Specification of the NE text editor

by

Philip Hazel

Edition 2.0
March 2004

Contents

1. Introduction

NE is a text editor that was originally designed to run on a wide variety of machines, from large servers to personal workstations. In the past it ran on a number of operating systems; however, the current version supports only Unix-like systems.

The main use of NE is expected to be as an interactive screen editor. However, it can also function as a line-by-line editor, and it is programmable, so it can be run non-interactively as a text manipulation tool.

NE is a re-implementation of a previous editor that was called E, which in turn evolved from one called Zed and a number of predecessors that ran on IBM mainframes. The lineage can be traced back to some very early Cambridge text editors of the 1960s.

NE is a large program with many facilities. They are described in this document grouped by function, but first there are definitions of some terminology and a description of the areas in which there are differences between the various versions of the program. The chapter which follows describes how to use the screen editing features of NE, while subsequent chapters cover the many different commands available. Then there is detailed information for each different implementation, and finally there are keystroke and command summaries.

In many places in the text there are cross-references to particular NE commands. These are given simply as a command name in square brackets, for example [RMARGIN].

Experience with a number of other editors influenced the design of E and NE. Similar facilities are frequently encountered, and it is difficult to trace the origins of many of them. The operations on rectangles and some of the operations on single lines and groups of lines are taken from the Curlew editor implemented by the University of Newcastle-Upon-Tyne in the mid-1980s. Members of the Computer Laboratory and other computer users in Cambridge contributed many useful ideas and criticism to the design process over a number of years, up to the mid 1990s, when further major development ceased.

This edition of the manual was minimally revised in 2004 when the code was tidied up to make NE more easy to build on Unix-like systems. The original document was designed as an A5 booklet. This version is formatted for A4, and as a result, layout is not as nice as it could be, but time constraints prevented more work being done.

2. Terminology

Several terms which are used with particular meanings in the rest of this document are defined in this chapter. [March 2004: Some of these definitions seem ‘obvious’ now. Ten years ago, when this document was written, they were not so widely known in the user community.]

A computer screen indicates the point at which the next character will be written by a mark known as a *cursor*. This may be an underline-like shape or a vertical line, or it may be a solid blob. It may flash or be constant in intensity.

When text is being edited on the screen, the cursor position marks the point in the text at which changes are made. This is called the *current point* in the file. Any references in this document to the cursor position or moving the cursor should be understood as also referring to the current point. The term *current line* is also used to refer to the line of text containing the current point (that is, the line on which the cursor is placed). When NE is obeying commands (as opposed to accepting screen editing operations) the current point and the current line may change without reference to an actual cursor.

The word *scroll* is used to describe NE’s action in moving text on the screen. Blocks of lines can be scrolled upwards or downwards, and also to the left or to the right.

A *buffer* in NE is an area of main memory in which lines of text can be held for editing. A file which is to be edited is read into a buffer to be modified; subsequently the buffer’s contents may be written back to a file. NE is capable of handling more than one buffer at once.

A *keypress* is the action required to generate a single character from a keyboard. In many cases it involves pressing a single key – however, in some cases a second (and sometimes even a third) key must be held down while a ‘primary’ key is pressed. When such a keypress is referred to, the names of the keys are separated by a slash. For example, the keypress **ctrl/a** involves holding down the key marked CTRL while pressing the key marked A.

The character engraved on the key top is normally used when referring to individual keys. In one or two cases where this could be confusing (for example, backslash) the name of the character is used instead. The four keys with arrows on them that appear on many keyboards are referred to as **up**, **down**, **left** and **right**.

The word *keystroke* is used to refer to keyboard operations which are seen by NE as a single action but which may on some terminals require more than one keypress. For example, on some keyboards it is necessary to press the ESC key followed by the digit one (two keypresses) in order to generate the keystroke **function-1**, whereas on other keyboards a single keypress of a function key may generate this keystroke.

3. Screen editing

This chapter describes the screen editing facilities. They are sufficient for carrying out many simple interactive editing jobs. New users of NE are recommended to try out these facilities and become familiar with them before reading the rest of this document. It is assumed that all the default options are being used.

Editing operations that are activated by pressing special keys are described in terms of *logical control keystrokes*, whose names are shown in bold italic type, for example, ***scroll-up***. The actual keys used for these keystrokes may vary from keyboard to keyboard and operating system to operating system. Examples of the relationships between logical control keystrokes and actual keystrokes are often given in the main text of this manual, while complete lists are given in the system-dependent and terminal-dependent chapters near the end.

It is possible for the user to change the relationship between actual and logical control keystrokes in many cases [KEY]. The examples given in this manual assume the standard configuration.

