will print "I love girls" which is due to the regular expression "t.*" (character "t" followed by zero or more of *any* character -- the ".*" metacharacter couplet is very useful when replacing the remainder of a line after a certain point).

The command:

```
  s/[oe]/x/p
```

will print "I lxve telephones"; the regular expression "[oe]" searches the line for either an "o" or "e", and the SUBSTITUTE command will act *only* upon the *first* match found in the line.

Substituting Globally Across a Line We may also change *all* occurrences of a regular expression in a line by appending the SUBSTITUTE command with the letter "g", (representing "globally, across the line") for example:

```
  s/[oe]/x/gp
```

will change "I love telephones" to "I lxvx txlxphxnxs". Remember, without the "g" suffix, only the *first* match in the line will be affected by the SUBSTITUTE command. Also note that the additional appending "p" will cause printing to occur, and only the order "gp" is allowed ("pg" is illegal).

Asterisk Metacharacter Anomalies Using the "g" suffix may cause problems when using the "asterisk" metacharacter in a regular expression. Consider the line "I lov!! t!l!!!phon!!s", which would require all *single* or *consecutive* occurrences of "!" to be replaced by a single "e". The proper substitution is:

```
  s/!!*/e/g
```

which would cause a conversion to "I love telephones". The regular expression "!*" would yield quite different results, i.e., "eIe eleoeve etelepeheoenese", as UNIX notes between every pair of "non-!" characters, there are *zero* or more "!'s" and, therefore, an "e" is substituted.

For the same line, "I lov!! t!l!!!phon!!s", the substitute command:

```
  s/!*/e/
```

may also yield unexpected results. The line will be converted to "eI lov!! t!l!!!phon!!s"; the editor determines that *no* "!'s" exist between the beginning of the line (a "new line" character) and the first character "I", and the substitution is made. The wording in the UNIX User's Manual is vague (at best): "If there is any choice, the *longest leftmost* string that permits a match is chosen"; this should be interpreted as "the *leftmost* match is chosen, *regardless* of how long the consecutive string is". An illustrative example is that the lines:

```
  h?lp???
  h??lp???
  h???lp???
  h????lp???
```

would all be converted to "help??? by the command:

```
  s/??*/e/
```

Only the *first* (or leftmost) occurrence of "?", followed by *zero* or more "?'s" will be affected, *not* the *longest* consecutive grouping of "?'s" in the line.

Breaking Up a Line There may be cases when you wish to "break a line in two". For the line "I love telephones", the command:

```
  s/e t/e\(CARRIAGE RETURN)
  t/
```

will create *two* separate lines "I love" and "telephones". The "backslash" before the "CARRIAGE RETURN" will cause the "new line character" to be transmitted to UNIX, *as replacement text*, although you are *physically* placed on a second line to complete the the SUBSTITUTE command.

For those who want to get fancy, this process may be used repeatedly in the same substitution, such as:

```
  s/e t/e\(CARRIAGE RETURN)
  Bell\(CARRIAGE RETURN)
  t/
```

will change the line "I love telephones" to *three* lines, i.e.,

```
  I love
  Bell
  telephones
```

Multiline Substitutions A more general form is:

```
  1,$s/regular_expression/replacement_text/p
```

which causes a *possible* substitution in *all* lines in the buffer; the substitution is made only if the "regular expression" is matched in any given line. Of course, the range "1,$" is illustrative and may be replaced by any valid range of line numbers, including those containing "dot" and relative or text addresses. You may also choose a single line number, or if none is indicated, a default to "dot" is made. Note, the "p" suffix, in this multiline case, will cause UNIX to print *only* the *last* line in which a substitution is made.

Removing Strings from a Line It is extremely useful to *eliminate* certain characters within a line; the command

```
  4,9s/^ *//p
```

will eliminate any number of "blanks" occurring at the beginning of lines #4 through #9; the "//" replacement (with no internal spaces) means "replace with nothing".

Removing Nonprintable Characters Another tricky point of interest is the case when a line *listed* ("l" not "p" command) indicates an erroneous non-printable character, such as:

```
  I love tel\07ephones
```

The "\07" represents the "bell" character ---- the question is how to remove it. The substitution

```
  s/\\07//
```

will not work. However,

```
  s/l.e/le/
```

will work, as the "." represents any single character (even non-printable ones) between the "l" and "e".

Metacharacters in Replacement Text Other than the characters "#" and "@," which have special meaning anywhere in UNIX programming (unless "escaped" by a preceding "backslash"), characters within the *replacement text* with special meaning are the "backslash (\)," the "slash (/)," the "percent sign (%)," and the "ampersand (&)".

BACKSLASH Metacharacter Revisited Consider our line "I love telephones", and the command:

```
  s/lo/l/o/p
```

We wanted to change the line to "I l/ove telephones" (don't ask me why!), but instead the editor will print "?". This is due to the fact that UNIX assumed the replacement string ended with the third "slash" --- to place a "/" literally in the replacement text, it must be "escaped," such as:

```
  s/lo/l\/o/p
```

The "backslash" itself may be entered into text by using the couplet "\\".

AMPERSAND Metacharacter (&) The "ampersand" in a *replacement text* (not in an RE) represents the *entire* "regular expression" just matched, i.e.,

```
  s/love/& \& &/p
```

will cause the line "I love & love telephones" to be printed. The first and third "&" in the replacement text cause the RE "love" to be substituted while the second "ampersand" is "escaped" (by "\&" couplet) and is a literal replacement.

Repeated REs in Substitutions Another time-saver is illustrated by the following command,

```
/love/s//like/p
```

this command will search for a line containing the string "love", set "dot" to this line number, and obey the SUBSTITUTE command. The "s" command will change "love" to "like" in that line, because "two consecutive slashes (//)," or the "null regular expression," is shorthand for "the last regular expression used".

Repeated Searches The use of this shorthand notation may also be used in repeated searches. Assume the search "/love/" is instituted and the wrong line is found -- the text wanted is another line containing the string "love". Simply type "//" for another forward search or "??" for a reverse search, as the editor remembers the most recent pattern, or "regular expression," searched for, (in this case, "love"). This procedure may be repeated as many times as necessary.

Repeated Replacement Text If "%" is the *only* character in the *replacement*, the replacement text in the *most recent* substitute command is used as the replacement text in the current substitute command. The "%" loses its special meaning when the replacement text is *more* than one character long, or when it is escaped by a "backslash" (i.e., the couplet "\%").

Undoing Bad Substitutions (u) An extremely useful command is the UNDO command, or simply:

```
u
```

which negates the effect of the last SUBSTITUTE command issued, if you really "botch" things up. You should use the UNDO command *immediately* after the "s-command" has been executed, if you want it to work (as some intervening commands may confuse UNIX and the last substitution may not be reversible --- nobody is perfect). In a *multiline* substitution, *all* affected lines will revert back to their "original form".

2.8 Global Modifications

Basic Global Command (g) The GLOBAL command "g" is used to perform one or more editing commands on lines containing a specified pattern (regular expression). The command:

```
g/love/p
```

will print *all* lines in the buffer containing the string "love", while

```
  4,20g/love/p
```

will print only the matched lines in the indicated range (line #4 through #20, in this case). In either of the above commands, the "p" suffix is optional, as matched lines will be automatically printed by default, if no other command follows "g/RE/".

Let's consider a more sophisticated global command, such as

```
  6,11g/love/s//like/gp
```

The first step is to mark every line, in the range line #6 to #11, that contains the indicated pattern "love". Then for every such line, the command following is executed, with "." initially set to that line. In this case *all* occurrences of "love" in that line are replaced by "like", and the line is printed. The string "love" is the RE of the "s" command (represented by the "//"), since the last regular expression used was "love" in the global search pattern; *all* occurrences in a line are changed because of the "gp" suffix on the "s" command, as explained previously.

Basic Exclusion Command (v) We have a diametrically opposed command EXCLUDE "v" which acts upon all lines *not* containing the pattern indicated by the global RE. For example:

```
  v/love/p
```

prints out all lines *not* containing the string "love".

Multiline Global Commands What about multi-line or multiple commands under control of a global command? Consider

```
  g/love/s/love/like/\
  s/telephones/radar/
```

This command will search the entire buffer for lines containing the string "love". For each such line found, the first occurrence of "love" is changed to "like", and the first occurrence of "telephones" is changed to "radar". The "backslash" at the end of the first line tells the editor that another command line will follow; this may be repeated for as many lines as desired. but the final line in the global command must *not* contain the terminating "\". The entire buffer is searched globally because, in the absence of any line number preceding "g," the range "1,$" is the default. Note that "//" was not used instead of "/love/" in the first substitute command, because, if more than one line contained "love", the *last* "regular expression" remembered, after the first line's substitution is complete, is "telephones" ---- be careful; this is due to the fact that the "g/love/" portion of the command is executed *first* to mark all lines containing the string "love", and *then* the SUBSTITUTE commands are sequentially executed for each matched line, *never* to return to the RE "love" associated with the "g" command.

A (perhaps obvious) note here is that you *can not* "split" lines as shown in Section 2.7, as a line terminated with a "backslash" will *not* be interpreted properly by the global command.

You may also use multi-line commands such as "a," "i" and "c", as:

```
v/love/a\
I really hate telephones!\
.
```

This command seeks all lines in the buffer *not* containing the string "love", and appends *each* one with the line "I really hate telephones!". Remember the terminating "\" must appear on all lines, except the last, in the global command.

As a final note, you may *not* nest another "g" or "v" command within a current global command.

Locating Unwanted Control Characters One of the useful attributes of global searching is the ability to locate all lines within a file that contain unwanted "control" characters. Try the following command:

```
g/[^SPACE—~TAB]/.=
```

When typing in this command, *SPACE* should be replaced by a single *blank space*, and *TAB* should be replaced by the *tab character*, i.e., depressing the "CONTROL" and "I" keys simultaneously. Since the RE starts with a "^", it will match any line containing a character. *other than* ASCII characters in the range *SPACE* to "~", or *TAB*; this represents all *control* characters, except *TAB*. The ".=" command to be executed every time a match is found will provide a list of *numbers* of all lines containing (probably) undesirable control characters. These lines can now be investigated using the list command, as described in Section 2.3.

Undoing Global Mistakes (u)
If the UNDO command is issued *immediately* after any "global" execution, your buffer will be restored to its "original form" (prior to the issuance of the "global" or "exclude" command).

In fact, the UNDO command has been made so powerful in Version 4.0 of UNIX, that it reverses the affect of the *most recent command* that modified the buffer. These include the SUBSTITUTE and GLOBAL commands mentioned, as well as editor commands APPEND, CHANGE, DELETE, INSERT, MOVE, READ or COPY.

Interactive Global Actions (G) An example of the INTERACTIVE GLOBAL command is:

```
5,20G/love/
```

This command will search the indicated line range ("5,20" in our example), and mark every line that contains a match to the "regular expression" (in this case, "/love/"); should the line range be omitted, the entire editor buffer is searched for matches. Then, sequentially, every matched line is printed, "." is changed to that line, and a *single* command line may be entered and executed (note that "a," "i," "c" or any global command will *not* be accepted by "G"). After the execution of that command, the process is repeated for the next marked line. The "carriage return" ("new line") acts as a "null" command; an "&" alone will cause the re-execution of the most recent command executed during the *current* invocation of the "G" command. The "G" command can be terminated by issuing an "interrupt" signal (DEL or BREAK key); also, any error in an interactive command you issue will cause UNIX to print a "?" and exit "G".

Interactive Exclusion Actions (V) This command is identical to the INTERACTIVE GLOBAL command, *except* that all lines *not* matching the designated "regular expression" are marked.

2.9 Miscellaneous Editor Commands

There are other important editor commands which deserve brief mention.

Temporarily Leaving the Editor (!) There are occasions when you may wish to issue a UNIX level command, but you want the editor buffer to remain intact for further work. If you type:

```
!any_UNIX_command
```

your current editing state will be suspended, and the UNIX command indicated will be executed. When this UNIX command is completed, the editor will prompt with another "!," at which time you may continue editing (note: "dot" is unchanged by this process). An example is

```
ed
{editing session}
!date
{UNIX prints the date}
!
{editing continued}
q
```

Using an Editor Prompt (P) If we issue the command:

```
P    [that's capital "P"]
```

the editor will prompt with an "*" for all subsequent commands. This "P" command alternately turns this prompt mode on and off; it is initially off. This property is very useful when using the "a," "i" or "c" commands, as you are sure the "terminating dot" has been issued, when you receive the "*" editor prompt.

Obtaining Help (h, H) Occasionally, you will obtain "?" diagnostic outputs, by making an error in a command issued, or attempting to quit (or edit) the current editor buffer before writing its contents unto a file. If you issue the command:

```
h
```

a short error message will be printed, explaining the most recent "?" diagnostic. For more "help," you may issue the following command upon entry to the editor:

```
H
```

and *all* "?" diagnostics will be explained as they are issued. Issuing the "H" command alternately turns this mode "on" and "off"; initially, it is "off".

Unconditional EDIT and QUIT Commands (E, Q) You may use the commands:

```
E
```

and

```
Q
```

which are analogous to the EDIT ("e") and QUIT ("q") commands previously described, *except* that the editor does *not* check to see if any changes have been made in the editor buffer *after* the last "w" command.

Joining Lines (j) The JOIN command "joins" two *contiguous* lines by removing the "new line" character between them, i.e.,

```
4,5j
```

will combine line #4 and #5 into a single line #4. If no line numbers are indicated, then the range (.,.+1) is assumed.

Marking Lines (k) The MARK ("k") command is useful to label special lines with a *lower* case letter address. A command

```
5ka
```

will identify the letter "a" with current line #5; if no line number is given, "dot" is the default, and any other lower case letter may be used in place of "a". The address form "'a" is thereafter an alternate valid line number.

Delimiter Replacement in Substitutions One technique, which is useful when the "replacement text" includes many "slashes," is to use a command similar to:

```
s!love!///!
```

which will replace "love" by "///". We have removed the special meaning of the "slash" by using "!" as a "delimiter" in the SUBSTITUTE command. The "!" could have been replaced by any character (other than a "space" or "new line") which does not appear in the "regular expression" or "replacement text," and is not a global metacharacter.

Missing Delimiters If the closing *delimiter* (/) of a RE or replacement text would normally be the *last* character before the CARRIAGE RETURN, that delimiter may be *omitted*. In this case, the closing delimiter is automatically supplied by UNIX, and the affected lines are printed. Although it may be better editing style to use complete expressions as previously described, forgetting the trailing "/" is one of the most common errors in editing, and UNIX saves you in many cases; if UNIX makes a substitution you really don't want, the UNDO command will save you. In fact, as you gain proficiency in editing, this concept can save the time necessary to type one or two characters, repeatedly (if you use one finger to type, like I do, the time savings could be considerable).

To illustrate this point, the following pairs of commands are equivalent:

```
s/love/hate       s/love/hate/p
s/love/           s/love//p      (replace with nothing)
g/love            g/love/p
/love             /love/         ("p" suffix implied)
/                 //             (repeated forward search)
?love             ?love?         ("p" suffix implied)
G/love            G/love/        (matched lines printed via "G")
```

2.10 Visual Editors

High speed data communication and display (CRT type) terminals leads to new concepts of text editing, the most common task of many computer users. In contrast to the normal "context" editor previously described, an interactive *visual* editor is presented with a display of the contents of a portion (usually more than 16 lines) of the buffer being worked on. By moving the cursor to appropriate points in the display page and utilizing one of a rich set of commands, characters, words or lines may be added to or deleted from the existing display; you benefit from instant feedback, as changes to the buffer appear on the screen immediately. You may read from and write to any UNIX file. You may do searches and change the buffer area you are working on at will. It's a beautiful tool!!!

Now that I've got your mouth to water, I'll try to explain why a "visual" editor will *not* be discussed in detail in this document:

- There are many visual editors installed by local "wizards." Their names include "ved," "hpved," "dmved," "vi," "ex," "emacs" and probably dozens of others that I am not aware of; not all editors are available on all UNIX systems, and many times they are available only from user's private directories. They are all *unsupported* tools.

- You must have a *display* (a CRT type, such as an ADM3 or HP) terminal; visual editors will not work on printing terminals. This is a good reason to learn the "ordinary" editor, which is effective on both printing and display type terminals (did you every try to take a CRT terminal home?).

- You should have a *high-speed* modem. Although you can use a visual editor at a speed of 30 characters per second generation (300 baud), it becomes quite tedious to wait long periods of time for the screen to fill up every time you change your working buffer. I would suggest a 1200 baud modem or high speed dedicated link to the UNIX system.

- There is a cost factor. Visual editors generally require from 3 to 20 times the computer resources than the "normal" editor, and this assumes you do not have a private copy of a particular editor in your login (each editor is several million characters of executable code in length).

Therefore, this author intends to include a description of the "visual" editor, once *one* type is fully supported as a standard product (hopefully by Version 5.0 of UNIX). Until then, the best bet is to ask users of these editors in your area for sources of documentation. and, possibly, some "getting started" help (documentation for some of these editors is sparse, and, in many cases, outdated or not in agreement with versions that have been modified locally).

3. BASIC UNIX COMMANDS

3.1 Listing Filenames and the File Structure

Simple List of Filenames (ls) Once you have created files, you may wish to list their *names* (not contents) using the LIST ("ls") command. Assume we have the files "junk," "number" and "stocks" in our "login directory"; for now, assume your "login directory" is simply a collection of files you've created via the "text editor". The command:

```
$ls                 [you type "ls" in response to "$" prompt]
junk                [UNIX
number                   lists
stocks                        filenames]
```

will cause filenames to be printed *alphabetically* (names beginning with numerals or strange characters will be listed first). It should be noted that *all* commands are issued in response to the "$" prompt from UNIX, and this fact will be *implicit* in most future examples.

Listing All Filenames (ls -a) If a filename begins with a "period" (called a "dot file"), the "ls" command, as shown above, will *not* print it ----- to list *all* filenames, an "ls -a" command is required, noting the space between "s" and "-" is necessary. The list produced will be headed by "." and "..", which have special meanings, as explained in Section 3.6.

Chronological Filename Listing (ls -t) With an optional argument "-t", a *time-ordered* sort is provided, i.e., files are listed in the order in which they were last changed or created, most recent first.

```
$ls -t              [space between "s" and "-" necessary]
junk                [assumes "junk" was worked on last]
stocks
number
```

Producing Long Listings (ls -l) Another option "-l" (that's lower case "L," *not* number "1") will give a *long listing* (a lot of useful information), such as:

```
$ls -l         [again, "space" between "s" and "-"]
total 4
-rw-rw-rw-     1  owner group_owner  31  Sep22 12:56  junk
-rwxrwx---     1  owner group_owner  28  Aug30  1:26  number
-r-xrwx-w-     2  owner group_owner 941  Jan1   3:12  stocks
```

Of great interest is the first 10 characters taken as a group. The first character printed is a "-" for an *ordinary file*, or "d" for a *directory*; other special files may also be designated, but they are of little interest to the average user. The next three characters indicates "what the *owner* can do with the file"; the owner may "read" (r) the contents of the file, modify or "write" (w) onto it, "execute" (x) it (if the file is executable), or a "-" indicates the corresponding permission is denied. The next three characters refer to permissions of the owner's "group" (which may be administratively formed). while the last three indicate permissions of all "others". Note that the concepts of "directory," "permissions" and "groups" will be discussed in detail shortly.

The next number will be explained later (Section 3.5) and need not concern you now. The entry "owner" symbolically represents the owner of the named file, while "group owner" designates the *group* that owns the file (for the average user, the "owner" and "group_owner" entries will be identical). The number following (i.e., 31, 28 and 941) indicates the number of characters in the corresponding file. Next comes the date and time of the last change to the file (or directory), and finally, the file (or directory) name is listed.

The entry "total 4" line before the "long listing" indicates the number of 512 character "blocks" used to store the listed files.

Producing Owner Listings (ls -o) Another useful option is "-o" (that's lower case "O," *not* the number "0"). It provides a "long listing," as described. except that *group* information (which many user's don't give a damn about anyway) is omitted.

Determining File Block Size (ls -s) As mentioned previously, a "block" of 512 character is a unit of file *size*. To determine the "block size" of files, use a "-s" option; the *number* of blocks will be the *first* item printed, followed by all other file data.

Combined Filename List Options The options described may be combined (in any order), such as:

```
  ls -lta
```

will cause a *long* listing of *all* files via a *time-ordered* sort.

Terminating a Listing Finally, if you get tired of a listing, hitting the "DEL(ETE)" or "BREAK" key will cause printing to cease and return you to the UNIX command level.

If you're working on a CRT type terminal you may wish to stop a listing *temporarily*, before it runs off the screen; the CONTROL-S alternately stops and starts the printing of the list for you.

Directories and the File Tree Structure Each user has a "home directory," identified by his/her login name (user ID). When you create a new file, unless you take special action, it resides in your "home (or login) directory," and is unrelated to any other file of the same name that might exist in someone else's directory.

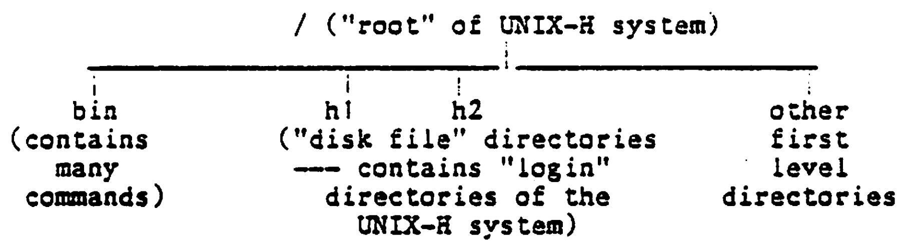
A *directory*, in general, represents a collection of files (and possibly other subdirectories); explicitly, it contains the names of the files/directories in it and control information necessary to access those files. An *ordinary file* is simply a collection of data or characters stored on UNIX.

The set of all files UNIX knows about are organized into a large "tree" structure. A "master UNIX directory" is the "root" of this "tree," and after several levels of "branches" (subdirectories), we arrive at the "leaves of the tree" (the ordinary files). The "root" directory, using a UNIX convention, is designated by "/". The following section will hopefully clarify the concept of the "tree structure."

Try the command:

```
ls /
```

which will list various files/directories in the "root" directory, including "bin" and "h1" (assuming you are working on the UNIX-H system). If the argument of an "ls" command is a "directory" name, the names of all files/subdirectories within that designated directory will be listed, according to the options "-l", "-t", "-a", "-o" which may be invoked. We may look at a "crude" diagram to illustrate this portion of the "tree," i.e.,


/ ("root" of UNIX-H system)
bin
(contains many commands)
h1
h2
("disk file" directories --- contains "login" directories of the UNIX-H system)
other first level directories

Next try the command

```
ls /bin
```

which will list the files and directories of a first branch directory "bin." You should recognize some of the filenames listed, like "ed" and "ls," as this directory contains the most commonly used commands. Other commands are located in the directories "/usr/bin" and "/usr/lbin."

Next try listing the contents of "branch h1" via the command:

```
ls /h1
```

and among the directories listed, you should note your own "login name." Directory "h1" is simply an illustrative "parent directory"; if your "login" is on a different file system, substitute its name for "h1." It should be pointed out here that the "h" identifies the physical computer (a PDP-11 or VAX system), while the digit "1" identifies a particular disk file on that system. Every directory (except the "root") has a "parent." If "directory_2" is contained within "directory_1," then "directory_1" is referred to as the *parent* of "directory_2."

Then proceed up the "tree" and try:

```
  ls /h1/your_login_name
```

which will list your personal files and subdirectories and is entirely equivalent to the command:

```
  ls
```

which lists the contents of your "current" directory, as a default, as *no* directory name argument is given. Your *current working* directory is simply whichever directory you are currently working in and issuing commands from (presumably, you are working in your "login" directory, but this need not be the case, as will be shown in Section 3.8).

Again, the above examples will work *only* if your login is on the "h1" disk file system; otherwise, you must substitute the name of *your* disk file, if you want to try these commands. To find out the name of your disk file (if you don't know it), issue the following sequence of commands while in your *login* directory:

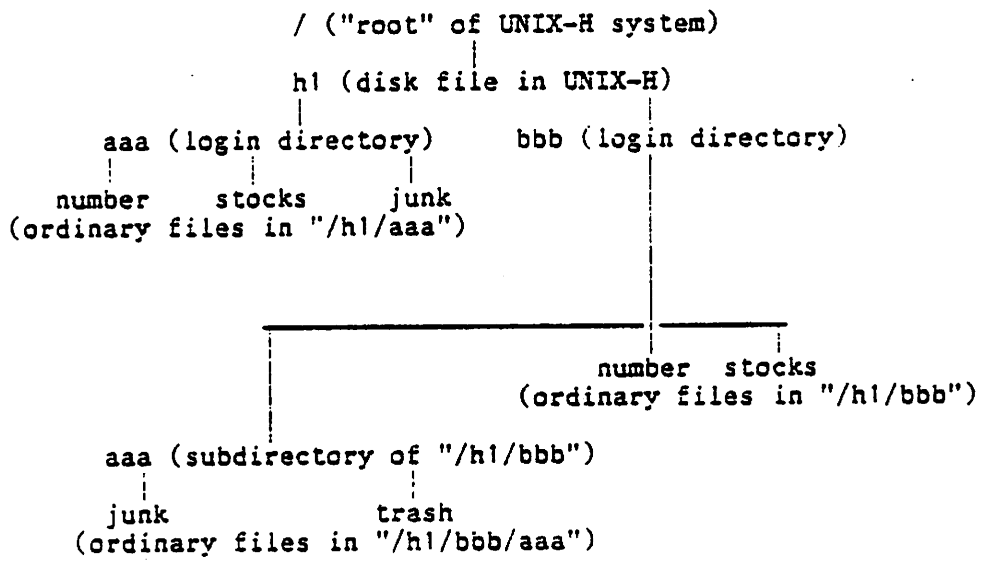
```
  cd ..
  pwd
  cd
```

and UNIX will print out your disk file name (the "cd" and "pwd" command will be explained in detail in Section 3.8).

Pathnames It is a universal rule in UNIX that anywhere you can use an ordinary filename, you can use a "full pathname." A "full pathname" obviously implies the *full* name of the *path* you have to follow through the "tree of directories," starting from the "root" ending with the desired file or directory name. When using the *full* pathname, separate each directory or file with a "slash"; an initial "/" is necessary to start the search in the "root" directory.

If that *leading* "/" is missing, a "branch" from your "current" directory is implied, as will be illustrated in the following examples (up to now, the "current" and "login" directories were assumed to be the same, but, as stated previously, they need *not* be).

Comprehensive List Command Example For the purposes of illustration, consider the following "tree structure":


/ ("root" of UNIX-H system)
h1 (disk file in UNIX-H)
aaa (login directory)
bbb (login directory)
number
stocks
junk
(ordinary files in "/h1/aaa")
number
stocks
(ordinary files in "/h1/bbb")
aaa (subdirectory of "/h1/bbb")
junk
trash
(ordinary files in "/h1/bbb/aaa")

Note that many names are repeated (on purpose), such as "aaa," which may be a "login" directory in the "h1" disk file system, or a "subdirectory" under "login" directory "bbb" ---- these are *distinct* directories that can be *uniquely* identified by their *pathnames*. In the following examples, *always* assume our "current" (working) directory is login "bbb" (*full* pathname "/h1/bbb").

Now that we are experts on the file structure, a general form of the "ls" command may be defined:

```
  ls options directory_name
```

The "options" are "-a", "-t", "-o", "-s" and "-l" previously described or more to follow (including some combination of these). The "directory name" may be a *full* or *partial* pathname. If a *partial* pathname (a pathname which does *not* start at the "root," designated by the leading "/") is indicated, the search for "directory_name" will be a "branch" from the current working directory; if "directory name" is *omitted*, the contents of the "current" working directory will be listed, as previously shown. In any case, the command will produce a list of files and directories *contained within* "directory_name." The command:

```
  ls aaa
```

assumes a "branch" from our current directory "/h1/bbb," and is entirely equivalent to:

```
  ls /h1/bbb/aaa
```

and will list the filenames "junk" and "trash." belonging to *subdirectory* "aaa" in *login* directory "bbb." However, a *full* pathname command:

```
  ls /h1/aaa
```

will list "junk," "number" and "stocks," the *ordinary files* contained within *login* directory "aaa."

If you are confused at this point, please reread this section, as it is *imperative* that you understand the concepts of "tree," "branch" and "pathname" to properly execute commands.

Directory Status Listing (ls -d) If the command contains the "-d" option as:

```
  ls -ld aaa
```

which is equivalent to the *full* pathname command (in our illustrative example):

```
  ls -ld /h1/bbb/aaa
```

then the *status* of directory "/h1/bbb/aaa" *itself* will be given (in this case, a "long listing"), and *no* information about its contents (ordinary files "junk" and "trash") will be printed.

Selective Listings We may also operate on selected ordinary *files* by listing their filenames at the end of the "ls" command (separate filenames by "spaces"), as:

```
  ls -l stocks aaa/junk
```

will give long listing information about the *ordinary* files, with full pathnames "/h1/bbb/stocks" and "/h1/bbb/aaa/junk," as "branching" from the current working directory is again implied.

If you request information about a non-existent file or directory, UNIX will cryptically notify you that it does not exist. If a *directory* name is included in the list, the name will be "echoed" by UNIX, and then its *contents* will also be listed. If the "-d" option is also used, it will have *no* effect upon the listing of *ordinary files*, but only directory status information will be listed (rather than listing the directory's contents).

As a final note here, it should be pointed out that many more LIST options are available to the interested user; see the "ls" command page in the UNIX User's Manual.

3.2 File Protection

Changing File Protection (chmod) For files you own, you may wish to deny file "read" or "execute" access, or more probably "write" permission, to other users. This can be accomplished by the CHANGE MODE ("chmod") command, the general format of which, is:

```
  chmod mode file1 file2 file3 . . . . .
```

where files named have their mode changed (permissions to read, write and execute) according to the "mode" argument which may be "octal" or "symbolic" in nature. Note that filenames listed are assumed to be branches from the "current" working directory, unless you use *full* pathnames.

Read, write and execute permissions are allocated to three classes of users: "owner" of file, "group members " (the formation of groups is discussed in Section 3.3), and "others" (all other UNIX users). The use of the "ls -l" command, described previously, will indicate the current permissions associated with a file.

Octal Mode The "octal form" of mode designation is a string of three octal digits (range 0 - 7). Each of the three digits assigns permissions as follows:

0: can *not* read, write or execute
1: execute only
2: write only
3: execute and write only
4: read only
5: execute and read only
6: read and write only
7: read, write and execute

Note that "reading a file" means you can access the contents of the file; to bring a file into the editor, you need this "read" capability. "Write" permission allows you to modify the contents of a file, which is required in an editor write, for example. "Execute" permission refers to files containing executable code, such as a compiled "C-program," or a "Shell Procedure" containing executable UNIX commands.

In the three digit number, the first digit refers to "owner" (so-called, "user") permissions, the second digit refers to "group" permissions, and the third refers to the permissions of all "others." Therefore

```
chmod 740 junk
```

indicates file "junk" can be read, written onto and executed by the "user/owner," can be read by "group" members, and can not be accessed at all by "others." Note that a *leading* fourth digit may be added to the "octal mode" — again, it is of little interest to the average user, but you may refer to the UNIX User's Manual for more information.

Symbolic Mode The "mode field" may also be entered symbolically, where the first character may be chosen from the list:

u	user/owner
g	members of owner's group
o	all others
a	everybody

The second character may any of the following:

```
  =   to list a new set of permissions
  -   to delete permissions
  +   to add to existing permissions
```

The remaining "mode symbols" may be chosen from the list:

r	read
w	write
x	execute

Examples are:

```
  chmod a+w junk
```

which adds the "write" permission to all users of file "junk" (in addition to existing permissions), and:

```
  chmod u=rw junk
```

will assign "write" and "read" permissions to the "user/owner" and will leave permissions for "group" and "others" unaltered.

Permissions When a File is Created When a file is created using the "ed" command, the "owner" is assigned "read" and "write" permission, while "group" and "other" users have "read" permission only (usually the default --- see the "umask" command in Section 3.15 for more details), that is "mode 644"; this is the usual protection you'll want to guard against unauthorized editor modifications. Any time thereafter, the "owner" of the file *only* may use the "chmod" command to alter file permissions. It might be pointed out that a large file could have "write" permission denied its owner -- a protection against careless overwriting or removal.

Directory Permissions and Changing Them A final comment is that "chmod" may also be applied to *directories* in the form shown above, except "read," "write" and "execute" must be reinterpreted. "Read" permission implies that you have the right to list the *names* of files contained in the directory. "Write" permission allows you to *create* or *destroy* files within the directory. "Execute" permission is equivalent to the right to "search" the directory. i.e., being able to enter it. Directory permissions are *not* as straight-forward as file permissions. If a directory has "write," but *not* "execute" (search), permission, you *cannot* remove a file because you can't gain access to the directory. If you wish to create/destroy files within a directory, make sure a "7" *directory* mode is assigned to you; if you wish to read/edit a file, a "5" *directory* mode is required. The usual default given when you create a directory is octal mode "755," which is the permission status most users desire.

File/Directory Permission Anomolies As a final example of file/directory permissions, consider a typical "tree-structure":

```
          / (system "root" directory - "mode 5")
                           |
          h2 ("disk file" directory - "mode 5")
                           |
           kew ("login" directory - "mode 7")
               |                          |
              aaa                        bbb
      (directory - "mode 0")   (directory - "mode 5")
               |                          |
              ccc                        ddd
      (directory - "mode 7")   (directory - "mode 7")
        |               |          |              |
       www             xxx        yyy            zzz
        (ordinary files)           (ordinary files)
```

Now lets attempt to list the names of the ordinary files shown. A command:

```
ls /h2/kew/bbb/ddd
```

will cause the names of files "yyy" and "zzz" to be listed, as all directories in the "tree branch" have permission mode "5" or "7" associated with them. Therefore, UNIX has the capability of *entering* the "root" directory, and *reading* the names of files and directories contained within it, including disk file directory "h2." Now UNIX is allowed to *enter* directory "h2" and *read* the names of "logins" contained within it, including directory "kew." UNIX can always proceed to the next level of directories possessing "x" permission (the right to "enter" it), and then list the contents of that directory, if it possesses "r" permission (the right to "read" those names). In our case, we ripple down the tree until the last directory in the path (i.e., "ddd") is entered.

If, however, "entrance" to a directory is blocked ("x" permission denied), and/or the contents of a directory cannot be "read" ("r" permission denied), *anywhere in the path*, a command using that pathname will fail. For example:

```
ls /h2/kew/aaa/ccc
```

will *not* succeed. Although the directory "ccc" has "mode 7" permission, certainly allowing the names of files within it to be "read," UNIX *cannot* penetrate directory "aaa," which will not allow "entry" nor allow its contents to be "read." Therefore, again let me advise you to use directory mode "755," unless you have a specific need to do otherwise (such as special "group permissions" discussed in Section 3.3, or the "rje" directory discussed in Section 3.11).

3.3 The Formation and Use of Groups

It is possible that some files or directories are of special importance to a select *group* of users (probably people working on the same project). For example, you may wish to assign "write" permission to a file being edited by several members of a *group*, but ensure that all other users can only "read" the contents of that file. In these cases, the *owner* of the file may assign permissions to "group" members different from those of all "other" users (via the "chmod" command), but the question to be resolved is: "How are administrative *groups* formed?".

Every user is the *owner* of *one group*, associated with his/her *login* ID, and *only* the owner has the right to alter membership in that group.

Adding Members to a Group (addgrp) To *add* members to *your* group, issue the command:

```
  addgrp group_name login1 login2 ....
```

where "group name" is the *login* name of the *group owner*, and "login1," "login2," etc. are the names of other users to be *added* to the group. For example, if my login is "kew," and I wish to add "wek" and "aaa" (two logins on the *same* UNIX system) to *my* group, the command

```
  addgrp kew wek aaa
```

will accomplish the task.

Deleting Members from a Group (delgrp) To *remove* members from *your* group, use the command:

```
  delgrp group_name login1 login2 ....
```

where, again, "group name" is the *login* name of the *group owner*. The command:

```
  delgrp kew aaa
```

would delete the member "aaa" from the group *owned* by "kew".

Remember, *only* "kew" can add or remove members of the group owned by "kew". Every file has an *owner* and a *group* identification associated with the *login* of the *creator*, regardless of the "current" working directory where creation took place, or the ultimate depository of that file. Only that owner may change permissions associated with that file, or alter group membership referring to that file.

Utilizing Group Permissions To avail yourself of special *group* permissions associated with a file, you must first be included in the *owner's group*, and you must make UNIX aware of the fact that you wish to work with a specific owner's files and be subject to special *group* permissions. By issuing the command:

```
newgrp group_name
```

you have accomplished this feat. If you are *not* a *member* of "group name", UNIX will reply with "Sorry," and disallow *your* use of the special *group* permissions associated with "group_name". This command issued *without* argument is equivalent to changing to that group associated with *your* login (i.e., the group that you own).

Special groups (other than those just described above) may be arranged by contacting your local UNIX administrator. Also, you may wish to consider *linking* (which will be discussed in Section 3.5) as an alternative to "group permissions."

3.4 Removing Files and Directories

Simple File Removal (rm) To REMOVE an existing file (or files), we may use the "rm" command, such as:

```
rm filenames
```

Leave a space between "rm" and each filename. Be careful here, as files listed are removed -- permanently!!! "Filenames" are assumed to be "ordinary files" in your *current* working directory, unless *full* pathnames are indicated.

If you attempt to remove an "unwritable" file, UNIX will notify you by printing the "permission mode" associated with that file, and expects a response from you. Such a message will be printed only if the file you've attempted to remove does not possess "write" permission. If you respond to UNIX with a line beginning with a "y", the file *will* be deleted; for any other response (preferably just a CR), the file remains intact.

Permissions to Remove a File This may seem strange, but only the *directory* containing the file *must* have "write" permission, for that file to be removed --- be careful. Just as strange is the fact that if you assign a "mode 5" *directory* permission (the right to "read" the contents of that directory and "enter/search" it), and a *file* within it has "mode 6 or 7" (the right to "read," "write" and, possibly, "execute" it), you *can edit* the file and effectively destroy it. Thus, to fully protect a file from destruction or modification, the most *lenient* permission which allows "reading" is to assign a "mode 4 or 5" permission to the file, and a "mode 5" permission to the directory containing that file.

Forced Removals (rm -f) An alternative is to use the command:

```
rm -f filenames
```

where the "-f" option *forces* the removal of a file (in a "writable directory"), regardless of the permissions associated with that file, and will inhibit the "mode warning."