As well as those logical keystrokes that have pre-determined functions, such as ***scroll-up***, there are 60 logical keystrokes whose effect is to obey a string of NE commands that is stored in a *function keystring* [FKEYSTRING]. These logical keystrokes have names of the form ***keystring-<n>***, where <*n*> is number between 1 and 60. When the keyboard contains function keys, these are usually bound to the corresponding logical keystroke, but they do not have to be. Function keys can be bound to pre-determined actions, and other control keys can be made to invoke function keystrings.

The common default actual keystroke is normally given with the first mention of each logical keystroke. This is the keystroke that will normally be bound to the logical keystroke by default, provided the keyboard supports it.

3.1 Getting started

To edit a file interactively using NE, enter the command

```
ne <file name>
```

The screen will be cleared and re-written with the first lines of the file, in the format described below. Any error in the command, for example, the naming of a non-existent file, provokes a system-dependent error message.

If the command name is not followed by a file name, NE is entered with no existing lines of text to edit, and the text area of the screen is empty. New text can be created using the editing operations described below, and when editing is complete, a file name can be given for the output (see the section entitled *Leaving NE* below). This is one way of creating a new file using NE. Alternatively, a new file can be created by using the **-to** keyword on the NE command line, without naming an existing file for editing. For example:

```
ne -to <new file name>
```

3.2 The screen display

An example of a typical screen display on entry to NE is shown below. Because of limitations of space on the page, it is shown as 40 columns wide and 10 lines deep, though screens (or windows) are at least 80 columns wide and 24 lines deep. Characters that would appear in inverse video on the screen are shown in boldface type.

```
----*----1----*----2----*----3----*----|
This is the first line of the file.
The second line of the file is a bit lon
The second line of the file is longer
than the width of the screen.
This is the last line of the file.
End of file

1    IRUA  test ---2----*----3----*----|
NE version 0.098/0.11 (22-Apr-1994)
```

The screen display consists of four sections. The top line is an inverse video line containing indications of column numbers. Every tenth column contains a digit indicating the decade – column 30 contains the digit 3 for example. At the halfway point in each decade there is an asterisk, while the rest of the positions are filled with minus signs. When NE is initialized, the last column in this separator line contains a vertical bar character. This marks the position of the first character beyond the *right-hand margin* about which more will be said later [RMARGIN].

Most of the rest of the screen is used to display lines from the file being edited. This is called the *text area*. For lines that are wider than the screen, only the initial part is shown, with the last character in inverse video to indicate that there are more characters in the line. The second text line in the example above is such a line. At the end of the file the text 'End of file' is displayed in inverse video, usually on a line by itself.

Below the data lines there is another inverse video separator line with column indicators, but also containing additional information. At the extreme left a number is often displayed. This is the number of the line on which the cursor is resting. Line numbers are counted from the start of the file when it is read; they do *not* change when lines are deleted or inserted into the edited text. Undisturbed lines retain their original number at all times, unless explicitly renumbered [RENUMBER]. If the cursor is on a newly-inserted line, blanks are displayed at the beginning of the lower separator line.

Three capital letters follow the line number – initially they are 'IRU'. These indicate the state of certain editing options, and are explained in detail later on. The first one indicates whether new characters will inserted into the text (I) or will overstrike it (O) [OVERSTRIKE], the second indicates whether portions of text that are cut out will replace what is in the cut buffer (R) or be appended to it (A) [CUTSTYLE], and the third indicates whether text searches are to be independent of case (U) or verbatim (V) [CASEMATCH]. A fourth letter (another 'A') is present if auto-alignment has been enabled [AUTOALIGN].

Following the indicator letters a file name is usually displayed. This is the name of the file to which the edited text will be written at the end of the editing session. In the simple case it is the name of the file which is being edited.

The final line of the screen is used to display messages as editing proceeds, and also for command input, as described later. Initially it contains an identification of the version of NE that is being used.

3.3 Refreshing the screen

When using NE on a timeshared system, the screen may sometimes get overwritten by extraneous messages, such as warnings from operators. When this happens, the keystroke ***refresh*** can be used to cause NE to re-write the entire screen. By default, ***refresh*** is bound to **ctrl/d**.

3.4 Moving about the file

When NE is entered without any options, the cursor is positioned at the top left-hand corner of the text being edited. It can be moved about the screen by pressing any of the four 'arrow' keys: ***up***, ***down***, ***left*** and ***right***. On keyboards containing real arrow keys these keystrokes do correspond to the actual arrow keys **up**, **down**, **left** and **right**; on other keyboards CTRL keystrokes are normally used.

Special actions take place if an attempt is made to move outside the text display area.