Removals of Directory Contents (rm -r) The "rm" command with a "-r" option, i.e.,

```
  rm -r directory_name
```

will remove *all* files in the specified directory, including *all* subdirectories and their contents, recursively. Finally, "directory name" *itself* will be removed, if it is not your "current" working directory. It is important to note that "directory_name" itself and any subdirectory you wish to affect must have a "mode 7" permission associated with it. If any *ordinary* file to be removed is "unwritable," you will receive a "mode warning," but it can still be removed as described above.

Fortunately, to remove *all* files in your "current" directory, it must be *explicitly* named via a *full* pathname (or its abbreviation ".", explained in Section 3.6), and the command

```
  rm -r
```

without directory name argument, will do *nothing*. Probably, you would never want to issue this command *on purpose* anyway.

Deleting Files Interactively (rm -ir) Another useful command is DELETE INTERACTIVELY ("rm" command with a "-ir" option), of the form:

```
  rm -ir directory_name
```

In response, UNIX will *sequentially* list all files/subdirectory names in "directory_name", expecting a response from you.

If the UNIX printout is *prefixed* by the word "directory," such as:

```
  directory some_directory_name:
```

UNIX wishes to *enter* that (sub)directory for the removal process. A "y" response *allows* UNIX to *enter*, while a response beginning with any character other than "y" (preferably a simple "carriage return") will cause that directory to be *skipped* in the removal process. If you attempt to enter a (sub)directory without possessing a "mode 7" permission, you will not succeed in removing any files and the "interactive deletion" process will abort prematurely.

A listed *file* (or file partial pathname), *without* a *prefix*, indicates UNIX is awaiting your instructions concerning the removal of the named file. Again, a "y" response will cause *removal* of the named file, while a response beginning with any character other than "y" will leave the file *intact*. If the file in question is "non-writable," a "mode warning" will *not* be issued.

A (sub)directory may be subject to the same removal process, *if* all files/subdirectories within it have been removed previously. This "interactive deletion" process will sequentially penetrate *all* levels of subdirectories.

Hitting the "DEL(ETE)" or "BREAK" keys will cause you to *exit* the REMOVE command.

If you wish to "interactively delete" files in your "current" working directory, you must *explicitly* indicate its full pathname (or its abbreviation "."); this is a useful tool to periodically "clean out" your login directory of unwanted files. The command:

```
rm -ir
```

without directory name argument, will do *nothing*.

Permission Modes Revisited It should be pointed out one more time that a file may *not* be removed by any of the above techniques, unless the *parent* directory has "mode 7" permission assigned to the *remover* (whether he or she is the owner, group member or other user of that directory). By denying "x" permission, you cannot enter the directory for the removal process, and by denying "r" permission, you cannot read the names of files you are trying to remove. Also, *only* the permission of the *parent* directory affects the removal process, and the permission modes of all other directories, right up to the "root," need only be mode "5."

Careless File Removal Should you inadvertently destroy a "valuable" file, you still have a chance to retrieve an *older* version, saved on a *tape* copy of the disk files made at the end of each business day, by contacting the UNIX administrator of your particular system.

3.5 Copying, Renaming and Linking Files

Copying a File (cp) The COPY ("cp") command makes a duplicate copy of a file, leaving the original intact, i.e.,

```
cp file1 file2
```

will form a copy of "file1" under filename "file2". Beware that if you COPY a file to another one that already exists, the original contents are overwritten, if the the "target" file (in our example, "file2") is *writable*.

In the last example, we have copied ordinary files within the "current" working directory. It is possible to enhance this capability, by using full pathnames, such as:

```
cp /h2/txx/one /h1/kew/first
```

In this case, file "one" in directory "/h2/txx" will be copied into file "first" in directory "/h1/kew".

A restriction is that the files and directories involved must possess the proper permissions. Specifically, you must be able to *enter* directory "/h2/txx" and be able to *read* the names of files contained within it (directory mode "5" or "7" required), while the file "one" must be *readable* (at least mode "4"). Similarly, you must have mode "7" permission associated with directory "/h1/kew", such that new copy "first" may be *created*: if "first" already exists, directory "/h1/kew" must possess mode "5" or "7" permission to allow *entry*, and file "first" must be *writable*. The discussion of permission modes in the REMOVE command section was included to provide a *warning*; points raised here and in the LIST command section were included simply for your information. If the "cp" or "ls" commands fail due to improper permission modes, you will *not* be harmed, but you can spend a lot of time "cursing at UNIX," if you are unaware of what the problem is. Also note that directories involved in the "cp" command must reside in the same UNIX system --- disk file systems "h1" and "h2" reside on the "UNIX-H system"; note that the "rm" and "mv" (to follow) are similarly restricted to the UNIX system containing the "current" working directory.

Before we leave this example, there is one more interesting point to note. When a COPY command is issued, *whoever* issues the command is the *owner* of the duplicated file. In our example, if login "txx" issues the command, the file "/h1/kew/first" is *owned* by "txx". The duplicate file "first" (if it didn't previously exist) will be *created* with the same permissions as the original file "one", probably "mode 644." This means that login "kew", in whose directory file "first" resides, can only *read* its contents (as "txx" is still the *owner* of that file), and "kew" cannot even change its permission mode. Therefore, if "kew" wishes to edit the contents of file "first", a *personal* copy (say "kew_first") must be created; luckily, "first" may be *removed* by "kew", as he/she is the owner of the *parent directory* (the login directory "kew" itself).

To avoid any of these strange permission or ownership problems, it is best to create (see Section 3.8 and the "mkdir" command) a *remote job entry* subdirectory for your login, called "rje". Your login directory should possess permission mode "755" and the "rje" directory should be mode "777." Now anyone can send you copies, as long as they are directed to your "rje" subdirectory; you, in turn, can make a second copy in any of your other (sub)directories and be the owner of this new duplicate. This "rje" subdirectory will also be used when sending files to another UNIX system (see the "usend/nusend" command in Section 3.12).

Copying Many Files (cp) Another variation can be demonstrated by the command:

```
cp file_name directory_name
```

which copies "file_name" in the "current" working directory to a target file with pathname "directory_name/file_name". The argument "file_name" may be replaced by a sequence of names separated by "spaces."

You may *not* copy "directories" using the "cp" command. See the "cpio" command in Section 3.13, if you are interested in doing this.

Renaming Files (mv) The MOVE ("mv") command should perhaps be called "RENAME THE FILE." The command:

```
  mv file1 file2
```

changes the name of file "file1" to "file2". As usual, unless you specify *full* pathnames, the files listed are assumed to be "branches" of the "current" working directory.

It should be noted here that if you attempt to "move to" a file that already exists, the original contents of that file will be overwritten (lost forever!!), provided the file is "writable": in our sample command, "file2" is in danger. If that file is "unwritable," UNIX will print its "permission mode" (similar to the REMOVE command) and await a response from you; if the first character you type is "y", the "move" *will* be executed, while any other response will neglect the MOVE command.

Moving Many Files to a New Directory (mv) A command such as:

```
  mv file1 file2 file3 .... target_directory
```

will also work, moving the named files to a new "target_directory" (which must be on the same UNIX system).

Changing Directory Names (mv) A command:

```
  mv subdirectory_1 subdirectory_2
```

will change the name of "subdirectory_1" to "subdirectory_2". noting both subdirectories must appear in the same "parent" directory.

Linking Files (ln) The same *ordinary* file may appear in several directories under possibly different names. This feature is called "linking." If changes are made to the file, *all* users linked to that file see the same modifications; it is, in fact, the *same* file. To create a "link" to an existing file, invoke the command:

```
  ln existing_file target
```

where "existing file" (an *ordinary* file) may be a simple filename, if it is located in your current working directory, or a *full* path name, if not. The "existing file" will now appear in the "target" *directory* with name "existing file" (or the *last* component of "existing file"). If you wish to *rename* the file in the "target" directory, then "target" should be a *full* pathname down to the "filename level." The commands:

```
  ln /h1/kew/junk /h1/knn
  ln /h1/kew/junk /h1/too/garbage
```

will produce a file with *three* "links"; "/h1/kew/junk", "/h1/knn/junk", and "/h1/too/garbage" are the *same* file with access from *three* directories.

Note that you may *not* link files in different disk file systems, such as:

```
ln /h1/kew/junk /h2/bbb
```

is an *illegal* command: "h1" and "h2" are *different* disk files on the *same* UNIX-H system.

You may *not* link directories. Also, if *one* user linked to a file invokes the "rm" command, *only* that user's link to the file is removed (the file is destroyed if it has *only one* link associated with it). The number of "links" associated with a file is indicated by the number *before* the *file owner's name* listed via the "ls -l" command. Finally, note that you may link to another user's file *without* his or her consent, provided the directory containing that file has "mode 7" permission for you.

You may also link many files at once to a "target_directory", via a command of the form:

```
ln file1 file2 file3 .... target_directory
```

Ownership/Permission Anomalies The problems associated with ownership of a "copied file" have already been discussed. To recap, the "owner" of a file *copy* is the *issuer* of the COPY command. If a new file is *created* for the copy, the permissions will be the *same* as the *original* file; if one copies onto an *existing* file (with "write" permission), only the contents, but *not* the permissions, of that overwritten file change.

As links to a file are formed, the owner of *all* linked copies is the same as the owner of the *original* file. If the linked file has standard permission mode "644" associated with it, then *only* the *owner* may modify the file contents, which is reflected in all linked copies. In many cases, however, the link was formed such that *all* linked users could make file modifications; if this is the case, the *owner* must elevate the permission assigned to "others" (or possibly "group members") to mode "6." Obviously, if protection of the file is important, it must be protected at the *parent* directory level, by assigning a mode "0" directory permission to "others" (and possibly "group members").

It should be obvious by now that the whole business of security and file/directory permissions is a giant "pain in the ___." If you set modes up "logically," probably you won't get the protection you want, or you'll fail when you try to execute certain commands. Documentation on these points is very poor, and most of the comments pointed out in this memo were due to thorough investigations of why certain commands failed. You may run into these problems one at a time, and, possibly, with the warnings presented here, you may have a "glimmer of hope" in determining what went wrong (or you can just "throw up your hands" in exasperation, assign "r", "w", and "x" permissions to all you files/directories, and pray that all your files survive the "abuses of the outside world").

Getting a little more serious, we still must determine the ownership of "moved" files. If files are "moved" within the *same* "disk file system," ownership is *not* changed (although the *issuer* of the MOVE command may *not* be the owner of the file moved) and *all* links to that file are retained. However, if you move a file to *another* "disk file" (on the same UNIX system, of course), ownership belongs to *whoever* issued the MOVE command. In the latter case, a "copy" is actually made and the original file is "destroyed" (or has its link removed).

Changing the Owner of a File (chown) To solve some of these problems, you, as the *owner* of a "copied/linked/moved" file may desire to change ownership of that file, if it makes sense to do so. Probably, if you copy or move a file to someone else's directory, it would be "nice" to have the file, and directory it resides in, have the *same* owner. The *owner* of that file may do this by issuing the CHANGE OWNER command ("chown"), as follows:

```
  chown new_owner filenames
```

where obviously the *current owner* of "filenames" relinquishes ownership to a new login ("new_owner"). Be careful, as once ownership has "changed hands," *only* "new_owner" may issue a "chown" or "chmod" command for that file: you have given up your ownership *permanently* and *irrevocably* (UNIX is not sentimental about "past" owners).

As a final note here, a change of ownership of a "linked" file, changes the owner of *all* linked copies, regardless of the directories they reside in.

Copy and Convert Command (dd) Most user probably don't require any alternatives to the COPY command, but the "convert and copy" command, cleverly named "dd," may be useful when transferring files to tapes or other raw physical I/O devices. The basic format is:

```
  dd if=input_file of=output_file options
```

Without options, "dd" will create/overwrite "output_file" with a copy of "input_file"; designate either full or partial pathnames.

The interested reader should look at the "dd" manual page for specialized options available. Examples include:

conv=lcase	Converts all alphabetics to lower case
conv=ucase	Converts all alphabetics to upper case
conv=swab	Swap every pair of bytes.

3.6 Printing File Contents

Concatenating Files (cat) We can print the contents of files via a direct UNIX command without entering the editor. The CONCATENATE ("cat") command will print the contents of one or more files onto the terminal. Consider the command:

```
  cat file1 file2
```

which will *concatenate* (join together) the files "file1" and "file2", in the order listed (there may be more than two files listed), and print them on the terminal. Remember to leave a "blank space" between "cat" and each file name. As always, files *not* listed with *full* pathnames are assumed to be "branches" of the "current" working directory. Also, do *not* "cat" *directories*, as garbage will be printed.

Paginated Printing (pr) The PRINT ("pr") command has the following general form:

```
pr options file_names
```

and acts as the "cat" command except it prints the contents of each of the named files in a formatted manner. It provides a heading on each page (including date, time, page number and filename), followed by 56 lines of text, when no "options" are present.

"Options" is a "space-separated" list, including those listed below, and many more to be found in the UNIX User's Manual.

+k	Begin printing with page "k" (default is page 1)
-k	Produce "k-column" output (default is single column, i.e., normal text printing)
-w#	Set width of the page to "#" characters, rather than the default 72
-l#	Set the length of the page to "#" lines, rather than the default 66 (i.e., 56 lines of text, plus a header and trailer)
-h xxx	Use "xxx" as the header to be printed on each page, instead of the filename (noting you must enclose the header in "quotes," if it contains more than one word)
-p	Output will pause *before* beginning *each* output page, a *bell* will sound at the terminal, and a CARRIAGE RETURN will begin printing again (this is very useful for printouts on the CRT, if the page length is adjusted properly)
-t	Print neither the five-line header or trailer normally supplied
-n#	Provides line numbering. Numbers start at "1" for each named file; line numbers are "#" digits long, with default value "5".

Stopping the Printing Process It should also be noted that printing due to an issued "cat" or "pr" command may be *terminated* by hitting the "DEL(ETE)" or "BREAK" key. Also, as described previously, "CONTROL-S" will alternately *stop* and *start* the printing process, which may be a lifesaver as text is scrolling off a CRT screen.

Offline Printing (opr) There is also an OFFLINE PRINT ("opr") command which produces a output on a lineprinter or Xerox laser printer in the computer center. The general format of this command is:

```
  opr options filenames
```

where "filenames" is the list of files (separated by "spaces") you wish to print. Useful "options" (again separated by "spaces") are:

-b #	This option will direct output to *bin* number "#," rather than the default "bin" read from your password file (see Section 3.15, if you don't know what your default "bin" is).
-d dest	This option directs output to a remote station "dest": the default is the local computer center.
-t type	This option will direct output to a particular "type" of printer. Valid entries for *type* are "xr" for the 9700 Xerox printer, or "pr" (default) for standard line printers.
-f form	For Xerox 9700 printers, you may specify whether you want copies produced without or with holes. The argument *form* may be designated "nohole" or "hole" (the latter is the default).
-p mode	For Xerox 9700 printers, you may specify the "mode" of output. Possible values for the argument *mode* are "land" (for computer output or *landscape* 11x8.5 orientation, which is the default), "por" (for letter or 8.5x11 *portrait* orientation), or "2on1" (for two landscape pages on one side).
-u name	Places "name" (1 to 8 characters), instead of your login name (default), on the banner page of the printout.
-j job	Places "job" (1 to 8 characters), instead of the default jobname internally generated by UNIX, on the banner page of the printout.
-n	This option will cause *line numbers* to be printed. These numbers begin at "1" for each file argument.
-c#	Causes "#" copies of the output to be produced. Default is "1" and maximum is "99."

For more options, see the manual page for the "opr" command. Also it should be noted that the "form," "dest," "type" and "bin" may be preselected when you log in; see the "user's profile" description in Section 3.15.

Once an "opr" job is queued, UNIX will respond with an OPR JOB NUMBER: at this point, the named files may be modified without affecting the printing.

Offline Printer Status The OPR command utilizes RJE ("remote job entry") hardware to connect your UNIX system to IBM hardware, which actually orchestrates the printing. If this is not operational, your print jobs may be lost, or at least delayed for some time. To obtain the current status of the RJE hardware, simply issue the command:

```
  rjestat
```

3.7 Special Characters Associated with Filenames

When specifying filenames as command arguments, such as "ed", "ls", "cat", "pr", "opr", "ln", "mv", "cp", "rm" and more to come, there is a new group of "metacharacters" to help (and maybe confuse) you. They are the "asterisk (*)," "question mark (?)," "square brackets ([])" and "exclamation point (!)" ---- and of course the "#" (erase) and "@" (kill line) characters still work. Their characteristics are somewhat or drastically different than their "editor" counterparts --- be wary.

For illustrative purposes, assume our "current" working directory contains the following files:

```
  .profile
  junk
  junky
  bad_junk
  jinx
```

QUESTION MARK Metacharacter (?) The "?" may be used to match any *single* character (except the ".", if it is the *first* character of a filename). The command

```
  rm ?profile
```

will *not* match file ".profile" or any other file in our directory, and UNIX will so indicate. However

```
  rm j?n?
```

would successfully REMOVE files "jinx" and "junk".

SQUARE BRACKETS Metacharacter ([...]) A string of characters enclosed in "square brackets" will match any *single* character in the string. You may indicate a lexical range with a "minus sign," such as "4-9", which is expanded to "456789". For example, the command

```
  cat j[a-z]n[kx]
```

would concatenate and print files "jinx" and "junk". Note that this metacharacter grouping will again *not* match a ".", if it is the *first* character in a filename.

EXCLAMATION POINT Metacharacter (!) The "square brackets" has an *negation* character, similar to the "leading ^" in the analogous RE: in this case, it is the "exclamation point" (!). For example, the sequence

```
  [!hnm]
```

matches any *single* character *except* "h", "n" or "m". Similarly, the command:

```
  cat j[!k-z]n[xk]
```

would print the contents of file "jinx" only; [!k-z] represents any single character *not* in the lexically contiguous range "k" through "z" inclusive.

ASTERISK Metacharacter (*) The last metacharacter is the "asterisk," which may be translated to mean "anything at all." A leading "*" will match any or no prefix (except the ".", if it is the *first* character in a filename). Thus

```
  pr *junk
```

would cause a formatted printing of files "bad_junk" and "junk".

A trailing "*" will match any or no suffix, i.e.,

```
  pr junk*
```

would print "junk" and "junky". Note that

```
  pr junk??*
```

matches a filename of *six* or more characters, starting with the string "junk", and this matches *no* files in our directory. The important point to note here is that "?" matches a *single* character, while "*" matches *zero* or more characters.

A string enclosed by two "*'s" will act upon all files containing the string, i.e.,

```
  pr *junk*
```

would print "junk", "junky", and "bad_junk". Again

```
  pr *ro*
```

would not yield a match with file ".profile", as the first "*" does not match the "." in a filename, when it is the first character.

Finally, "*" alone matches any file, except those beginning with a "."; hence, a very dangerous command in UNIX is

```
  rm *
```

which could wipe out all files except ".profile", in our illustrative directory.

Note that "metacharacters" can be used in the last directory or file in a *pathname*; all the following pathnames would match "/h2/abc/help" (and possibly other files in directory /h2/abc):

```
  /h2/abc/h*
  /h2/abc/[ho]?lp
  /h2/abc/*e*
```

A pathname such as "/h*/a??/help", while not illegal, is confusing; it is best to use "metacharacters" *only* in the *last* file or directory in the path.