- Pressing *up* at the top of the area has no effect if the top line in the display is the first line of the file. Otherwise it causes NE to move backwards in the file by one or more lines [SET AUTOVSCROLL]; the text on the screen is scrolled down and new text is displayed at the top. The cursor is left on the line before the one it was previously on.
- Pressing *down* at the bottom of the display area causes NE to move forwards in the file by one or more lines [SET AUTOVSCROLL] unless the bottom line is the end-of-file line; the text on the screen is scrolled up and new text is displayed at the bottom. The cursor is left on the line following the one it was previously on.
- Pressing *left* at the left-hand edge of the screen causes the cursor to move to the end of the previous line, unless the screen has been scrolled right, in which case a leftwards scroll takes place. If this keystroke is used at the beginning of the top line on the screen, NE scrolls the screen to bring the new current line into view.
- Pressing *right* at the right-hand edge of the screen causes the screen to scroll to the right, that is, the lines are displayed starting from a character other than the first, enabling characters that were previously off the right-hand end to be displayed.

The arrows are not the only keystrokes for moving the current point around the file. The following additional horizontal movements are provided:

- The ***word-left*** keystroke moves the cursor to the start of the current word if it is in the middle of a word, or to the start of the previous word otherwise. Non-word characters are passed over. This keystroke is not associated with an actual keystroke by default.
- The ***word-right*** keystroke moves the cursor to the start of the next word. Non-word characters are passed over. This keystroke is not associated with an actual keystroke by default.
- The ***tab*** keystroke moves the cursor to the next tab position to the right. If the keyboard contains a key labelled TAB, it is normally used for the ***tab*** keystroke. Otherwise **ctrl/i** is used. Tab positions are set at every eighth column. The ***tab*** keystroke does *not* cause the insertion of tab characters into the file. If the right-hand edge of the screen is reached, the action is the same as for the ***right*** keystroke.
- If the ***previous-tab*** key is pressed (default **shift/tab** or **ctrl/tab**) the cursor moves to the previous tab position. If the left-hand edge of the screen is reached, the action is as for the ***left*** keystroke.
- The keys ***first-char*** and ***last-char*** (default **ctrl/left** and **ctrl/right**) cause the cursor to move, within the current line, to the first printing character on the screen or to just past the last printing character, respectively. If the line contains no printing characters, both these keystrokes move the cursor to the left of the screen. If there is a printing character in the final column on the screen, ***last-char*** puts the cursor under it, as it cannot put it beyond it.
- If ***screen-left*** is pressed (default **ctrl/backslash**), the cursor is moved to the left-hand edge of the screen.

Note that there is no difference between ***first-char*** and ***screen-left*** unless the current line begins with spaces. In this case ***first-char*** moves the cursor to just after the leading spaces, while ***screen-left*** moves it to the left-hand edge of the screen.

Additional vertical movements, which in general cause the screen to be re-written, are also provided. They are as follows:

- ***Scroll-up*** and ***scroll-down*** (defaults **shift/up** and **shift/down** or **ctrl/j** and **ctrl/k**) cause NE to move up or down the file by almost one screenful of lines. If the current point is near the start or end of the file the only effect may be to move the cursor to the first or last line, as appropriate. When one of these keystrokes occurs, if the cursor is on one of the lines that is still visible on the revised screen (that is, if it was originally near the bottom for ***scroll-down*** or near the top for ***scroll-up***) then it remains on the same text line afterwards, but on a different screen line. Otherwise the cursor stays on the same screen line. This ensures that scrolling down a file and then back up again (or *vice versa*) brings the cursor to the same place. The horizontal position of the cursor always remains unchanged.

- ***Scroll-top*** and ***scroll-bottom*** (default **ctrl/up** and **ctrl/down**) can be used to move to the beginning or end of the file respectively. The cursor ends up either on the first line or on the end-of-file line, as appropriate. Its horizontal position is not changed.
- The keystroke ***keystring-8*** has the same effect as ***scroll-bottom***, except that the cursor is also moved to the beginning of the end-of-file line. Similarly, the keystroke ***keystring-18*** moves the cursor to the start of the first line of the file.
- The keystroke ***keystring-58*** (default **ctrl/circumflex**) causes the screen to be re-displayed with the current line as the first line on the screen. The current point in the file is not altered [TOPLINE].
- The keystroke ***keystring-59*** (default **ctrl/underline**) causes NE to return to the region of the file where the last change was made. The 20 most recent areas of change are available via this keystroke. If the file has not been changed, this keystroke has no effect [BACK].

In addition to these keystrokes, there are a number of NE commands for moving about the file; these are described later [>, <, BACK, BF, CSD, CSL, F, M, N, P, PA, PB, PLL, PLR, TOPLINE].

3.5 Changing data on the screen

Typing any of the normal printing characters (or **space**) causes the character to be inserted into the line immediately before the cursor position. The remainder of the line is moved one position to the right, and so is the cursor [OVERSTRIKE]. This may have the effect of pushing a character off the right-hand edge of the screen, but though it is now invisible, it does not get lost. Note that, on most keyboards, holding down a key causes multiple insertions to occur.

If any characters are inserted into the end-of-file line, it is converted into a data line, and a new end-of-file line is created.

If a character is typed when the cursor is just beyond the right-hand margin, NE splits the current line at the nearest previous space character in the line. The position of the first character beyond the margin is indicated in the separator lines by a vertical bar. Characters after the splitting point are placed on a new line, for which room is made by scrolling the screen. For example, suppose the top of the screen is as follows, with the cursor at the end of the first line, immediately following the letter ‘l’:

```
----*----1----*----2----*----3----*----|
The quick brown foxes jumped over the l
The slow red badgers hopped about.
```

The next character to be typed by the user is beyond the right-hand margin. Suppose the letter ‘a’ is typed. The screen will then be changed to the following:

```
----*----1----*----2----*----3----*----|
The quick brown foxes jumped over the
la
The slow red badgers hopped about.
```

with the cursor after the letter ‘a’ on the second text line. The user can therefore type text continuously, leaving NE to split lines when they become full. This feature is known as *power typing*.

Automatic line splitting happens only when a data character is typed immediately after the right-hand margin. If, for example, ***right*** is used to move the cursor further past the margin to the right, then subsequent data characters are simply added to the line in the normal way.

The ***split-line*** key is used to request that a line be split at the current cursor position. By default, ***split-line*** is bound to the **return** key. Normally lines below the current one are scrolled down to make room, but if the current line is near the bottom of the screen, lines above it are scrolled up instead [SET SPLITSCROLLROW].

When a line is split, either as a result of the power typing feature, or because of an explicit use of *split-line*, the second part of the line becomes a new line, starting in column one. It is possible to arrange that such lines are automatically given the same indentation as the previous line. See the description of the **autoalign** command in the section entitled *Entering NE command lines* below.

The ***delete-previous*** and ***delete-here*** keystrokes provide two complementary ways of deleting individual characters. ***Delete-previous*** deletes the character immediately *before* the cursor and moves the cursor one place to the left, while ***delete-here*** deletes the character *at* the cursor and leaves the cursor position unchanged. In both cases the line is closed up to remove the space where the character was. Holding down either of these keys causes multiple deletions.

Most keyboards have a key labelled DELETE or DEL which often generates character 127. Unfortunately, conventions as to whether this key is used for ***delete-previous*** or ***delete-here*** vary from system to system. The default keystroke assignment for character 127 is ***delete-previous***. However, in some environments, the DELETE key may generate a control sequence rather than the single 127 value. For example, in some *xterm* windows, it generates the same code as **shift/f1**, which is treated as 'function key 21' by NE.

If ***delete-previous*** is pressed when the cursor is at the beginning of a line, it causes that line to be joined on to the end of the previous line. Pressing ***delete-previous*** at the beginning of a line is in effect a different logical keystroke, and this is given its own name, ***concatenate***, which can be bound to a different keystroke if required.

When concatenation occurs, the screen is scrolled up one line and a new line is displayed at the bottom. The cursor remains under the same character, in its new position. An erroneous ***concatenate*** can be cancelled by pressing ***split-line***, and an erroneous ***split-line*** can be cancelled by pressing ***concatenate***. For most terminals this means that the normal keystroke for immediately cancelling an erroneous data character also applies to the RETURN key, though cancellation may not be exact if the auto-alignment facility is being used [AUTOALIGN].

If ***concatenate*** is pressed on the end-of-file line, the only effect is to move the cursor to the end of the previous line.

The ***delete-left*** and ***delete-right*** keystokes (by default **ctrl/x** and **ctrl/v**) provide two complementary ways of deleting many characters on a line. The former deletes from the character immediately before the cursor to the start of the line, while the latter deletes from the character at the cursor to the end of the line.

The ***delete-to-word-left*** and ***delete-to-word-right*** keystrokes, which are not associated with any kepresses by default, provide other ways of deleting more than one character on a line.

The ***delete-to-word-left*** keystroke deletes characters to the left in the current line, starting at the character preceding the cursor position, and ending with the character at which a ***word-left*** cursor-moving operation would stop. In other words, it deletes back to the start of the previous word.

The ***delete-to-word-right*** keystroke deletes characters to the right in the current line, starting from the character at the cursor, and ending at the last character before the next position in which a ***word-right*** cursor-moving operation would stop. In other words, it deletes everthing from the current position to the start of the next word.