3.8 Other Directory Manipulations

Changing Directories (cd) If you want to work on someone else's files rather than your own, you can change directories using the CHANGE DIRECTORY command "cd", for example

```
  cd /h2/knn
```

will change the directory you are currently in to login name "knn" in disk file "h2". Hence, your "current" (or "working") directory need *not* be your "login" directory. Once you have changed to a foreign directory, you can only manipulate files to the extent the owner of that directory allows permissions; obviously, you must have "x" permission assigned to you, or you will be unable to enter the new directory in the first place. Also, you may only change directories within the same machine; for example, if your "login" is on UNIX-H, you have "cd" access to file systems "h1" and "h2" only. As usual, if you do *not* specify a *full* pathname, the "cd" command will seek a "branch" from the "current" working directory.

Also note that the command "cd" with no argument will automatically return you to your "home" or "login" directory.

Current and Parent Directory Abbreviation You may go up one level in the tree structure by typing

```
  cd ..
```

where ".." is shorthand for "parent of whatever directory you are currently in." If you are in your subdirectory "nose", i.e., "/h1/your_login_name/nose", the above command will leave you in the "parent of nose," i.e., "/h1/your_login_name". For completeness, "." is a shorthand name for the directory you are in (your *current working* directory), as mentioned previously. In fact, the command "ls -a" (which prints the names of all files/subdirectories contained within your working directory, including those beginning with ".") will always list as its first two names, your current working directory "." and your parent directory "..".

Determining Your Working Directory (pwd) By this time, you're probably so damn confused, that you don't know what directory you're in. UNIX saves you with the PRINT WORKING DIRECTORY command.

i.e.,

```
  pwd
```

will print the full pathname of whatever directory you're currently working in.

Creating New Directories (mkdir) It is often convenient to arrange one's files, so that all files related to one special area of work are grouped together. We can accomplish this with the MAKE DIRECTORY command "mkdir".

Assume we wish to write a "book of two chapters," we could make a new directory:

```
  mkdir book
```

which forms a *directory* of pathname "/h1/your_login_name/book" (again, I am assuming throughout that you are working in your "login" directory, which is located on the "h1" file system). Now you can get into this directory by the command:

```
  cd book
```

Note that a "full pathname" is not necessary here, as a "branch" starting from your "current directory" is implied. Now you can form files "chap_1" and "chap_2" via the editor, and you can locate any chapter, say "2," by using the full pathname "/h1/your_login_name/book/chap_2" . You may extend this directory creation procedure to any level desired (within reason).

Removing Empty Directories (rmdir) You may remove an *empty* directory using the REMOVE DIRECTORY command "rmdir" (possibly by referring to the directory by a "full pathname"); if it is *not* empty, you must delete all files in the directory first. For the "book" directory above, the following sequence will eliminate the directory:

```
  cd book            [changes from "login" to "book" directory]
  rm chap_1 chap_2   [removes all chapters from "book" directory]
  cd                 [return to "login" directory]
  rmdir book         [removes empty directory "book"]
```

Obviously, the "rm -r book" command, as previously described, can be used to accomplish the same feat (and more easily).

Locating Specific Files/Directories (find) FIND will recursively descend a named directory structure, and for each file/directory encountered, it will evaluate one or more expressions. With several more pages of descriptions and examples, we will hopefully find out what the hell the above statement means. First, consider the general format of the FIND command:

```
  find directory_list expression_list
```

The argument "directory_list" specified is a space-separated list of *full* or *partial* pathnames of directories to be investigated. As you will generally be interested in only *one* directory, the discussions to follow will assume this fact; once you have mastered single directory searches, extrapolation to the multiple case is really intuitively obvious. The directory specified is just a starting point, as the FIND command will recursively penetrate all levels of subdirectories. For *each* file/directory encountered in this search, the "expression_list" will be executed.

The argument "expression_list" is composed of a series of *primaries*, separated by spaces. A *primary* is a boolean expression, which may be evaluated to a "true" or "false" condition, and, in some cases, performs an action. For each file/directory found in this recursive search, the first primary is evaluated; if the evaluation indicates a "true" condition, the second primary is evaluated. This process is repeated until all primaries have been evaluated, or until a "false" condition is encountered. In either case, the entire sequence of events is repeated for the next file/directory found in our search.

All that remains is a list of commonly used *primaries*, and then we'll be ready for some illustrative examples. For the list delineated below, two abbreviations are used:

[1] The string *LC* will be used to represent the *last component* in a pathname, regardless of whether it is a file or directory.

[2] The argument "#" can be a *positive decimal integer* "N," or "+N" (which means more than "N"), or "-N" (which means less than "N").

Those common *primaries* are:

-name *NAME*	Primary is "true," if *NAME* matches the current *LC*. Normal filename metacharacters ("[", "?", "*" and "!") will work, if *NAME* is enclosed in *single quotes*.
-perm *OCT*	Primary is "true," if *OCT* matches the *permissions* associated with the *LC* exactly. *OCT* is a three or four digit, octal number, in the style of the "chmod" command (see Section 3.2).
-type d	Primary is "true," if *LC* is a *directory*.
-type f	Primary is "true," if *LC* is an *ordinary file*.
-links #	Primary is "true," if *LC* has "#" *links* associated with it.
-user *NAME*	Primary is "true," if *NAME* is the *owner* of the *LC*.
-atime #	Primary is "true," if the *LC* has been last *accessed* "#" days ago.

-mtime #	Primary is "true," if the *LC* has been last *modified* "#" days ago.
-newer *FILE*	Primary is "true." if the *LC* has been modified *more* recently than the argument *FILE*, which should be a designated with a *full* pathname.
-print	This primary is *always* "true." It cause the full pathname of the current LC to be *printed*.
-exec *CMD*	This primary executes the designated command *CMD*, and is "true," if that command is successfully executed. Any UNIX command is permissible, with the following restrictions: • The command must be terminated by an *escaped semi-colon*, i.e., the couplet "\;" • A command argument "{}" is replaced by the current pathname. • Be wary of using characters which may have special meaning to UNIX; their effect can often be negated by preceding them with a "backslash" or enclosing groups of characters with "single quotes."
-ok *CMD*	Primary is similar to "-exec," except that the generated command line is first printed out, followed by a "question mark." This command is executed only if the user responds by typing a "y" followed by a "carriage return"; a "carriage return" *alone* will cause the command to be skipped.
!	The "!" primary is used to *negate* the meaning of whatever primary follows it.
-o	The "-o" operator provides *logical oring* of primaries. Any grouping of "ored" primaries should be enclosed by "\(" and "\)" to be interpreted properly.

As our first example, assume we wish to determine the full path name of all *ordinary* files within the directory "/a1/junk" (or its subdirectories) that have names "trash" or "garbage"; try the following FIND Command:

```
find /a1/junk - type f \( -name trash -o -name garbage \) -print
```

Note that the use of "\(" and "\)" causes the *oring* of names "trash" and "garbage" to be treated as a single Boolean expression.

Next assume you wish to remove all files in the current directory (and all of its subdirectories) that *end* in ".o"; try the command:

```
find . -name '*.o' -exec rm {} \;
```

Note that the "{}" argument causes any files matching the "-name" primary to be removed: the names we are trying to match make use of metacharacters, and should be enclosed in "single quotes."

Our next example allows us to print the full path name of all files in the current directory (and all of its subdirectories) that are *not* owned by login "smart" and have *not* been accessed in the last week, i.e.,

```
  find . ! -user smart -atime -7 -print
```

Note that the "! -user" group is *true* if the *owner* of the *LC* is *not* "smart" via the *negation* primary; the argument "-atime -7" is *true* if the *LC* has been accessed last *more* than seven days ago.

Our last example attempts to produce an "owner listing" of the files in *some* directories in our UNIX system, whose names *begin* with "x" and *end* in "y":

```
  find / -type d -name 'x*y' -ok ls -o {} \;
```

Note the "-ok" primary is used rather than "-exec." as only *some* "long listings" are desired. If a match is found to the desired directory name, you may interactively cause the LIST to be produced, as FIND will print out the "list" command generated, and wait for a response from you; a "y" response will cause the listing to be produced, while a "carriage return" will disallow the listing, but will allow the search for more directories to continue.

Although the FIND command may look a bit ominous at first, it should not be passed up to quickly, as it is a very *powerful* and *useful* tool. Set up a practice directory structure, load it with some files, make up FIND commands using many different primitives, view the results, and soon you'll be an expert.

Free Disk Space (df) On congested UNIX systems (especially those dedicated to specific projects), it may be impossible to do any work because there is no *free space* on disk file systems. The following command will list the number of "free blocks" (a block is 512 characters) available in the named disk file systems (such as "/a1," "/a2," "/usr," "tmp," etc.):

```
  df -f disk_file_list
```

If the "disk_file_list" argument is omitted, information for all disk file systems will be produced by default. If you also wish to know the total number of *allocated* blocks for a particular disk file system, add a "-t" option to the above command.

Disk Usage (du) To summarize disk usage within particular directories, the following command can be used:

```
  du -options directory_names
```

The argument "directory-names" is a space separated list of directories you wish to be investigated for disk usage; if this argument is omitted, the *current* directory is assumed by default. This command gives a count of used blocks in the named directory, and recursively gives identical data for all *subdirectories* contained therein. The possible *options* are single letters (which may be concatenated), i.e.,

s	Causes only the *grand total* for each of the specified directory names to be outputted; this includes disk usage found within subdirectories.

a	Causes an entry to be produced for each *file* encountered in the recursive directory scan.

3.9 Redirection of Inputs and Outputs

Most of the commands we have seen so far produce output on the terminal. Some commands, like the editor, also take their input from the terminal. In UNIX, it is possible to replace the terminal with a file, for either (or both) input and output.

Output Redirection (>) If we issue the command:

```
ls
```

a list of filenames is printed on the "standard output" (the terminal) by default. This output may be redirected to a file, say "names" in this example, with the following command:

```
ls >names
```

This command creates file "names" if it does not already exist, or will overwrite it, if it does; the output of the "ls" command is delivered to file "names" *only* —— *no* output is printed on the terminal. The symbol ">" is used throughout UNIX to mean "put the output on the following file, rather than on the terminal."

Combining Files Another example is to combine several files into one file, i.e.,

```
cat file1 file2 file3 >single
```

will concatenate files "file1", "file2" and "file3", and, instead of printing them on the terminal, will write them onto the file "single".

Creating Empty Files Note that a command of the form:

```
>junk
```

will create an *empty* file named "junk" in the current directory.

Append Redirection (>>) There is an associated symbol ">>" which means "*append* the output to the *end* of the following file, rather than print on the terminal." The command:

```
  cat file1 >>file2
```

will add the contents of "file1" to the end of "file2". Note that if "file2" does *not* exist, UNIX will create it.

Input Redirection (<) Another universal symbol "<" which means "take the input from the indicated file, rather than the terminal." Assume we have a file "do_it" with the following contents:

```
  $d
  1m$
  $-9,$w
  Q
```

Now, we initiate the command:

```
  ed - file1 <do_it
```

which brings "file1" into the editor for a working session and accepts editor commands from file "do_it" (i.e., deletes last line of "file1", moves the first line to the end of the editor buffer, write onto "file1" the last 10 lines of the editor buffer, and quits the editor) rather than from the terminal. It is possible to combine input and output redirection in a single command.

Shell Description Before continuing, the term "shell" will appear from time to time, but will not be explained in detail in these notes. The mysterious "shell" is simply a program which interprets what you type as commands and arguments; it acts as the interface (a transparent interface) between the user and the raw computing power of the machine. In fact, all redirections discussed here are "shell properties."

General "shell procedures" would require another large tutorial memorandum as an independent topic. A brief description of *elementary* "shell scripts," useful to a newer UNIX user is provided in Section 3.16. If you want more information, you should start by taking an offered course on "shell programming" to obtain up-to-date list of documents; there are many outdated "shell" documents, which can easily lead you astray, so be careful.

Pipelines (|) To understand the concept of a "shell pipeline," represented by the "|" (vertical bar) symbol, consider the following problem: we have three files "a", "b" and "c", which we would like to combine and print in a paginated format using the "pr" command. We could accomplish this with the following commands:

```
  cat a b c >tempfile    [combines files]
  pr tempfile            [prints combined file in paginated form]
  rm tempfile            [remove temporary file]
```

but a single command line containing a "pipe" would suffice, ie,

```
  cat a b c | pr  ["space" is not necessary before and after "|"]
```

The "pipe" takes the output from the first command (in this case, "cat a b c") and, instead of outputting it on the terminal (the "standard output"), forces it to be the "standard input" to the second command (in this case, "pr"). Any command, which would normally write onto the terminal, can drive a "pipe": any command, which reads from the terminal, can read from a "pipe" instead.

New Inputs for Printing Commands Note that the "pr" command without any "filename" argument accepts input from the "standard input," or in this case, a redirection from the "pipe output": the "standard input" means a series of text lines from the terminal, terminated by a line containing only CONTROL-D (the "end-of-file" character). The same holds true for the OFFLINE PRINT command "opr" and the CONCATENATE command "cat". Consider the command:

```
  cat one two | pr | opr
```

First the files "one" and "two" are combined. Next the combined file is passed through the "pr" command for pagination. Finally, the "opr" command produces the offline print of the combined file in paginated format.

Another example of interest is:

```
  date | cat - a b >c
```

The new "twist" here is the "-" argument in the "cat" command; this designation represents "standard input" from the terminal. In this case, "cat" accepts a redirection from the "pipe output." The above command will concatenate the "date/time" line (from the "date" command) with files "a" and "b", and write the entire text into the redirected output file "c".

There may be more than one "pipe" in a command line. Also, it should be noted that the "circumflex" (^) is the *old* "pipe" symbol, and it *is* still supported by UNIX.

Multiple Commands on One Line (;) Another command symbol is the "semi-colon"(;), which allows multiple commands to be executed (in the order listed) before control is returned to the terminal. The multiple-command:

```
  cat a b ; date
```

causes combined files "a" and "b" to be printed on the terminal, followed by a printout of the "date/time," and then control is returned to you.

Automatic Hang-Up A useful multiple command line is:

```
  a_long_running_command ; stty 0
```

Assume you have a time-consuming command to run, but you want to go home; simply add the command "stty 0" (which logically disconnects your terminal from UNIX, or produces an equivalent "hang-up") to the end of a multiple command line.

Grouping Commands Commands enclosed by "parentheses" and separated by "semi-colons" are are treated as a single command. such as:

```
  (cat a b ; date) | pr
```

will combine files "a" and "b" (and does *not* print them on the terminal because of the "pipe" designation). executes the "date" command (and again does not print it), passes combined files "a" and "b" followed by the "date/time" line through the "pipe" to the "pr" command, which treats them as one large file and prints them in a paginated format.

3.10 Background Processes

This section will cover the techniques necessary to execute commands in the background, while the terminal becomes immediately available for other uses. This is most useful in compiling or executing large "C" or text-formatting programs as a background process (which could take a reasonable amount of real time), while continuing to do other work.

Commands Processed in Background (&) To run a process (or execute a command) in "background," simply place an "ampersand" (&) at the end of the command line. UNIX will respond with a "process identification number," followed immediately by a "S" prompt, indicating you may continue with other work. This "process ID" uniquely identifies the process generated by the command line terminated by "&"; an exception is that many processes may be spawned by multiple commands in a "pipeline," in which case, the number of the last process in the "pipeline" is reported.

As a note, you may *not* "hang-up" on UNIX while background processes are still active, or these processes will terminate prematurely. It should be noted that background processes are assigned a slightly lower priority (i.e., they're slower) than those run interactively, and that the number of simultaneous background jobs is limited to 25 per user.

With another use for the "&" character just defined. it should be clear at this point that "metacharacters" and other special symbols will have different meanings when used in the "editor," or at the UNIX command level in file/directory manipulations. This causes a great deal of confusion to the casual user or beginner.

Process Status (ps) By issuing the command:

```
  ps
```

you may obtain information about all active processes, including those operating in "background." UNIX will provide a list of "process ID's," the terminal controlling the process, the cumulative execution time for the process (actual amount of computer usage, not the passage of real time), and the command that is being executed; these items are titled PID, TTY, TIME and COMMAND, respectively.

With the *full* listing option, i.e.,

```
ps -f
```

the above items are augmented by the UID (user ID of the process owner), PPID (the ID of the "parent" of the named process), STIME (the time the process was initiated), and full command names are produced (with all arguments listed).

Other options exist, i.e., "-e" will produce a listing of all active processes on your *entire UNIX system*, and "-a" will again investigate all UNIX processes, but only those that are associated with a *terminal (TTY) port*. You may also concatenate options, such as:

```
ps -af
```

will produce a full listing of all UNIX processes associated with a terminal in use. See the UNIX User's Manual for more PS options.

Killing a Command (kill) At some point, you may wish to terminate specific processes (without "hanging-up" and terminating *all* background work). The command:

```
kill -9 process_IDs
```

will "kill" or terminate the processes identified by a list of ID numbers separated by "spaces" (the argument "process_IDs"). The "-9" argument *is* optional, but, if included, guarantees a "sure kill." If the process you're killing is sensitive to an abrupt termination, you may substitute a "-2" option for the "-9"; this is equivalent to hitting the "BREAK" or "DEL(ETE)" key on a foreground process, and allows the process time to "clean up."

Immunity to Hang-Ups To make *any* command, *normal* or in the *background*, immune to "hang-ups" (including any disruption between your terminal and the UNIX machine), simply use the form:

```
nohup command
```

For "pipeline" or multiple commands, the "nohup" prefix must be inserted before *each* command in the line.

If the *output* of the command is *not* redirected to a file or sent to an offline printer, it will be sent to a file "nohup.out", which will be created by UNIX in your current directory, if it does not already exist; if "nohup.out" exists, output destined for the terminal will be *appended* to the end of this file.

Time-Delayed Commands (at) Another way to run background jobs is via the "at" command. You may specify:

```
at time date
command list
(Control-D)
```

The job list following the "at" command is terminated by a line containing only "Control-D" (simultaneously depressing the CONTROL and D keys). These jobs will be executed at the "time" and "date" indicated. The "time" may be specified by *one/two* digits, which is interpreted as *hours*, or by *three/four* digits, which is interpreted as *hours* and *minutes*; both designations assume a 24 hour clock, unless the suffix "am" or "pm" is used. The "date" may be the *month* followed by the *day number*, or a *day* of the *week* ("at" recognizes the month/day of week spelled out completely or abbreviated to three characters); also, two special days "today" and "tomorrow" are recognized. If the "date" is absent, *today* is assumed, if the hour specified is *greater* than the *current* hour; if it is less, *tomorrow* is assumed. Other variants for "time/date" exist; see the "at" page in the UNIX User's Manual for details.

You may also specify times incrementally via the form:

```
at now + increment
```

where the "increment" is a number suffixed by "minutes," "hours," "days," "weeks," "months" or "years" (the singular forms are also accepted).

Batch Processing (batch) You may also issue a command of similar format, i.e.,

```
batch
command list
(Control-D)
```

which submits a "batch job." Such jobs are run whenever the load on the system falls to an acceptable level.

One *important* note is that any "at" or "batch" job *cannot* send output to your terminal; all commands issued in this mode must have outputs *redirected* to a file (or an offline printer).

Listing and Monitoring At/Batch Jobs In response to either an "at" or "batch" command, UNIX will respond with a "job number." You can get a list of "job numbers" of all jobs not yet executed, by typing:

```
at
```

without any arguments. The list will include your login name, the "job number," an identifier *a* for "at" jobs or *b* for "batch" jobs, and the time when each "at" job is scheduled for execution (or when a "batch" job entered the UNIX system).

If you wish to see if your "at/batch" job is currently running, issue the command:

```
ps -f -u login_name
```

The jobs you are looking for will have a "?" process ID.