Note that, unlike the ***word-left*** and ***word-right*** keystrokes, these operations never move to a different current line. If issued at the start of end of a line (respectively), they have no effect. They do, however, operate when NE is reading a line of commands in screen mode.

The ***delete-line*** keystroke (by default **ctrl/u**) deletes all the characters on the current line, both to the left and to the right of the cursor, and in addition closes up the file vertically to remove the resulting blank line. The screen is scrolled up one line and a new line displayed at the bottom.

Four final keystrokes for operating on individual lines perform more specialized actions. They are of most use when operating on several lines at once (see *Multi-line editing* below, where examples of their use are given), but are described here because in the simple case they act on a single line.

Align-line (default **ctrl/a**) has the effect of *aligning* the current line with the cursor – the line is shifted to the right by inserting spaces at its start, or to the left by removing spaces at its start, until the first non-space character in the line is at the cursor position.

Align-previous (default **ctrl/z**) has the effect of aligning the current line with the previous line – the line is shifted to the right by inserting spaces at its start, or to the left by removing spaces at its start, until the first non-space character is in the same column as the first non-space character of the previous line. The cursor is moved to the alignment point.

Close-up (default **ctrl/c**) has the effect of closing up the line at the cursor position – if there are one or more space characters starting at the cursor position, these are deleted, and the remainder of the line is shifted left so that the first non-space following the cursor position is now at the cursor position. Note that ***close-up*** is equivalent to ***align-line*** if the cursor is at the left-hand edge of the screen.

Close-back (default **shift/delete**) has the effect of closing up the line leftwards, by deleting any spaces immediately before the cursor position, and shifting the remainder of the line to the left. The cursor remains on the same character as before, but moved to the left if any spaces were actually deleted.

3.6 Undeleting lines and characters

When characters or lines are deleted, NE does not discard the data immediately. Up to 100 deleted lines are remembered, in reverse order. Deleted characters are held in a packed form, and up to 64 consecutive ones can be held in one 'line' for this purpose.

The keystroke ***keystring-4*** causes NE to insert the most recently deleted thing back into the file [UNDELETE]. Undeleted characters are inserted back into the text at the current point, and NE remembers whether they were deleted forwards or backwards so that a sequence of deletes followed by a sequence of undeletes should put the text back as it was. Undeleted lines are treated as text to be inserted at the current point. Note that this is *not* a general 'undo' feature.

Text deleted with commands such as **e/a//b** is also added to the undelete stack, as is text cut out with the **delete** command (the **ctrl/q** keystroke). Text cut or copied to the cut buffer is *not* added to the undelete stack.

3.7 Overstriking characters

In its default configuration, NE always inserts newly-typed characters into existing text, opening up the line to make room for them. This is the safest mode of operation, because it lessens the possibility of accidentally deleting characters. However, when the text that is being edited is in a fixed format, it is sometimes more convenient to operate in *overstrike mode*, where newly-typed characters overstrike (i.e. replace) existing characters on the screen.

NE can be switched from insert mode to overstrike mode, and *vice versa*, by pressing ***keystring-60***. This is normally bound to the keystroke **ctrl/o** by default. The first of the three capital letters in the bottom separator line shows which of these two modes NE is in at any one time, displaying 'I' in insert mode and 'O' in overstrike mode.

3.8 Editing wide lines

If the file being edited contains lines that are wider than the screen, there are some differences in the behaviour of NE. When any such line is displayed, the last character on the screen is shown in inverse video, to indicate that it is not in fact the last character of the line.

NE operates with a *right-hand margin*, which initially is set at one less than the screen width, and the first position beyond it is indicated in the separator lines by a vertical bar character.

The margin controls where the power typing feature comes into effect, but it does not prevent editing from taking place to the right of it.

To disable automatic line splitting, the keystroke ***keystring-10*** is used. The vertical bar in the separator lines changes to a backslash. This indicates that the margin, while still at the same position, has been disabled. Pressing ***keystring-10*** a second time re-enables the margin.

A number of keystrokes may cause changes of the horizontal position of the 'window' on the file. When a line is split by the ***split-line*** keystroke, there is a movement back to column one if necessary. When lines are concatenated by pressing ***concatenate*** (usually implemented as ***delete-previous*** when the cursor is in column one), there may be a movement to the right if the previous line is longer than the screen width.

Movement to the left or right can be explicitly requested by means of the keystrokes ***scroll-left*** and ***scroll-right*** (default **shift/left** or **ctrl/h** and **shift/right** or **ctrl/l**). In these cases the cursor is left in the same text column if it is still visible in the new window. For example, if the screen is showing columns 1– 80 and the cursor is in column 53, after ***scroll-right*** it is still in column 53, though this is now in column 48 of the screen. If the old text column is not visible in the new window, the cursor is placed as near to it as possible.

The keystrokes ***first-char***, ***last-char*** and ***screen-left*** operate only on the text that is displayed on the screen, and never cause any movement of the viewing window. ***First-char*** moves to the first visible non-space character of the current line, while ***screen-left*** moves to the left of the screen. ***Last-char*** moves just past the last visible non-space character of the current line, unless there is a non-space character at the right-most column on the screen, in which case it puts the cursor under it.

There are also two keystrokes which move the cursor to the true beginning or end of the line, causing a horizontal scroll if necessary. These are ***start-line*** and ***end-line***, defined by default to be **shift/ctrl/left** and **shift/ctrl/right**. Not all keyboards support those keystrokes, so in addition two function keystrings are defined to have the same effect. These are ***keystring-6*** and ***keystring-16***. When the line is not wider than the screen, they are synonymous with ***first-char*** and ***last-char***.

Movement up and down the file by means of the arrow keys, ***scroll-up***, ***scroll-down***, ***scroll-top***, or ***scroll-bottom*** does not affect the horizontal position of the window. However, when such movement occurs as a result of a command [F, BF, M, N, P, BACK], there may be a change of window position. The function keystrokes ***keystring-8*** and ***keystring-18*** (which move to the end and the start of the file respectively) operate via the command mechanism, and move the cursor to the start of the appropriate line.

The right-hand margin is also used by the **format** command. Its value may be changed by the **rmargin** command. Details of these facilities are given later.

3.9 Editing non-printing characters

Files which contain non-printing characters can be processed by NE. On the screen, non-printing character is displayed as a question mark [EIGHTBIT]. (See the **?** command for a way of displaying the hex codes of non-printing characters.) Some implementations of NE allow non-printing characters to be input directly from the keyboard during screen editing, while others may not. They can, however, always be inserted into the file using the **a**, **b**, **e**, or **iline** commands which are described later on.

3.10 Multi-line editing

Certain keystrokes for editing individual lines can be made to operate (independently) on each of a sequence of lines. The following actions accomplish this:

(1) Place the cursor anywhere on either the first or the last line of the sequence and press the ***mark-line*** key (default **ctrl/b**). The message 'Bulk line operation started' appears in the message area at the bottom of the screen, and the character at the cursor position is displayed in inverse video [MARK LINE].

(2) Move the cursor to the line at the other end of the sequence. Then press one of the line-editing keys listed below. The operation is carried out on the current line, the line marked by ***mark-line***, and all the lines in between.

The operations that can be performed on many lines at once in this way are as follows:

- ***Align-line*** (default **ctrl/a**): All the lines in the block are aligned horizontally with the position of the cursor when ***align-line*** is pressed. That is, spaces are added or deleted at the start of the lines as necessary, until the first non-space in each line is in the same column as the cursor [ALIGN].
- ***Align-previous*** (default **ctrl/z**): All the lines in the block are aligned horizontally with the line that precedes the block. The horizontal position of the cursor is not relevant; it gets set to the alignment position.
- ***Close-up*** (default **ctrl/c**): For each line in the block, if one or more spaces are present at the cursor column, they are removed and the line is closed up [CLOSEUP].
- ***Close-back*** (default **shift/delete**): For each line in the block, if one or more spaces precede the cursor column, they are removed, and the line is closed up [CLOSEBACK].
- ***Delete-line*** (default **ctrl/u**): All the lines in the block are deleted, and the file is closed up vertically [DLINE].
- ***Delete-right*** (default **ctrl/v**): For each line in the block, all characters at and to the right of the cursor position are deleted [DRIGHT].
- ***Delete-left*** (default **ctrl/x** or **ctrl/delete** or **ctrl/backspace**): For each line in the block, all characters before the cursor position are deleted, and the line is closed up. The cursor ends up in column 1 and there may be a movement of the viewing window to the left [DLEFT].

When one of these operations is performed, the ‘mark’ which was set by pressing ***mark-line*** is automatically deleted.

Sometimes is is necessary to carry out a number of these functions on the same group of lines. In this case, ***mark-line*** should be pressed *twice* before starting the operations. After the second press, the word ‘operation’ in the message at the bottom of the screen changes to ‘operations’, and the mark is no longer automatically deleted when any action is performed on the line group. When all the operations are done, ***mark-line*** must be pressed a third time to remove the mark, except when the final operation is ‘delete’ (***delete-line***), which always automatically removes the mark.

If ***mark-line*** is pressed in error, the bulk line operation can be abandoned by pressing ***mark-line*** twice more.