Terminating At/Batch Jobs (atz) If you wish to terminate one or more of your jobs before execution, use the command:

```
  atz job_numbers
```

where "job_numbers" is a series of numbers separated by "spaces."

3.11 Communications with Other Users

Inter-User Mail (mail) Within UNIX, there exists the capability of sending and receiving "mail." To *send* mail to another user on the same UNIX system (for example, disk files "h1" and "h2" belong to system "UNIX-H"), simply type the series of commands:

```
  mail login_name     [indicates person to receive mail]
  {text of message}
  .                   [line with "." only terminates message]
```

You may alternatively terminate the message with a "CONTROL-D" line, if you wish.

Note that a file's contents may also be mailed, via input redirection, i.e.,

```
  mail login_name <file_name
```

will "mail" the contents of the named file to person identified by "login_name".

If you attempt to send a letter to a *non-existent* login, UNIX will reply with

```
  mail: can't send to "invalid_login"
  Mail saved in dead.letter
```

where "dead.letter" is a file created in your "current" directory.

If you have mail *waiting*, UNIX will notify you upon login with the line:

```
  you have mail
```

The mail is stored in a special file "/usr/mail/login_name", and you can *retrieve* it by typing:

```
  mail
```

without any argument. Your mail is printed in reverse chronological order of receipt and is "postmarked" (the name of sender and the time of dispatch). After each message, UNIX issues a "?" query. A response of "d" will delete the current message and go on to the next; a response "p" will cause the message to be printed *again*; an "s filename" will cause the current message to be appended to "filename" in your current directory (if "filename" is omitted, "mbox" in your *login* directory is the default, and UNIX will create it , if it doesn't already exist); a "w filename" will append the message, without a "postmark," to the named file (file "mbox" is the default); a "q" will leave unread messages intact in the file "/usr/mail/login_name" and exit the "mail" command.

Mail to Another UNIX System If you wish to send "mail" to another UNIX system, you will have to modify the command format, as follows:

```
mail system_name!login_name
{text of message}
.
```

where a complete list of the destination system names ("system_name") can be found on the "usend" page of the UNIX User's Manual, and the "!" between the system and login names is mandatory.

Who is Logged on the System (who) If you wish to determine the users that are currently logged into the system, simply type the command:

```
who
```

A list of login names, their corresponding terminal identifications (TTY number), and the time that user entered the system, will be produced. A slight variation, i.e.,

```
who am I
```

tells you who you are logged in as.

Interactive Communications (write) The WRITE command copies lines from your terminal to that of another user on the same UNIX system; the destination user must be *logged on*, which may be determined by issuing the WHO command. When the command

```
write user
```

is first invoked, the following line is produced on the receiver's terminal:

```
Message from senders_logname senders_tty
```

Lines of input will then be sent from your terminal to the destination user, until a "CONTROL-D" line is typed: at this point, WRITE produces an "EOF" character string on the receiver's terminal and terminates.

You can also transmit the output of a UNIX command by beginning a line with the "exclamation point" (!) character. An example is

```
!cat letter
```

which will ship over the contents of file "letter."

When using WRITE, the receiver should *immediately* institute another "write" command to establish a two-way conversation. A communication protocol is also beneficial; it is suggested that individual messages should be terminated by a line with a lone "o" (for "over"), and that "oo" (for "over and out") should be sent when the conversation is to end.

As an alternative, you can pass the contents of a file by printing it on the terminal (TTY) of another user. Simply issue the WHO command to determine the "tty#" of the user you wish to surprise with the contents of your file. Then issue the following command:

```
  cat filename >/dev/tty#
```

When you first log on, receiving messages from WRITE or redirections through "/dev/tty#" is allowed by default.

Stopping Interactive Communications (mesg) If people become pests and messages are arriving in the middle of more important work, you can inhibit your terminal from being a recipient by typing the command:

```
  mesg n
```

You can allow allow messages at any time by issuing a counter-command, i.e.,

```
  mesg y
```

3.12 Sending Files to Another UNIX System

Sending Files Via USEND A vehicle exists to send *ordinary* files to a user on *another* UNIX system. The command "usend" has the following general form:

```
  usend -m -d dest -u user filenames
```

With the above command you will send the files (represented by the last argument "filenames", which is a list of files separated by "spaces" --- note that filename "metacharacters" *will* work in this listing) in your current working directory to a destination system (argument "dest") and a destination user (argument "user").

For a complete list of destination systems available, consult the "usend" page in the UNIX User's Manual or issue the command:

```
  cat /usr/asp/udest
```

for an up-to-date table. To find out the official "code name" of *your* system, issue the command:

```
  uname
```

The argument "user" is simply an appropriate "login" name on the destination system.

The "-m" argument is optional, but, if present, will report to the *sender* via "mail" when the file transfer is completed; the "destination login" *always* receives mail, when files are received from "usend". Also, the "sender" will always receive a "USEND JOB NUMBER," when the file has entered the "intersystem pipeline."

Creating an RJE Directory By default, the transferred files are delivered to a subdirectory "rje" (stands for "remote job entry") under the destination user's login directory. Subdirectory "rje" must have been previously created in "mode 777" for everything to work, and the "login" directory must be mode "755" (or "777"); the following commands (issued in the destination user's directory) will accomplish this:

```
  mkdir rje
  chmod 777 rje
```

The names of the destination files are the same as those sent, and will reside in the "rje" subdirectory upon completion of the transfer. Finally, the owner of the files transferred is the *destination login*, and the permission modes are those of standard file creation (usually "mode 644" ---- see the "umask" command later in this section). Note, if an "rje" subdirectory has not been created in advance, try *mailing* the files in question. The owner of the files transferred is the *destination login*, and the permissions associated with these files is that of the standard file creation process (usually mode "644" --- see the "umask" command later in this section).

As an example assume I am in working directory "/h1/kew", and issue a command:

```
  usend -d ihuxa -u eps file1 file2
```

which will send files "/h1/kew/file1" and "/h1/kew/file2" to the user "eps" on general purpose UNIX system A (via the destination code "ihuxa"). You will receive a response from UNIX, a USEND JOB NUMBER, indicating the file transfer will take place. Upon completion, the transferred files will appear as "/a2/eps/rje/file1" and "/a2/eps/rje/file2" (assuming login "eps" is on the "a2" disk file system).

There are other options/variations available; see the "usend" command page in the UNIX User's Manual.

An Alternative Via NUSEND An alternative to USEND is the "nusend" command. All arguments indicated above for "usend" are also applicable to this alternative command. In fact, from a users point of view, both commands provided the same service, with one exception: "usend" makes a copy of the files to be sent, while "nusend" uses pointers to the files to be transferred. Therefore, files sent via "usend" can be modified as soon as the USEND JOB NUMBER is received, and those changes will not be transferred to the receiving system; "nusend" copies files on a character per character basis, and any file modifications made while the transfer is in progress may be relaid to the receiving system.

Since USEND and NUSEND use different hardware, if you have both available between the sending and receiving UNIX system, you have an alternative to shipping files, if one transmitting facility is inoperative.

USEND/NUSEND Status You may determine the status of the USEND facility by issuing the command:

```
rjestat
```

Similarly, status of the NUSEND facility may be ascertained by:

```
nscstat
```

Sending Special Archives Via CPIO An *archive* is simply a group of files combined into a single file; it is not a simple concatenation, as the archive also contains an "index" of all files contained within it.

Although all the uses of archives are beyond the scope of this introductory document, I believe the average user can make use of some of its properties. Assume you wish to send all files within a given directory structure to another UNIX system; to date, the only resource is to issue NUSEND or USEND commands for each of the individual files. With archiving, the entire directory structure can be combined into a single file, with an associated "map" indicating how the files are organized within the archive. Now, this single file can be sent to another system via USEND (NUSEND), and at the receiving end, it can be "dearchived" into a duplicate of the original directory structure.

Creating a CPIO Archive The command "cpio" (copy in and out) will allow you to create a special type of archive, and then to dismember these archives into their original file/directory structure. To *create* a CPIO archive, use the command:

```
cpio -o
```

This command reads the "standard input" to obtain a *list of pathnames*, and copies these designated files, along with archive information, onto the "standard output." This obviously is not the most thrilling action available, but it has possibilities when you replace the "standard input" by a *pipe output*, and replace the "standard output" with a *file redirection*. Consider this alternative, i.e.,

```
ls -a | cpio -o >temp
```

Now, all filenames within the current directory ("ls" command) are piped to "cpio," and the archive of those files is redirected to a "temp" file.

If you intend to send this archive to a UNIX system running on different hardware (UNIX is an operating system which can be used on PDP-11, VAX, IBM, 3B-20, etc. computers), use a "-oc" rather than a "-o" option. This will produce archive header information in a universal ASCII format, which is portable between different processors.

Dearchiving CPIO Files The next step is to take an archive, previously created by a "cpio -o" command, and separate them into a group of files and directories. The required command is:

```
  cpio -i_plus_options <temp
```

where "temp" is illustrative of the file containing the archive. The argument "i_plus_options" means the letter "i" (indicating this is a cpio *input* process) concatenated with any of the option letters listed below:

d	Directories will be created as needed in the dearchiving process
c	Assumes the archive header is written in portable ASCII format
r	Interactively rename files (CPIO will printout original name, while user supplies new name -- if a null line is typed, the file is skipped)
t	Print "table of contents" of the archive (no files/directories are created)
u	Produces files *unconditionally* (normally, an older file will *not* replace a newer file of the same name)
s	Swap bytes.

Other options exist; the interested reader should refer to the CPIO manual page in the UNIX User's Manual.

Let's consider several examples. Assume we were asked to duplicate all *files* in the directory "/c2/kew/pony" on another UNIX system as directory "/b1/wek/horse." First, we must form an archive file, i.e.,

```
  ls | cpio -oc >animal
```

This command must be issued on the IHUX-C system, in directory "/c2/kew/pony"; it is assumed here that this directory contains *only* ordinary files. Note that the archive header is produced in portable ASCII format. Now, file "animal" must be sent to the "rje directory" of login "wek" on the UNIX-B system, via the USEND (or NUSEND) command, i.e.,

```
  usend -d ihuxb -u wek animal
```

Once transmission is complete, we enter the UNIX-B login and change directories to "/b1/wek/horse." Now "dearchiving" is accomplished via the command:

```
cpio -ic </b1/wek/rje/animal
```

For our second example, assume we wish to copy the entire *directory structure* under "/c2/kew/pony"; in this case, it is assumed that "/c2/kew/pony" could contain various levels of subdirectories, in addition to ordinary files. files. The procedure is as described above except the archive must be produced differently, i.e.,

```
find . -print | cpio -oc >animal
```

Recall that "find . -print" will produce a list of all files and subdirectories in "/c2/kew/pony,"assuming the command was issued in that directory. The only other change would be to add the "d" option to the "dearchive" command (cpio -icd) to ensure that new subdirectories will be created as needed under "/b1/wek/horse."

Selective Dearchiving Another interesting property of the dearchiving process is to selectively create only a *portion* of the files contained within the archive. Consider the modified command:

```
cpio -i_plus_options patterns
```

In this case, only files specified in the argument "patterns" will be used in the file creation process. "Patterns" is a *space-separate list* of filenames. To be effective, *filename metacharacters* ("?, "!", "*", "[...]", described in Section 3.7) should be used; they do match the "slash" character in pathnames. If "patterns" is missing, the default is "*" (i.e., select all files in the archive header). For example:

```
cpio -id *black *white <archive_file
```

will create directories as needed in the dearchiving process, but will only create files "black" and "white" found in *any* directory in the header of "archive_file."

3.13 Miscellaneous File Manipulations

Duplicating Directories with CPIO There is also a *pass* option for CPIO, which archives and dearchives in a single operation. It is extremely useful for duplicating entire directory structures within the same UNIX system. The general form is:

```
cpio -p_plus_options directory
```

where "p-plus-options" is the letter "p" (designating "pass files") concatenated with options "d," "r" and "u" previously described, and one more:

l   Whenever possible, *links* will be used, rather than copying files; this option is lower case "L," not the number "1."

All files produced will assume destination pathnames relative to the argument "directory." For example, assume directory "/c1/kew/pony" contains three files "black," "white" and "brown." Then the command issued in *that* directory:

```
ls ¦ cpio -pu /c1/kew/cows
```

will produce copies of original files "black," "brown" and "white" in the directory "/c1/kew/cows; the "u" option indicates the files will be copied *unconditionally*, even if those files already existed in the destination directory. This simple example made a copy of all files in a given directory, but the concept could easily be extended to an entire *directory structure*, which could not be accomplished with a single COPY command.

Searching for Strings (grep) If you're interested in locating specific character strings in a group of files without using the editor, you may wish to use the "grep" command. The name "grep" means GLOBALLY search a file for matches to a REGULAR EXPRESSION and PRINT the corresponding lines; the equivalent editor command would be "g/RE/p". This command searches a file for a particular character string, and has the general form:

```
grep -n 'regular_expression' filenames
```

where any "regular_expression" to be located is specified (the special meaning of "metacharacters" are valid). If the RE contains any characters with special meaning to the UNIX command processor, it should be enclosed by "single quotes." The "blank," "()", "[]". "{}", "^", "¦", "*", "$", "&", "?", etc. *all* have special meaning to the "shell." To "play it safe," it is wise to use the "single quote enclosure," unless the RE is a single, alphanumeric string. Also, if you wish to search for character strings containing either "single" or "double quotes," replace these symbols with the "." metacharacter. Remember that any RE "metacharacter" will have its special meaning "turned off," if *preceded* by a "backslash." The argument "filenames" obviously represents any files you wish to search (separated by "spaces," with filename metacharacters completely valid). The response from UNIX will be the filename and the line number (if the "-n" option is included), followed by the printing of the line containing the character string matched.

The "grep" command can also accept "pipe inputs," such as:

```
ls /usr/news ¦ grep mail
```

The "ls" command passes the names of all "news articles" in directory "/usr/news/" into the "pipe." The "grep" command accepts the "pipe output" and selects only articles containing the string "mail," printing their names on the terminal.

Searching for Fixed Strings (fgrep) Another related command is FIXED GREP "fgrep," which has the following form:

```
fgrep -n -x 'string' filenames
```

It will only find lines containing exact pattern matches to the "string" noted, with no metacharacter expansion permitted. It is faster than "grep," if you don't need the metacharacter properties. The "-n" and "-x" are optional; "-n" acts as previously described. and if the "-x" is present, *only* lines that match "string" in their *entirety* are printed.

Searching with Enhanced REs (egrep) Another interesting command is ENHANCED GREP "egrep." which acts like the ordinary "grep" command, except that it has a "richer" set of RE expansions. It accepts any RE described for ED (except *ranges* within "[..]" won't work), but allows the following enhancements:

- A regular expression followed by a "+" matches *one* or more of that RE.
- A regular expression followed by a "?" matches *0* or *1* occurrences of that RE.
- Two regular expressions separated by "|" matches strings that are matched by *either*.
- A regular expression may be enclosed by "(...)" for grouping.

For example, the following command will locate any lines of text *beginning* with a "." or *ending* with one or more "right parentheses," in any file in the current directory:

```
  egrep '(^\.)|(\)+S)' *
```

Note that the couplets "\." and "\)" seek literal matches for these characters, by escaping their "metacharacter magic."

Searching for Control Characters As described previously in Sections 2.7 and 2.8, the appearance of unwanted control characters can be quite annoying. The GREP command can be used to search several files, and locate all lines containing those unwanted characters, i.e.,

```
  grep -n "[^SPACE-~TAB]" filenames
```

Note..when typing in this command, **SPACE** should be replaced by a single *blank space*, and **TAB** should be replaced by the *tab character* (simultaneously depressing the "CONTROL" and "I" keys). The output produced will include the *filename* and *line number* of each line containing a control character, other than **TAB**.

If you only desire the *names* of files containing these control characters, change the "-n" option to "-l" (lower case "L") on the GREP command line.

Counting Words in a File (wc) Another useful command is WORD COUNT ("wc"). In general

```
  wc file1 file2 file3 ----
```

will count the number of *lines*, *words* (a word is a maximal string of printing characters delimited by "spaces," "tabs" or "new line") and *characters* in each file listed, and print the tabulations on the terminal, unless redirected. The *name* of the file "counted" is also printed, and if you count more than one file, a "grand total" will also be given.

Options "-l", "-w" and "-c" also exist, and then *only* the number of lines, words or characters are listed, respectively. These options may be combined, such as:

```
  cat junk trash | wc -lc >number
```

will place the number of *lines* and *characters* in a combined file, composed of "junk" and "trash", into file "number". Note that the "wc" command, without *filename* arguments, reads from the "standard input" (the terminal), or a "pipe redirection." Also, since the input is from a "pipeline," the WORD COUNT output will *not* include a filename designation.

Determining Differences between Files (diff) The DIFFERENCE ("diff") command:

```
  diff file1 file2
```

compares the two files listed and indicates the lines that must be changed to bring them into agreement. This can be a useful check of modifications made during a long editor session.

Typical output lines are headed by a printout containing line numbers and one of the following letters:

[1] "d", if lines must be *deleted* from "file2" to bring it into agreement with "file1"
[2] "a", if lines must be *appended* to "file2" to bring it into agreement with "file1"
[3] "c", if lines must be *changed* to bring the two files into agreement.

Following each "header" are the differing lines of text. Affected lines in "file1" (the first file argument) are flagged by an "<" at the beginning of the output line, while lines in "file2" (the second file argument) are flagged by ">".

For a more complete description and details of available options, see the "diff" manual page.

Splitting Files (split) There are occasions when a file becomes too large, and must be "split" into smaller manageable units. For example, this document was created by concatenating independently created segments. Now, if a typographical error has to be corrected, the entire document file is too large to be brought into the editor, and must be separated into smaller segments first. The command

```
  split -n file_name prefix
```

will split the named file into segments *n lines* long (if the "-n" option is missing, 1000 line files are produced by default), but will leave the original "file_name" intact. The new smaller files formed are named "prefixaa," "prefixab," "prefixac," "prefixad," and lexically so forth. Obviously, "prefix" is an option, and the user is free to choose any appropriate character string; "x" is the default, if this argument is missing.

Splitting Files by Context (csplit) Another command exists that allows you to dissect a given file into several smaller sections: it is called "csplit" or *context split*. It allows more flexibility than the ordinary "split" command in that the points where separation occurs are much more closely controlled by the user.

Consider a typical command line:

```
  csplit filename arg1 arg2 ... argN
```

This command will cause "filename" to be separated into N+1 sections; by default, the sections are placed in files "xx00,"."xx01,"..., "xxN," where "N" must be *less* than 99. The contents of these new files is based on "arg1." "arg2," ... . "argN," as follows:

xx00:	Contents from the beginning of "filename" up to (but not including) the line referenced by "arg1"
xx01:	Contents from the line referenced by "arg1" up to (but not including) the line referenced by "arg2"
	....
	....
xxN:	Contents from the line referenced by "argN" to the end of "filename"

The three most commonly used "args" (other arguments exist --- see the UNIX User's Manual page for "csplit") are:

/RE/	An argument to locate the first line containing the *regular expression* "RE." You must enclose all regular expression arguments in *double quotes*, if they contain "blanks" or other characters with special meaning to the "shell." These arguments may be suffixed with an *increment* or *decrement* of some number of lines, such as "/RE/-7" or "/RE/-3".
line#	A *line number*.
{num}	If this argument follows a "regular expression" type, that argument will be applied "num" *more* times. If it follows a "line#" type argument, the file will be split into "num" segments, each of length "line#," from the point when the "line#" argument becomes effective.