As an example of the use of the multi-line editing facilities, suppose the screen contained the following lines:

```
This is some messy unprocessed data
which needs tidying up a bit. We only
want to retain the second two
columns, and we want them tidy.
123.45     67.98      100.00  xyz
999.00   45.87        456.78     pqr
456.86        88.23   854.67        abc
```

To delete the four lines of text, press ***mark-line*** with the cursor on the first line, move to the fourth line, and press ***delete-line***.

If ***mark-line*** is now pressed with the cursor anywhere on the new first line, and then ***delete-left*** (delete to left of cursor) is pressed after moving the cursor to just after ‘86’ on the last line, the entire first column of figures is deleted and the lines look as follows:

```
     67.98      100.00  xyz
   45.87        456.78     pqr
        88.23   854.67        abc
```

To align all the lines, press ***mark-line*** (the cursor being still on the last line), then move to column 5, say, on the top line and press ***align-line***. The lines become:

```
67.98       100.00  xyz
45.87         456.78     pqr
88.23   854.67        abc
```

To tidy up the messy second column, press ***mark-line*** (the cursor being now on the top line), then move to the bottom line at the start of ‘854’ and press ***close-up***. The lines are now:

```
67.98   100.00  xyz
45.87   456.78     pqr
88.23   854.67        abc
```

Finally, to remove everything except the columns of numbers, press ***mark-line***, then move to the top line just after ‘100.00’ and press ***delete-right***. The result is:

```
67.98   100.00
45.87   456.78
88.23   854.67
```

Certain types of more complicated editing operation, such as the deletion of the *second* column of a table, cannot be performed using the multi-line actions described in this section. For such operations rectangular blocks must be used (see below).

3.11 Cutting, pasting and block deletion

It is often necessary to move text from one position in a file to another, or to make a copy of some text at a different point in the file. NE provides these facilities using a *cut and paste* model. Text is identified in the file, and then either *cut* or *copied* from the file into the *cut buffer*. The cursor is then moved to the point where insertion is to occur, and the text is *pasted* into the file. There is only one cut buffer, and successive cut or copy operations destroy any text that was previously in it [CUTSTYLE, DCUT].

The word ‘paste’ is perhaps a little misleading in this context because it has connotations of overlaying existing text. In NE, a pasting operation *inserts* text into the file; it does not destroy any existing text, even if NE is operating in overstrike mode, as this mode applies only to newly-typed characters.

If an attempt is made to leave NE while there is text in the cut buffer that has never been pasted, a warning is given, and the user is prompted for permission to proceed. A prompt is also issued if text in the cut buffer that has never been pasted is about to be overwritten by new text.

The operation of deleting text from the file is provided in a similar manner to cutting or copying. However, in this case, the portion of the file that has been marked is discarded [UNDELETE].

When marking text for cutting, copying, or deletion, NE regards the end of the text as being just before the cursor position. Therefore, when indicating the beginning of a block, the cursor must be placed *on* the first character, but when indicating the end of a block, it must be placed *after* the last character.

Two different views of text are supported by the cut, paste and deletion operations. It can either be considered as a stream of characters, with a notional ‘newline’ between lines, or it can be considered as a rectangular array. These different approaches are described separately in the following two sections.

3.12 Text blocks

A text block is a piece of text viewed as a stream of characters, with notional ‘newline’ characters between lines. To cut, copy or delete such a block from the file being edited, the following actions are required:

(1) Move the cursor to either:

  (a) The first character of the block; *or*

    (b) Just beyond the last character of the block.

(2) Press ***mark-text*** (default **ctrl/t**). The message ‘Text block started’ appears in the message line at the bottom of the screen, and the character at the cursor position is displayed in inverse video. If ***mark-text*** is pressed in error, the text block operation can be abandoned by pressing ***mark-text*** again [MARK TEXT].

(3) Move the cursor to the other end of the block and press one of:

    (a) ***Cut-delete*** (default **ctrl/w**) to cut the text out of the file and place it in the cut buffer [CUT];

    (b) ***Cut-copy*** (default **ctrl/e**) to make a copy of the text from the file in the cut buffer [COPY];

    (c) ***Delete-marked*** (default **ctrl/q**) to delete the text from the file [DMARKED].

    While moving the cursor to the other end of the block, other editing operations may be carried out.

After a cut or copy operation (a copy of) the text is in the cut buffer, replacing whatever was there previously, whether it was a text block or a rectangle [CUTSTYLE].

To insert the contents of the cut buffer into the file, move the cursor to just after the insertion position and press ***paste*** (default **ctrl/p**) [PASTE]. The contents of the cut buffer are not destroyed by this, and therefore the same text can be inserted many times.