Several options also exist (for complete details, again see the UNIX User's Manual), the most useful of which is:

```
  -f prefix
```

This will cause split files to be named "prefix00," prefix01," ..., "prefixN," instead of the default "xx00," "xx01," etc.: the argument "prefix" is illustrative of any character string. Finally, it should be noted that the original file "filename" will be untouched by the splitting process.

Consider the following file:

```
  one
  two
  three
  four
  five
  six
  seven
  eight
```

and the following command:

```
  csplit -f num /f/-1 4 2 {2}
```

The resultant split files are:

```
  num00  num01  num02  num03  num04

  one    three  four   six    eight
  two           five   seven
```

Stream Editing (sed) The *stream editor* ("sed") copies the named "file" (the default is the "standard input," if no file is designated on the command line) to the "standard output" (unless redirected), edited according to specified scripts. The SED command has the following general form:

```
  sed scripts file
```

The *scripts* consist of editing commands, which will be described below; they may be entered onto the command line as one of two option forms, i.e.,

-e script In this case, the script is embedded in the command line. The editing command should be enclosed in "single quotes" to avoid any misinterpretation of blank spaces or metacharacters.

-f sfile With this option, single line editing commands are extracted from a file "sfile" prepared in advance.

The types of script options may be intermixed, but they will be executed in the order encountered on the command line.

The normal operation of SED will cyclically copy a single line of input into a working buffer, called the *pattern space*. Then it executes all editing commands, whose *addresses* select the "pattern space" (either by line number or context match); editing commands are investigated in the order presented on the SED command line. After all editing commands have been investigated, the "pattern space" (possibly modified) will be written into the "standard output" (or a file will be overwritten via redirection). This concept of working with a "pattern space" allows SED to handle files of *unlimited* size, whereas the normal editor buffer is limited.

Next, a description of typical *scripts* must be investigated. In the description below, "#" indicates a *line number* within a file; the editing command that follows will be executed on every "pattern space" that matches this address. The number may be decimal, "S" for the last line in the file, or a *context address* of the form "/regular expression/"; these are analogous to the style of the ordinary editor previously described, noting "." (representing the "current line") can *not* be used. The designation "#,#" indicates a *range* of lines; the editing command that follows will be executed for every "pattern space" that matches the inclusive address range indicated. Again, in the style of the ordinary editor, the second number in a line range must be greater than the first, and a "range" can always be replaced by a single line number, if desired; abbreviations for line ranges "," and ";" can *not* be used. If line numbers or ranges are *omitted* from an editing command line, then *every* "pattern space" is selected by default.

Typical SED editing commands include:

#a — This is the *append* command; it is assumed that one or more lines of text will follow. The command line and all lines of text, *except* the last one must be terminated by a "\" character. SED will write the designated text lines on the "standard output" *before* bringing in the *next* line into the "pattern space."

#i — This is the *insert* command. Action is similar to "append" except the text lines are written onto the "standard output" *immediately*.

#,#c — This is the *change* command. This will delete the "pattern space" of any lines designated by the command range. Then, after the "pattern space" corresponding to the *last* line in the range, the "text" lines associated with the "change" command will be written onto the "standard output" before the next line is accessed by SED. Like "a" and "i," the command line and all lines of text, *except* the last, must be terminated by a "\" character.

#,#S/RE/replacement/flag

This is identical to the normal editor *substitute* command. Feel free to use regular expression and replacement text metacharacters, and the "g" flag for substitutions globally across the line.

#r rfile — Analogous to the editor *read* command. Writes the contents of file "rfile" onto the "standard output" before SED reads in the *next* input line.

#.#d	This command will *delete* the "pattern spaces" that correspond to the line range indicated.

#.#y/string1/string2/

The *transform* command replaces all occurrences of characters in "string1" with "string2." The lengths of "string1" and "string2" must be identical.

It should be noted that many more editing commands are available; the interested reader should interrogate the manual page for the SED command and a document "SED - A Non-interactive Text Editor" by McMahon.

To complete this section, let's consider a few examples, such as the SED command:

```
sed -e 's/\**/-/g' -e '4d' -e '3.5s/Jim/John/' junk
```

and file "junk" contains:

```
***NAMES**
Jerry
Jim
Jim
Jake
**END****
```

The printout produced by SED is:

```
-NAMES-
Jerry
John
Jake
-END-
```

Note that the RE "**" represents one or consecutive "*" characters, and that the "pattern space" for line #4 would be removed before the substitution from "Jim" to "John" could be attempted.

For our second example, consider the command:

```
sed -f script_file greeting >good
```

where the "script_file" contains:

```
s/Happy/Merry/
3a\
Happy
/Dumpy/d
```

and file "greeting" contains:

```
Happy
Christmas
and
Dumpy
New Year
```

The results of the SED command, redirected to the file "good", is:

```
Merry        [via substitute script]
Christmas
and
Happy        [via delete and append scripts]
New Year
```

Printing the End of a File (tail) The command TAIL will copy the *last part* of a file to the "standard output" (the terminal); this is useful in checking where you ended the last editing session or for splitting files. An example is:

```
tail -6 junk
```

which will print the *last* six lines in the file "junk." The "-6" option may be replaced by "-#" in general: "#" may be as large as the total number of lines in the file, or the limit of the temporary buffer utilized by the "tail" command. If no option is specified, the default is "10 lines."

Trying a slightly different format, i.e.,

```
tail -8 junk >>garbage
```

Note the "-#" option will cause copying to begin "#" lines from the *beginning* of the file, rather than the end. Also illustrated is redirection; in this example, lines #8 to the end of the file "junk" will be appended to the end of file "garbage."

Continuous Tailing (tail -f) One final example to consider has a "to follow" option, i.e.,

```
tail -f junk
```

In this case, the last ten lines of the file "junk" will be printed, *followed* by any lines appended to "junk" between the time TAIL is initiated and killed. An "endless loop" is entered, wherein TAIL sleeps for a second and then attempts to read and copy further additions to the input file "junk"; the process may be terminated by hitting the "DEL (ETE)" or "BREAK" key. This process is very useful for monitoring the creation of a file via a background command.

Updating File Timestamps (touch) The TOUCH command causes the *access* and *modification* of named files to be updated The command:

```
touch mmddhhmmyy files
```

will "time stamp" the named "files" with *mmddhhmmyy* (i.e., "month-day-hour-minute-year"): note that the "yy" field is optional, and, if the entire "time stamp" is missing, the default is the current time. The use of touch is very useful, if you are involved with project software management systems that treat files differently, dependent upon their "time stamp."

3.14 Other Useful Commands

Printing the Date and Time (date) The DATE command, discussed previously, is quite simple, as typing

```
  date
```

in response to a command prompt "$", will cause UNIX to print the current date and time on the "standard output" (the terminal), unless redirected.

Options exist --- see the UNIX User's Manual for complete details, but note that

```
  date +%D
```

produces a "date" printout of the form "month/day/year", and

```
  date -DATE:%D%nTIME:%r
```

will produce an output

```
  DATE:month/day/year
  TIME:hour/minute/second AM/PM
```

UNIX News reports You might want to consult "current news reports" concerning UNIX by typing the command:

```
  ls /usr/news
```

which will print a list of "news articles," some of which, may be of interest to you.

Many systems have the article "phones," which may be of particular interest to you, as it contains a listing of all the telephone access codes for general purpose UNIX machines at IH. (If this news item does not exist on your system, interrogate files in the directory "/usr/adm/info".)

To view a particular article, simply type:

```
  cat /usr/news/name_of_article
```

Filename metacharacters may be used freely in the "name-of-article".

An abbreviated command (the "article_names" must be "space-separated" and spelled out completely, without metacharacters) is:

```
  news article_names
```

If you decide you're really not interested in the article after all, hitting the "DEL" or "BREAK" key will cause printing to cease and return you to the command mode, if you use the "cat" version first described above. If you use the abbreviated form, a "DEL/BREAK" will *only* cause the printing of the *current* article to cease, and then another article is started (if possible): another "DEL/BREAK," within a second of the first, causes exit from the "news" command.

Note, when you first login, UNIX will notify you as to all current "news reports" that you have *not* yet seen; to view these current items, simply type:

```
  news
```

in response to a "S" command prompt. A marker ".news_time" keeps track of those articles not yet seen by you; do *not* destroy this file or UNIX will request that you read *all* news articles *again*. In this case, you may issue a command:

```
  news >temp_file
```

which will deload the "news" queue and store these articles in a file for later inspection, or

```
  news >/dev/null
```

if you want the news articles destroyed, as "/dev/null" is a *write-only* file (or a "bottomless pit").

Online Manual Pages (man) The MANUAL ("man") command will print the latest copy of the page(s) in the "UNIX User's Manual" of any command of interest, such as:

```
  man 1 at
```

would give full instructions of how the "at" command is utilized. The "1" option indicates a search of only "Section 1" of the manual will be made (where all commands discussed in this course are located). An obvious drawback, is that you must know the *exact* abbreviation for the *command argument* for "man" to work; situations, such as the "ln" and "mv" commands being located on the "cp" manual page, add to the confusion. The command is extremely useful in obtaining "current updates" of command capabilities (or a description of *new* commands), before they are formally published.

For a nice "hard copy" (an 8 by 11 Xerox print with four holes punched) of a manual page, try the command:

```
  man 1 at | opr -txr -p port
```

or

```
man 1 -T9700 at
```

Changing Your Password (passwd) The PASSWORD ("passwd") command allows you to change the "password" associated with your "login name," via

```
passwd
```

for which UNIX will request your old and new password, and a confirmation of that new password (since "echoing" or printing is turned off during the change process). UNIX also requests the number of months you wish the new "password" to be valid, and at the end of that interval, the system will automatically invoke the "passwd" command. UNIX will not allow you to change your "password," if it has been changed recently. Also, it will not accept "extremely simple passwords" (usually, at least six characters are required, unless you use a "rich" mixture of letters, numbers and other symbols); it is anticipated that even more stringent requirements will be instituted shortly.

Finding Spelling Errors (spell) A very useful command is SPELL, which will try to find typographical/spelling errors in text files. The command:

```
spell file_name
```

will produce a list of likely spelling errors in "file_name." It should be noted that SPELL is not perfect, and will usually err via omissions. Also if the file to be interrogated is long, you can take a nap before results are printed; an alternative is to redirect the output to a file, and make the execution a background process, such as:

```
spell file_name >spell_file&
```

3.15 User Profile and Options

User Information A special administrative file you may find useful is "/etc/passwd", which contains a multitude of information about all logins on *your* UNIX system. To determine information associated with a specific login, simply issue a command:

```
fgrep login_name /etc/passwd
```

which will cause a line of the following form to be printed out:

```
login_name:passwd:UID:GID:dept-name(acct)bin:pathname
```

noting,

[1] "login_name" is the login you've requested information about

[2] "passwd" is that login's password, encrypted into an unintelligible code

[3] "UID" is that login's equivalent "user's identification number"

[4] "GID" is that login's equivalent "group identification number"

[5] "dept-name" is the login's "department number," followed by his or her real "name"

[6] "(acct)" is an administrative indication of the login's account, which appears on all Computer Services Request forms and UNIX usage billing

[7] "bin" is the "bin number" associated with the login, to which all offline printing will be sent, unless redirected by the "-b" option associated with the "opr" command

[8] "pathname" is the "full pathname" of the corresponding login directory, indicating his/her "disk file system."

Group Information The file "/etc/group" contains all pertinent "group information." The command:

```
  fgrep login_name /etc/group
```

will provide pertinent "group" information, associated with *all* groups in which "login_name" is a member. A typical printout is of the following form:

```
  group_name::GID:group_members
```

where "group name" represents the *owner* of the group, "GID" is the numerical equivalent of that owner, and "group_members" is a comma-separated list of the members of that group (the first entry is always the group owner).

If you wish to determine information *only* about the group *you* own, issue either the command

```
  grep '^your_login' /etc/group
```

or

```
  fgrep GID /etc/group
```

noting the leading "^", in a "regular expression," indicates a match to a line containing the string "your_login" will only be made if it appears at the beginning of a line (note that the "single quotes" *are* necessary, as UNIX recognizes the "^" character as an alternative to the "pipe" symbol). Alternatively, the second form uses the argument "GID," which may be determined after the contents of "/etc/passwd" or "/etc/group" have been examined *once*.

Who is Doing What If you're really nosey, and you want to know what each logged in user is doing, issue the command:

```
/etc/whodo
```

which will produce a merged and reformatted output from the "who" and "ps" commands. If you want to know who is logged in, but not doing any work, type in:

```
whoo
```

Information produced is similar to the WHO command. except the next to the last column indicates the number of minutes that a particular user has done *no* work.

Customizing Terminal Options (stty) An important command SET TERMINAL OPTIONS ("stty") invokes certain input/output options, some of which are very important to the "C-language programming" or "text formatting." The general form of the command is:

```
stty option1 option2 option3 .......
```

where possible options are:

[1] The option "-tabs" will replace the "tab character" by an appropriate number of spaces when printing on the terminal. On terminals available, tabs are set across the page at an "8 space" interval. If this option is set, every time a "tab character" is encountered in printing, the carriage will move to the *next* "tab stop"; "tabs" are introduced into text via the "CONTROL-I" key (simultaneously hitting the "CONTROL" and "I" keys). *Without* this option, when reading a file containing "tab" characters, they will be *ignored* by the terminal.

[2] The option "erase Z" will cause "Z" ("Z" is illustrative of any single character, such as "^h" which represents "CONTROL-H" or the "backspace" character) to be the "erase character," rather than the default "#" symbol. Note the "^" can be used to represent the CONTROL key *only* in the "stty" command.

[3] The option "echoe" will replace the *erase character*, whatever it is defined as, with the sequence "backspace-space-backspace." This has the effect of typing the erased character anew on a *CRT* type terminal; the option should *not* be used on a "printing terminal."

[4] The option "kill Z" similarly makes "Z" the "kill character" rather than "@" (many people prefer "^x").

[5] The option "ek" resets the "erase" and "kill" characters back to the default "#" and "@", respectively.

[6] The option "-echok" will inhibit the "new line" produced (as a default) after the "kill" character has been invoked; the option "echok" will return the default mode.

It should be noted that you may use the "stty" command as many times as you desire during a UNIX session, but when you log in anew, you will return to the usual "default options" (unless you take special action --- see the "user profile" topic, which will be discussed next). Also, see the UNIX User's Manual for other less used "stty" options.

User profile There is a special file of interest: ".profile". After logging in, UNIX (the "shell" in particular) searches your "login directory." If it finds a file called ".profile", the commands therein are executed first, before reading commands from the terminal. You were assigned a standard ".profile" when you applied for your "user ID." Open ("cat") it to view its contents; there may be many commands within it that you are unfamiliar with --- these incorporate "shell procedures," most of which need not concern you at this introductory level. What is important is that you can edit this file to add "personal" commands that are executed before you receive your first "$" prompt. I find the command "stty -tabs" a useful addition, especially when working on "C programs": some people like the date/time printed, via the "date" command; I also modify my "erase" and "kill" characters via a "stty" command (note that the modified "erase" and "kill" characters will only take effect *after* you receive your *first* command prompt --- during the "login sequence," you must use the *default* "#" and "@" characters).

File/Directory Creation Mask (umask) You should have a line

```
umask 022
```

in your ".profile". This indicates the *permission modes* associated with files (or directories), as they are created by the "editor" command or redirections (or by "mkdir").

You may determine your current "creation mask" by issuing the command:

```
umask
```

without any arguments. Similarly, you may change the "mask" via

```
umask ABC
```

where "A", "B" and "C" can be integers in the range "0" to "6". The "permission mode" during creation will be equal to "666 - ABC" for *ordinary files* and "777 - ABC" for *directories*. Thus, if "ABC = 022," files are created in "mode 644," and directories in "mode 755," which has been implied throughout this document.

Where You Find Commands When you issue a command, UNIX (actually, the "shell") must locate the directory in which the command is found. The standard is to search your "current" directory first, then "/bin", then "/usr/bin", then "/usr/lbin", and, finally, the subdirectory "bin" in your login directory (used to store custom commands --- see Section 3.16 for more details): once the command is found, it is executed and the search ceases. This default search scenario is dictated by the line in ".profile":

```
PATH=$PATH:$HOME/bin
```

You may alter the directory search by changing the "PATH=" line, which has the following general format:

```
PATH=path1:path2:path3:path4:.........
```

i.e., a sequence of full directory pathnames separated by "colons"; your "current" directory is indicated by the "null string" or a single "." abbreviation. For example

```
PATH=:/bin:/usr/bin:/usr/lbin:/h1/kew/bin
```

is equivalent to the default search for *my* login directory, since the first pathname is the "null string," representing the "current" directory. Another example is:

```
PATH=/bin::/usr/bin:/usr/games
```

which searches for commands in directory "/bin" first, then in the "current" directory (again represented by a "null string"), then in "/usr/bin", and finally in the "/usr/games" directory (no games now --- you're supposed to be working).

Defining Other Profile Variables There are other variables of interest. You may note a ".profile" line of the form:

```
MAIL=/usr/mail/login
```

which tells the MAIL command where to look for your inter-user letters.

You may change your *command prompt* from the default "$" by defining a new variable, such as:

```
PS1="YES MASTER"
```

to humble the "damn computer" (YES MASTER is only illustrative of a new command prompt --- use your imagination).

You can predefine variables to control the default "bin," "type" and "forms" used by the OFFLINE PRINT command. For example, adding the following lines to your "profile":

```
BIN=49
DEST=IH6G4
TYPE=xr
FORM=nohole
```

will cause all "opr" output to be directed to "bin" number 49 (rather than the default in the "password" file) in the remote center located at IH, "aisle 6G4" (rather than the default local computation center), and output will be nohole, landscape Xerox format (rather the default line printer format).

While many variables are "set" at login time automatically by including them in your ".profile", you may reassign the value of any variable (or define new ones) any time after you are logged in. The definitions will only be in effect for *that* working session or until redefined again; the benefit of utilizing ".profile", is that these parameters will be defined *every* time you log in.

Exporting Variables (export) You will also see a line in ".profile" headed by the word "export," following by several variable names. When you log in, you are assigned a "working shell" and the commands in ".profile" are executed in that environment. Unless you "export" variables of interest that you've defined, whenever a new process is "spawned," it will not know about any nondefault variables; especially noticeable will be a reversion to the default "PS1" and "PATH" variables.

User Environment (env) To recap your "working environment," you may issue the command:

```
env
```

The printout UNIX provides will include PS1 and PATH (as described above), LOGDIR (the full pathname of your "login directory"), LOGNAME (your "login name"), LOGTTY (the full pathname indicating the "terminal port" you are connected to), and other less useful environmental parameters.

Printing Arguments and Parameters (echo) The ECHO command has a simple form, i.e.,

```
echo argument_list
```

and provides the simple function of writing its arguments on the "standard output," separated by blanks, such as:

```
echo dog cat pony
```

will yield a printout:

```
dog cat pony
```

At first glance, you would think such a command is not only simple, but also useless; it does, however, have some redeeming qualities.

Consider the command:

```
echo *
```

As noted previously, the filename metacharacter "*" represents all files in the current directory (except those beginning with a "."). Therefore, those filenames will be listed with as many names as possible packed into each line. The information outputted is identical to the "ls" command, except it is more compact.

Another use for the ECHO command is to evaluate environmental variables; if you want to know your current *path*, issue the command:

```
  echo $PATH
```

It works equally well with any previously defined parameter. but remember to precede the parameter name by a "$" or you will not witness the desired expansion.

3.16 Rudimentary Shell Programming

Simple Shell Scripts Although "shell programming" could be the basis of another long memo, there are certain features that even a beginner would find useful. At a rudimentary level, a "shell script" is simply an *executable* file containing a series of UNIX commands.

For example, enter the editor and create a file "junk" with the following lines:

```
  echo "SHELL SCRIPT"
  echo *
  ls -ld garbage
```

If you typed in each of these lines sequentially, you would expect an output stream of the character string "SHELL SCRIPT," followed by a compact listing of all file and subdirectory names contained within the current directory, followed by long listing status information for subdirectory "garbage" (if it exists). Executing the "shell script" (which is equivalent to executing the file "junk") would produce identical results. To accomplish this, simply change the mode of file "junk" (via the "chmod" command) so that it is *executable* by the owner (and/or group, other users). Then, by issuing the command:

```
  junk   [full pathname, if you're in a different directory]
```

you will cause the contained commands to execute.

Positional Parameters This still doesn't really allow you the flexibility generally desired, as the "shell script" is *fixed*; it would be nice to be able to write simple scripts, with *variables* embedded that can be set when the "shell script" is executed. Consider a single line command:

```
  pr $1 | opr -c$2 -txr
```

which would paginate a file named "\$1" and produce "\$2" printouts in default form (via the "pr" to "opr" pipe). Items "\$1" and "\$2" are *shell variables*; it should be noted that naming files with a leading "\$" is bad practice in general, as will be seen in the following development. Now, create a file named "junk," containing the above line and make it executable. If the following command is then issued:

```
  junk garbage 14
```

the following shell execution will take place:

```
  pr garbage | opr -c14 -txr
```

Therefore, the first argument on the "junk" command line (i.e., "garbage") will be substituted for "S1" in the shell script lines, and the second argument "14" will be substituted for the variable "S2"; in general, the argument "n" will replace a variable labeled "Sn" in the corresponding shell script, contained, where "n" is the integer *position* of the argument on the command line.

We can improve this command slightly by forming a two line "shell script," i.e.,

```
  echo "pr S1 | opr -cS2 -txr"
  pr S1 | opr -cS2 -txr&
```

This script is identical to the previous one, except the command which will be executed is "echoed" first, and the actual "pr/opr" command will be executed in background. Note that the use of the ECHO command in the form:

```
  echo "comment"
```

has interesting properties. By enclosing the "comment" string in "double quotes," all characters (including *filename metacharacters*) will be echoed exactly, but "Sparameters" will be expanded in the printed output.

Positional Parameters Using Metacharacters Assume that we have three files "cow," "dog" and "horse" in our current directory (those are the only files in that directory), and that we would like to produce two paginated printout of each. We could issue the following series of commands, using our "junk" executable file:

```
  junk cow 2
  junk dog 2
  junk horse 2
```

or you may get clever and try to use *metacharacters* to save time. The command:

```
  junk * 2
```

will *not* work, as metacharacter "*" is expanded into three arguments "cow," "dog" and "horse" on the command line, and, therefore, the following (nonsensical) equivalent command is actually executed:

```
  junk cow dog horse 2
```

Now "cow" would be passed as parameter "S1" and "dog" as "S2," while the other arguments "horse" and "2" would not even be used in the script; this is obviously not what is desired. What you want to do is to pass the argument "*" as the single parameter "S1" in the shell script. To stop the expansion of characters which have special meaning to the "shell" on the command line, simply enclose them in "single quotes," and they will be passed to the script as a single variable. Therefore, either

```
  junk '*' 2
```

or

```
  junk 'cow dog horse' 2
```

are equivalent.

This ends our mini-discourse on "shell scripts"; once you become an experienced UNIX user, you may wish to pursue shell programming in greater depth.

Your Private Bin Once you have a group of these "shell scripts" for your own use, create a directory called *bin* under your login, and modify your PATH in ".profile" (as shown in Section 3.15) to look in this new directory for your private commands.

3.17 Where To Go From Here

In addition to "shell programming," there are numerous other commands which may be of interest to the individual user, such as:

- An emulated desk calculator (see "dc")
- An Assembler, or Fortran or Basic compiler (see "as," "fc," and "bs")
- Fancy text formatters (see "mm," "nroff," "tbl," and also various papers in "Documents for PWB/UNIX Time-Sharing System," or my *great* memo "Preparing Documents On UNIX (Version 4.0)"
- Games (see Section VI of the UNIX User's Manual --- "/usr/games/startrek1" isn't listed, but it's on the computer for your enjoyment, after 4:45, of course)
- The "C" programming language and compiler, and an input/output package for UNIX (see the "cc" command page in the UNIX User's Manual, and the book "The C Programming Language" by Kernighan and Ritchie, published by Prentice-Hall)

------ ENJOY YOURSELF -----

4. HOMEWORK

4.1 LEARN Program

The LEARN program is a good primary source of practice problems to gain familiarity with the editor and the UNIX file structure.

To *enter* the LEARN program, simply type "learn" in response to a "S" command prompt. You will be given explicit instructions on how to proceed, including a list of offered topics. You will be interested in the topics: "files," "editor" and "morefiles" (in that order). Each "course" (or topic) will be presented in a "tutorial manner," i.e., each lesson is prefaced with the background principles necessary to answer a specific question (this ideal is usually realized). The "teaching script" can follow three paths: a "fast" track for those who make *no* mistakes, and two levels of "slow" tracks (i.e., extra lessons) for those who are *not* perfect. If you're stuck on a question for a long time, you have the option (by refusing to try the problem again) of passing it up and going on, probably down a ."lower track."

To *exit* the LEARN program, simply type "bye" in response to a "S" command prompt. You may reenter a subject at the point where you left off, by remembering the subject *name* and the *number* of the last lesson you've successfully completed. Note that it is wise to use a *hard-copy* terminal while executing the "learn" program, so important information will not roll off the screen.

4.2 Final Exam

After completing those sections of the LEARN program noted above, you might try this FINAL EXAMINATION (solutions are given in the next section). Do all the "examination" problems in a subdirectory "EXAM", which may be created via a command issued in your "login directory":

```
mkdir EXAM
```

and can be entered via the command:

```
cd EXAM
```

You can return to your "login" directory at any time by issuing the command:

```
cd
```

It should be noted that the following problems are quite "artificial," in that you may be *forced* to execute commands in a strange or impractical way, or analyze commands with little or no practical use. The intent of the problems is to make you aware of *all* of the commands at your disposal, and then *you* may choose a *subset*, and develop a style, suited to your needs.

Problem 1

Form a file named "begin" (in subdirectory "EXAM"), with contents:

```
  one
  two
  three
```

Problem 2

Enter the editor anew (type the "ed" command with *no* filename associated with it). Load the following lines into the editor buffer:

```
  four
  five
  six
  seven
```

Use the "f" command to associate the filename "number" with the buffer. Next, "read" the contents of file "begin" at the *beginning* of the buffer. Check to see what filename is *now* associated with the buffer, and "write" the seven line contents of the buffer onto file "number", and quit the editor.

Problem 3

Bring the file "number" into the editor buffer, print its contents, and try the following commands, in the order shown. Try to figure out the editor responses or actions attributable to each command *before* actually working with the terminal.

```
  .=                         [Command #1]
  8p                                 [#2]
  .=                                 [#3]
  4,5p                               [#4]
  -1,$p                              [#5]
  p                                  [#6]
  ----                               [#7]
  .2n                                [#8]
  (carriage return)                  [#9]
  $-1,3p                            [#10]
  .=                                [#11]
  $l                                [#12]
  1,$-2w                            [#13]
  Q                                 [#14]
```

Problem 4

Bring file "number" back into the editor buffer and perform the following actions:

(1) Copy line #1 at the end of the buffer
(2) Delete lines #1 and #2 of the buffer
(3) Move lines #1 and #2 of the current buffer after line #3
    in the current buffer
(4) Copy lines #1 to #4 of the current buffer after line #3
    in the current buffer
(5) Change line #4 in the current buffer to two lines:

```
eight
nine
```

(6) Insert, at the beginning of the buffer, the line:

```
six
```

(7) Append the end of the buffer with line:

```
seven
```

(8) Write lines #1 to #9 only onto file "number"
(9) Quit the editor

Now try to figure out the current contents of the buffer, and *then* print it out for verification.

Problem 5

Create a file "welcome" containing the following lines:

```
Hello!
I am UNIX.
This is to
inform you
that I am
your
powerful master
and
that this
terminal
will electrocute
you, should you
make an
error
- - - I demand
perfection!!!
- - - I demand
respect!!!!
```

Now bring "welcome" back into the editor buffer and perform the following actions:

(1) Change the word "UNIX" in line #2 to lower case letters
    (this humbles the damn computer)
(2) Change the word "electrocute" to "notify" (use a global
    command to make the substitution, noting that "electrocute"
    is the only word of text, not at the beginning of a line,
    that begins with the letter "e")
(3) Change the "powerful master" line to two lines "humble"
    and "servant" (note, the line you seek is the only line
    ending in "er" --- use a "context address" to locate it)
(4) Change all appearances of "demand" to "would like"

(5) Remove any number of "!" at the end of any line
(6) To make UNIX friendlier, change the first line to read "Hello Hello Hello Hello Hello Hello" using a single substitute command
(7) Print the contents of the modified buffer (see how humble UNIX has become), and destroy the buffer, i.e., leave the editor without writing the buffer contents onto file "welcome"

Problem 6

Create a file "wisdom" containing:

```
  If nodeders
  nodet
  nodedings the
  way summers
  wrote sums,
  then the first
  woodpecker that
  came along would
  write.
```

Now bring the file into the editor buffer, and investigate the actions of the following commands (determine the action of each command *before* using the terminal):

```
  (1) 1s/node/buil/
  (2) +,Ss//%/
  (3) g/w[a-dr][c-j]../s//destroy civilization/
  (4) g/sum/s//program/
```

Print the entire buffer and overwrite the file "wisdom".

Problem 7

This problem is a ."eal (expletive deleted)." Form a file "weird" containing the following text (BE CAREFUL and ENTER TEXT EXACTLY AS SHOWN):

```
  To tire bhd nurd
  it takes to 4 a task: tile
  hb nord you bhink \\* should xxxake,
  multiply by 4, and find the hair
  change the tibble of measure
  to the next 4 tibble. Thus we
  locate 4 days for a hour
  tark.
```

Now, for each command listed (in the order shown), determine the substitutions that will be made (without using UNIX), and then use the computer to physically execute the commands; after *each* command is completed, print out the entire buffer and compare the results with your "prophecies." The commands are:

```
(1)  1,-5g/d/s//e/\
     s/b/t/
(2)  3s/x*/t/
(3)  1,.2s/n.rd/time/
(4)  3s/\\\\*/
(5)  v/eS/s/4/2/
(6)  g/ti[^m]e/s//estimate/
(7)  g/^[^i]/s/4/highest/
(8)  v/^t/s/..in..the.*//
(9)  g/[:,]/s/4/do/\
     s/\*/it/\
     s/xx*/t/
(10) g/2/s/ho/one-&/\
     s/^[^monkey]/al&/
(11) g/bb/s/..bb../unit/\
     +2s/rk/sk/
(12) w
     q
```

If you analyzed *each* command correctly, you are an EDITOR EXPERT.

Problem 8

Create the following files in the directory "/your_system/your_login/EXAM/":

(1) Create the file ".when", containing three lines:

```
one
t>w>>o
>thr>ee
```

where ">" represents the "tab" character, entered into text via (CONTROL-I)

(2) Create the file "was/where" (this is file "where" in an "EXAM" subdirectory called "was"), containing one line:

```
two
```

(3) Create the file "why" (enter exactly as shown with all spelling errors), containing:

```
If one advarces confidtly
in the directious of his dreams
and edevors to live a
life which he has imagined,
he will meet with sucess
unexeced in cummun huors.
```

(4) Create the file "was/words"

```
cane
cone
lane
lone
nine
none
```

Recall that files "begin", "number", "weird", "welcome" and "wisdom" in directory "EXAM" have been created previously in other problems.

Problem 9

Determine the effects of the following LIST commands issued in subdirectory "EXAM":

```
(1)  ls
(2)  ls was
(3)  ls -a
(4)  ls -t
(5)  ls -l
(6)  ls why where when
(7)  ls -d was why
(8)  ls -dl EXAM was
(9)  ls w*
(10) ls *h*
```

Problem 10

Again assuming you are in subdirectory "EXAM" under your login directory, perform the following actions:

(1) Print your working directory
(2) Change to the "was" subdirectory
(3) Create a copy of "welcome" in directory "EXAM" and place it in the current directory with name "hello"
(4) Append files "where" and "hello" in the current directory to the end of file "begin" in directory "EXAM" (your current working directory is still "was")
(5) Return to the parent of your current working directory
(6) Produce a paginated output on the terminal of all ordinary "non-dot" files in directory "EXAM" and all of its subdirectories --- as an aid, printout all the names of files in "EXAM" and its subdirectories first
(7) Print out all lines containing the string "one" (including the filename and line number) in all files in directory "EXAM" (including subdirectories), whose filename contains a "w"
(8) Form a file "count" which contains the number of lines, words and characters (and the filename) contained in file "weird"
(9) Use the SPELL command to determine the spelling errors in file "why"

Problem 11

Try the following problems using the OFFLINE PRINTER:

(1) Produce an 8x11 Xerox print of the manual pages for the commands "banner", "at", "usend" and "sleep"
(2) Produce a landscape Xerox print of the file "weird"
(3) Produce a default computer printout of combined files "weird" and "welcome"
(4) Produce a nohole, landscape Xerox print of the file "words"
(5) Produce a paginated Xerox print (two input pages per one output page) of any file in directory "EXAM", whose name contains an "h" or "1" (no subdirectories)

Problem 12

A few more problems for your enjoyment:

(1) Combine the words "HI THERE" with file "welcome" and print it in a paginated format (do not create any temporary files and do not modify "welcome")
(2) Form a file "WWW", which contains a list of all files and subdirectories in "EXAM" beginning with the letter "w"
(3) "Cat" the file "welcome", using a "nohup" prefix ---- explain what actions UNIX will take
(4) Print a list of filenames in subdirectory "was", followed by the DATE/TIME, followed by the contents of file "welcome" on the terminal, using a single command line
(5) Deposit all spelling errors of files beginning with the letter "w" in directory "EXAM" (not subdirectories) into the file "tmp" --- create a background process
(6) Have the output of the "date" command sent to file "777" in 10 minutes and the contents of file "welcome" printed offline in 2 hours
(7) Kill the job to be run 2 hours from now
(8) Mail yourself a letter "I love myself" and the file "was/words"
(9) Read your mail --- destroy the "I love myself" message and save the file sent (with the postmark) in file "junk"
(10) Create a file "NEWS" containing the names of all news articles containing the character "2"

Problem 13

Try a few more, i.e.,

(1) Print out file ".when" on the terminal
(2) Return to your "login" directory
(3) Modify ".profile", such that the "tab" character is recognized by the terminal

(4) "Logoff" and "login" again
(5) Issue the command:

```
PS1="READY "
```

---- what does this command accomplish?
(6) Return to subdirectory "EXAM"

Problem 14

Answer the following questions, but do *not* issue the commands *physically* to UNIX.

(1) You wish to send some files to login "abc" on disk file system "b2" --- what preparations must login "abc" have completed before the files can be transferred
(2) Write the command necessary to send copies of files ".when" and "was/words" (both in directory "EXAM") to login "abc"
(3) Under what complete pathnames will the copies of the files appear in login "abc"

Problem 15

Now its time to clean up the files you've created in directory "EXAM":

(1) Change the mode of subdirectory "was" to allow only write permission to yourself --- all other permissions denied
(2) Deny all permissions to everybody for the files "welcome" and "why"
(3) Remove all "dot" files
(4) Change the name of file "hello" to "welcome"
(5) Delete as many files in directory "EXAM" and all of its subdirectories as you can, without changing "modes" --- remove interactively
(6) If you have any files left, change modes as needed, and remove "EXAM" and its contents

Problem 16

Consider the following hypothetical situation:

- You are in an unknown directory on the UNIX-A system, containing files *one*, *two* and *three* and subdirectory *directory1*; "directory1" contains 62 ordinary files.

- You issue an "rjestat" command on the UNIX-A system, and you are notified that the RJE hardware is down; the computer center states the problem will not be corrected for two days.

- You need a Xerox printout of file "one" by tomorrow morning.

- You want copies of files "two" and "three" (but not file "one") to reside in login directory "/b1/xxx," which has "700" permission on it, but fortunately you know the password of login "xxx" on the UNIX-B system. Note that the UNIX-B and UNIX-A systems are using a VAX and IBM computer, respectively.

- There exists a directory "/b1/xxx/rje" with "777" permission, but you lost your UNIX manual and your knowledge of USEND and NUSEND only allows you to transfer *one* file at a time. This "rje" directory is empty now, and should be empty when you are done.

Your task is to accomplish the above actions (with constraints indicated) using *five* commands. You must use USEND or NUSEND, CPIO, ECHO and FIND in this exercise.

4.3 Examination Solutions

Below are my solutions for the preceding "exam" problems.

Problem 1

The solution is:

```
Sed            ["S" is the UNIX command prompt]
a
one
two
three
.
w begin
14             [character count]
q
```

Problem 2

The solution is:

```
Sed            ["S" is the UNIX command prompt]
a
four
five
six
seven
.
f number
number         [UNIX response]
0r begin
14             [character count]
f
number         ["r" does not change buffer filename]
w
34             [character count]
q
```

Problem 3

The results of the commands are summarized by the following editor session:

```
  $e number        ["$" is the UNIX command prompt]
  34               [character count]
  .=               ["dot" is set to the last line read in]
  7                [response #1]
  8p               ["8" is non-existent line]
  ?                [response #2]
  .=
  7                [response #3, "dot" unchanged]
  4,5p
  four             [response
  five                       #4]
  -1,$p            [equivalent to "6,7p"]
  six              [response
  seven                      #5]
  p                [equivalent to ".p"]
  seven            [response #6]
  ----             [equivalent to ".-4p"]
  three            [response #7]
  .2               [equivalent to ".+2n"]
  5       five     [response #8]
  (CR)             [equivalent to ".+1p"]
  six              [response #9]
  $-1,3p           [equivalent to "6,3p, invalid command]
  ?                [response #10]
  .=
  6                [response #11, "dot" unchanged]
  $l               [equivalent to "7l"]
  seven            [response #12]
  1,$-2w           [equivalent to "1,5w"]
  24               [character count, response #13]
  Q                [command #14, unconditional quit]
```

The file "number" now contains the first *five* lines of the buffer, which were written:

```
  one
  two
  three
  four
  five
```

Problem 4

The editor session necessary to accomplish the stated tasks is:

```
  $e number     ["$" is the UNIX command prompt]
  24            [character count]
  1t$           [command #1]
  1,2d          [command #2]
  1,2m3         [command #3]
  1,4t3         [command #4]
  4c            [command #5]
  eight
  nine
  .
  1i            [command #6]
  six
  .
  Sa            [command #7]
  seven
  .
  1,9w          [command #8]
  Q             [unconditional quit]
```

The buffer. upon quitting the editor, contained:

```
  six
  five
  three
  four
  eight
  nine
  three
  four
  one
  one
  seven
```

and the file "number" now contains the first nine lines of the final buffer contents.

Problem 5

These are the commands that I would use (not necessarily the only way to do the problem) are:

```
  [1] 2s/UNIX/unix/
  [2] / e.*/s//notify/
  [3] /erS/c
      humble
      servant
      .
  [4] g/demand/s//would like/g
  [5] g/!!*S/s///
  [6] 1s/.*/& & & & & &/
  [7] ,p
      Hello Hello Hello Hello Hello Hello
      I am unix.
      This is to
      inform you
      that I am
```

```
your
humble
servant
and
that this
terminal
will notify
you. should you
make an
error
- - - I would like
perfection
- - - I would like
respect  .
Q
```

Problem 6

The commands will accomplish the following:

[1] Line #1 will be converted to:

```
If builders
```

[2] The command is equivalent to "2,9s/node/buil/" and will affect lines #2 and #3 as follows:

```
built
buildings the
```

[3] The only match to the global regular expression "w[a-dr][c-j].." is the string "write" in line #9 ------ the command is equivalent to "9s/write/destroy civilization/" and line #9 becomes:

```
destroy civilization.
```

[4] The global regular expression "sum" is matched in lines #4 and #5, and the command is equivalent to "4,5s/sum/program/", producing modified lines:

```
way programmers
wrote programs,
```

The modified file "wisdom" now contains:

```
If builders
built
buildings the
way programmers
wrote programs,
then the first
woodpecker that
came along would
destroy civilization.
```

Problem 7

Here are the answers to the "world's most difficult problem" (in great detail):

[1] The global prefix "1,-5g/d/" is equivalent to "1,3g/d/", since the relative address "-5" is equivalent to ".-5" and "dot" is "7" (the last line read into the buffer); lines #1 and #3 contain the string "d", and are subject to the substitutions that follow.

For line #1, the two substitute commands are equivalent to "s/d/e/" and "s/b/t/", recalling that the "null" regular expression associated with the first substitute is equivalent to "d", the last regular expression used by UNIX. Therefore:

```
  To tire bhd nurd    [original line #1]
```

is changed to

```
  To tire the nurd    [new line #1]
```

For line #3, the two substitute commands are equivalent to "s/b/e/" and "s/b/t/", noting that the "null" regular expression in the first substitute command is *now* equivalent to "b", the *last* regular expression used by UNIX (during line #1 substitutions). Thus:

```
  hb nord you bhink \\* should xxxake  [original line #3]
```

becomes

```
  he nord you bhink \\* should xxxake
```

after the first substitution and

```
  he nord you think \\* should xxxake  [new line #3]
```

after the second substitution.

Wasn't that fun!!!!!!

[2] The command "3s/x*/t/" changes the *leftmost* occurrence of *zero* or more consecutive "x's" to a "t", in current line #3. Thus:

```
  he nord you think \\* should xxxake  [current line #3]
```

is changed to

```
  the nord you think \\* should xxxake [new line #3]
```

as UNIX found *zero* "x's" between the "beginning" of the line (a "new line" character) and the first letter "h", and placed a "t" there. Were you fooled (again)???

[3] The command "1..2s/n.rd/time/" is equivalent to "1.5s/n.rd/time", since the relative address ".2" is equivalent to ".+2" and "dot" was set to line #3 after the last substitution. Matches to the strings "nurd" in line #1 and "nord" in line #3 are the only ones found in the designated range; the "." in the regular expression represents any single character. Thus:

```
  To tire the nurd    [current line #1]
```

is changed to

```
  To tire the time    [new line #1]
```

and

```
  the nord you think \\* should xxxake   [current line #3]
```

is changed to

```
  the time you think \\* should xxxake   [new line #3]
```

That was an easy one!!!

[4] The command "3s/*/" is equivalent to "3s/*//p" (due to the missing final delimiter), which changes

```
  the time you think \\* should xxxake   [current line #3]
```

to

```
  the time you think * should xxxake    [new line #3]
```

Again, I've tried to trick you. The regular expression "\\*" represents *one* or more consecutive "\" characters; remember that "\\" is a *literal* "backslash" and "*" is a metacharacter.

[5] The command "v/e$/s/4/2/" is straight-forward. The entire buffer is searched for lines *not* ending in the letter "e", i.e., lines #3, #4, #6 and #7; in these matched lines, the leftmost "4" (if one exists) is changed to a "2". Thus:

```
  multiply by 4, and find the hair   [original line #4]
```

is changed to

```
  multiply by 2, and find the hair   [new line #4]
```

and

```
  locate 4 days for a hour   [original line #7]
```

is changed to

```
locate 2 days for a hour    [new line #7]
```

Lines #3 and #6 do *not* contain the string "4", and, hence, were *not* affected by the substitutions.

[6] The command "g/ti[^m]e/s//estimate/" is also quite straightforward. The buffer is searched for lines containing the string "ti", followed by any character *except* "m", followed by "e": lines #1 and #2 provide matches. In these lines, the first occurrence of the matched strings are replaced by "estimate". Thus:

```
To tire the time      [current line #1]
```

is changed to

```
To estimate the time  [new line #1]
```

and

```
it takes to 4 a task: tile       [original line #2]
```

is changed to

```
it takes to 4 a task: estimate   [original line #2]
```

[7] The command "g/^[^i]/s/4/highest/" will affect any lines *beginning* with any character *except* "i". All lines *except* #2 qualify, but the replacement of "4" with the string "highest" can only be accomplished in line #6, i.e.,

```
to the next 4 tibble. Thus we         [original line #6]
```

is changed to

```
to the next highest tibble. Thus we   [new line #6]
```

[8] The command "v/^t/s/..in..the.*//" can affect any line *not* beginning with the letter "t"; this includes lines #1, #2, #4, #5 and #7. The only match to the string "..in..the.*" occurs in line #4, i.e., " find the hair", and

```
multiply by 2, and find the hair  [current line #4]
```

is changed to

```
multiply by 2, and                [new line #4]
```

[9] This global command seeks lines containing the character ":" or ","; matches are found on lines #2, #3 and #4. On each of those lines, any or all of the substitutions "s/4/do/", "s/*/it/" and "s/xx*/t/" may be made, in that order. Note that the regular expression "*" represents the character "*" *literally*, and "xx*" matches *one* or more "x's" literally. Therefore:

```
  it takes to 4 a task: estimate  [current line #2]
```

is changed to

```
  it takes to do a task: estimate [new line #2]
```

by the first substitute command. and

```
  the time you think * should xxxake  [current line #3]
```

is changed to

```
  the time you think it should take   [new line #3]
```

via the second and third substitute command. No substitutions are possible in line #4.

[10] This global command seeks lines containing the character "2"; lines in the current buffer that match are #4 and #7. In each of those lines. the commands "s/ho/one-&/" and "s/^[^monkey]/al&/" will be attempted: the first command replaces the string "ho" with "one-ho", due to the replacement text metacharacter "&", and the second command will place the string "al" at the beginning of a line *not* beginning with an "m", "o", "n", "k", "e" or "y", due to the "^" and "&" metacharacters. Therefore:

```
  locate 2 days for a hour       [current line #7]
```

is changed to

```
  allocate 2 days for a one-hour [new line #7]
```

via both substitute command, and line #4 is unaffected.

[11] The last global command seeks lines containing the string "bb", and lines #5 and #6 qualify. In those lines, a six character string with the two center letters being "b's" (if one exists) is converted to "unit" via the command "s/..bb../unit/". *Two* lines *after* each matched line are also (possibly) affected by the command "s/rk/sk/". For line #5, the first substitution is effective, i.e.,

```
  change the tibble of measure  [original line #5]
```

is changed to

```
  change the unit of measure    [new line #5]
```

and. in this case, the second substitution has no effect on line #7 (i.e.,".+2"). The first substitution also has an effect on line #6, i.e.,

```
  to the next highest tibble. Thus we  [current line #6]
```

becomes

```
  to the next highest unit. Thus we     [new line #6]
```

and line #8 (i.e., ".-2") is effected by the second substitution, i.e.,

```
  tark.  [original line #8]
```

is changed to

```
  task.  [new line #8]
```

I knew you'd make it!!!!!

[12] The "w" command is obvious, and a recap of all the changes yields a new file "weird" containing:

```
  To estimate the time
  it takes to do a task: estimate
  the time you think it should take,
  multiple by 2, and
  change the unit of measure
  to the next highest unit. Thus we
  allocate 2 days for a one-hour
  task.
```

Problem 8

Assuming you are in your subdirectory "EXAM", the following editor sessions will create the files requested:

```
Sed                      ["S" is the UNIX command prompt]
a
one
t(CTRL-I)w(CTRL-I)(CTRL-I)o
(CTRL-I)thr(CTRL-I)ee
.
w .when
4                        [character count]
,d                       [empties buffer]
a
two
.
w was/where              [partial pathname]
4                        [character count]
,d                       [empties buffer]
a
If one advarces confidtly
in the directious of his dreams
and edevors to live a
life which he has imagined,
he will meet with sucess
unexeced in cummun huors.
.
w why
158                      [character count]
```

```
.d                      [empties buffer]
a
cane
cone
lane
lone
nine
none
.
w was/words
30                      [character count]
q
```

This editor session will only work properly, if the subdirectory "was" is created *beforehand*, via the command:

```
mkdir was
```

issued in directory "EXAM".

Problem 9

The list commands will yield the following responses, if issued in subdirectory "EXAM":

[1] The command "ls" produces an alphabetized listing of all "non-dot" files/subdirectories in "EXAM", i.e.,

```
begin, number, was, weird, welcome, why, wisdom
```

[2] The command "ls was" produces an alphabetized listing of all "non-dot" files in subdirectory "was", i.e.,

```
where words
```

[3] The command "ls -a" will yield the same response as "ls", except the list headed by the "dot" files:

```
.         [abbreviation for your current directory "EXAM"]
..        [abbreviation for the parent of "EXAM"]
.when
```

[4] The command "ls -t" will yield the same response as "ls", except the files are put in the order in which they were created or modified, most recent first, i.e.,

```
was, why, weird, wisdom, welcome, number, begin
```

[5] The command "ls -l" will list the same files as "ls", except that "long listing" information (which varies from user to user) is provided

[6] The command "ls why where when", noting file "where" is a file in directory "EXAM/was" *not* "EXAM", and "when" does *not* exist, produces a printout:

```
where not found
when not found
why
```

[7] The command "ls -d was why" produces a list:

```
was
why
```

noting the "-d" option tells you of the *existence* of subdirectory "was" (not the list of files contained in it), and has *no* effect on "why", which is *not* a directory.

[8] The command "ls -dl EXAM was" will indicate that "EXAM not found", as "EXAM" is *not* a file/subdirectory within itself, and "long listing" information about the subdirectory "was" is produced (due to the "-dl" option).

[9] The command "ls w*" is equivalent to "ls was weird welcome why wisdom", due to the expansion of the "*" metacharacter. The listing produced is:

```
weird
welcome
why
wisdom

was:
where
words
```

noting the contents of subdirectory "was" is displayed (in the absence of the "-d" option).

[10] The command "ls *h*" expands to "ls why", as "why" is the only "non-dot" file or subdirectory in "EXAM" that contains an "h", and UNIX verifies that the file exists by printing its name.

Problem 10

My solutions to the problem are listed below, but other answers may exist:

[1] To print your working directory, issue the command "pwd", which will yield a response "/your_system/your_login/EXAM".

[2] To change to the "was" subdirectory, issue the command "cd was".

[3] You are in directory "was", whose parent is "EXAM", and the proper command is "cp ../welcome hello" (noting ".." is an abbreviation for the parent of whatever directory you are currently working in).

[4] The proper command is

```
cat where hello >>../begin
```

noting the "cat where hello" portion combines files "where" and "hello", and, instead of printing them on the terminal, they are appended to file "EXAM/begin" (again, ".." is an abbreviation for the parent of "was") via the redirection ">>../begin".

[5] To return to the parent of "was", issue the command "cd ..".

[6] A possible command is

```
pr b* n* w[e-i]* was/*
```

noting the metacharacter expansions are equivalent to the *files* in "EXAM" beginning with "b", "n" or "w", and all files in subdirectory "was". If "*" was used rather than the string "b* n* w[e-i]*", we would have attempted to print the control information contained in "was", as "was" would have been one component of the metacharacter expansion.

[7] A possible solution is

```
grep -n one .w* w[e-i]* was/*w*
```

(again, trying to avoid "grepping" the directory "was" itself) and the resultant response from UNIX is:

```
.when:1:one
weird:7:allocate 2 days for a one-hour
why:1:If one advarces confidtly
was/words:2:cone
was/words:4:lone
was/words:6:none
```

[8] This task is easily accomplished by redirecting the output of the "word count" command, i.e.,

```
wc weird >count
```

[9] The output of the command "spell why" produces the list:

```
advarces
confidtly
cummun
directious
edevors
huors
sucess
```

missing the spelling error "unexeced". If spelling errors in "why" were corrected, the quotation would read:

```
If one advances confidently
in the direction of his dreams
and endeavors to live a
life which he has imagined.
he will meet with success
unexceeded in common hours.
```

Problem 11

My solutions (assuming all commands are issued within directory "EXAM") are:

[1] `man -T9700 banner at usend sleep`

[2] `opr -t xr weird`

[3] `cat weird welcome | opr`

[4] `opr -t xr -f nohole was/words`

[5] `pr *h* *1* | opr -p 2on1 -t xr`

Problem 12

My solutions (again assuming all commands are issued within directory "EXAM") are:

[1] My way of handling this problem is the sequence:

```
cat - welcome | pr
HI THERE
(CONTROL-D)
```

recalling that the "-" argument expects input from the terminal, and the (CONTROL-D) terminates that input.

[2] The solution is:

```
ls -d w* >WWW
```

noting the "-d" option is necessary to force the directory name "was" *itself* to be listed, rather than its contents.

[3] The command "nohup cat welcome" will cause a copy of the file "welcome" to be placed in a file "nohup.out", which UNIX created in directory "EXAM" (any output destined for the terminal will be redirected to file "nohup.out", if the "nohup" prefix is used).

[4] The multi-command line:

```
ls was; date; cat welcome
```

will provide the desired response

[5] The required command is:

```
spell w[e-i]* >tmp
```

where the subdirectory "was" is avoided by judicious use of metacharacters in the filename list "w[e-i]*".

[6] The solution to this problem requires the "at" command. as follows:

```
at now + 10minutes
date >777
(CONTROL-D)
at job number       [provided by UNIX]

at now + 2hours
opr welcome
(CONTROL-D)
at job number       [provided by UNIX]
```

recalling (CONTROL-D) terminates a particular "at job list". and UNIX provides a unique job number for each "at" command invoked.

[7] To "kill" a pending "at" job, use the following command:

```
atz number_of_job_to_be_run_in_2_hours
```

[8] To mail yourself the required message or file, try the following:

```
mail your_login_name
I love myself
.

mail your_login_name <was/words
```

[9] To read your mail, simply type "mail" with no arguments and respond to the queries by UNIX properly, i.e.,

```
From your_login_name date
{contents of file "was/words"}

? s junk
```

noting your response "s junk" will cause the mailed file headed by a postmark to be stored in file "junk". Then another message is then printed:

```
From your_login_name date
I love myself

? d
```

noting the "d" response destroys the message.

[10] The proper command is:

```
ls /usr/news | grep 2 >NEWS
```

noting "ls /usr/news" produces a list of *all* "news articles" available. "grep 2" accepts that list from the "pipe" and isolates those names containing the character "2". and those names are placed into file "NEWS" via redirection

Problem 13

My solutions are:

[1] You are in directory "EXAM", and the command:

```
cat .when
```

will yield a printout:

```
one
two
three
```

assuming your terminal does *not* respond to the "tab" character.

[2] The proper command is:

```
cd    [without a directory argument]
```

[3] The following sequence will accomplish the task:

```
e .profile
(character count of file read)
a
stty -tabs
.
w
(character count of buffer written)
q
```

[4] Hitting (CONTROL-D) will log you off and provide another "login sequence". This action allows ".profile" to be read, and instruct the terminal to react to "tab" characters.

[5] Now the command:

```
cat EXAM/.when
```

will result in a printout, in which "tab" characters are activated, i.e.,

```
one
t       w                o
        thr       ee
```

[6] This command forces the UNIX command prompt to be "READY ".

[7] The proper command is

```
cd EXAM
```

Problem 14

My solutions are:

[1] Login "abc" must have a subdirectory "rje" in "mode 777" created before the file transfer takes place. The required commands issued in directory "/b2/abc" are:

```
mkdir rje
chmod 777 rje
```

[2] The required "usend" command is:

```
usend -d ihuxb -u abc .when was/words
```

where "ihuxb" is the "destination code" for the UNIX-B system, and the "usend" command was issued in your "EXAM" directory.

[3] The received files are "/b2/abc/rje/.when" and "/b2/abc/rje/words", respectively.

Problem 15

The following scenario (all commands issued in directory "EXAM") is valid, *if* you've completed the previous questions *exactly* as instructed:

[1] The command:

```
chmod 200 was
```

will deny all permissions for subdirectory "was", except "write" permission to the owner of the directory.

[2] The command:

```
chmod 000 welcome why
```

will deny *all* permissions for the files "welcome" and "why".

[3] The command:

```
rm .*
```

will remove all "dot" files in "EXAM", noting you will be advised by UNIX that "." (the current directory "EXAM") is a directory and cannot be removed by "rm" without the "-r" option, and that ".." (the parent of "EXAM") cannot be removed from within the "child".

[4] The following command will change the name of "junk" to "welcome":

```
mv junk welcome
```

noting you will receive a "mode warning" from UNIX of the form "mv:welcome:0 mode", to which a "y" response will force the move. The new file "welcome" will possess the permissions of the old file "junk".

[5] In "EXAM", the following command will set up the "interactive deletion" process:

```
rm -ir .
```

noting "." (an abbreviation for the current directory "EXAM") must be explicitly designated. The interactive dialogue with UNIX is as follows:

```
./begin: y
./number: y
./welcome: y
./wisdom: y
./weird: y
./777: y
directory ./was: (CR)
./why: y
./count: y
./WWW: y
./nohup.out: y
./tmp: y
./NEWS: y
```

noting a "y" response removes the named file. The (CR) response to the entry request for directory "was" causes that directory to be skipped in the deletion process; had we typed a "y", entry to the directory would have been denied, due to its 200 mode, and the "rm" command would be exited.

[6] The final step is to change the mode of subdirectory "was", return to your "login" directory, and call for the elimination of "EXAM" and its contents, i.e.,

```
chmod 700 was
cd
rm -r EXAM
```

Problem 16

Noting that the RJE system on UNIX-A will be down for two days, we cannot utilize USEND or OPR on that machine. Therefore, my solution to the problems is to ship a CPIO archive of the three files to the UNIX-B system via the command issued in the unknown starting directory on the UNIX-A system:

```
echo one two three | cpio -oc >temp
nusend -d ihuxb -u xxx temp
```

The ECHO command pipes the names of the desired files to CPIO, which forms an archive with a portable ASCII header (as UNIX-A and UNIX-B are not running on the same computer equipment).

Now, we can enter the login "xxx" on the UNIX-B system, and issue the following commands

```
mv /b1/xxx/rje/temp .
```

which moves our archive file to the login directory "/b1/xxx" and leaves "rje" empty, as we found it.

Now, we can dearchive the file "temp," send file "one" to OPR and simultaneously remove it (via FIND primaries), and log off. The following commands should do the job:

```
cpio -ic <temp
find . -name one -exec opr -t xr {}\; -exec rm {} \;
```

This may not be a unique solution, but it was fun anyway.

CONGRATULATIONS TO ALL THE MASOCHISTS WHO COMPLETED THIS EXAM!!!!

KEW [illegible]

IH-55625-KEW-etj

[illegible]

K. E. Wendland