As an example of how to use text blocks, consider the problem of moving a sentence about in a paragraph of text. Suppose the top of the screen contained the following lines:

```
----*----1----*----2----*----3----*----|
The successful candidate will be
expected to reside in Cambridge and to
undertake approximately twelve hours a
week of teaching. Candidates should be
under the age of thirty-three on 1
October 1987. The salary will be
```

and that it is desired to reverse the order of the first two sentences by cutting out the second and inserting it before the first. Placing the cursor under the first letter of ‘Candidates’ and pressing ***mark-text*** begins the operation. The cursor is then moved *after* the space at the end of the sentence (that is, to the start of ‘The’ at the start of the third sentence) and ***cut-delete*** is pressed. The screen then looks as follows:

```
----*----1----*----2----*----3----*----|
The successful candidate will be
expected to reside in Cambridge and to
undertake approximately twelve hours a
week of teaching. The salary will be
```

Notice that the remainder of the sixth line has been joined to the remainder of the fourth line. This is in accordance with the view of the text as a linear stream of characters containing ‘newlines’. To insert (a copy of) the cut buffer, the cursor is now placed at the start of the text and ***paste*** is pressed. The screen becomes

```
----*----1----*----2----*----3----*----|
Candidates should be
under the age of thirty-three on 1
October 1987. The successful candidate w
expected to reside in Cambridge and to
undertake approximately twelve hours a
week of teaching. The salary will be
```

The newline after ‘be’ is reproduced in the inserted text, but the insertion has caused the third line to become wider than the screen. Since the cursor is left under the word ‘The’ it is a simple matter to press ***split-line*** to split the line and obtain

```
----*----1----*----2----*----3----*----|
Candidates should be
under the age of thirty-three on 1
October 1987.
The successful candidate will be
expected to reside in Cambridge and to
undertake approximately twelve hours a
week of teaching. The salary will be
```

Automatic tidying up of paragraphs after editing of this kind can be achieved using NE’s formatting facilities, which are described in a later section [FORMAT].

The text block facility can of course be used to move, copy or delete complete lines of text. In this case, the start of the block is the first character of the first line involved, while the end of the block is the first character of the line *after* the last line involved. If the end of the last line is used instead, the final ‘newline’ is not included in the block.

3.13 Rectangular blocks

For some operations it is convenient to regard a file of text as a rectangular array of characters. NE provides cutting, copying and deletion facilities for rectangles, and the ability to insert a rectangle of spaces into a file.

To cut or copy a rectangular block from the file being edited, the following actions are required:

(1) Move the cursor to any of the four corners of the rectangle. For left-hand corners the cursor should be *on* the first character inside the rectangle, while for right-hand corners it should be *after* the last character in the rectangle.

(2) Press ***mark-rectangle*** (default **ctrl/r**). The message ‘Rectangular block started’ appears in the message line at the bottom of the screen, and the character at the cursor position is displayed in inverse video. If ***mark-rectangle*** is pressed in error, the rectangular block operation can be abandoned by pressing ***mark-rectangle*** again [MARK RECTANGLE].

(3) Move the cursor to the opposite corner of the rectangle and press one of:

  (a) ***Cut-delete*** to cut the rectangle out of the file and into the cut buffer [CUT];

  (b) ***Cut-copy*** to make a copy of the rectangle in the cut buffer [COPY];

  (c) ***Delete-marked*** to delete the rectangle from the file [DMARKED].

  While moving the cursor to the other corner of the rectangle, other editing operations may be carried out.

When a rectangle is cut or deleted from a file, all the lines involved are closed up by an equal amount. After a cut or copy operation (a copy of) the rectangle is in the cut buffer, replacing whatever was there previously, whether it was a rectangle or a text block [CUTSTYLE].

To insert the contents of the cut buffer into the file, move the cursor to where the top left-hand corner of the rectangle is to be positioned, and press ***paste***. The contents of the cut buffer are not destroyed by this, and therefore the same rectangle can be inserted many times.

The insertion of a rectangle affects the current line and a number of lines below it, depending on the number of lines in the rectangular block. Each of the relevant lines is ‘opened up’ at the insertion point by the width of the rectangle, and the appropriate line of the rectangle is then inserted.

As an example of the use of rectangles, consider the problem of re-arranging the columns in a table. Suppose the lines on the screen are:

```
----*----1----*----2----*----3----*----|
First      Second      Third
123.45      76.99      88.23
999.00       8.4       45.06
  2.33        -          -
```

To interchange the second and third columns, the cursor is placed on the ‘S’ of ‘Second’ and ***mark-rectangle*** pressed. Then the cursor is moved to the start of ‘Third’ and down to the fourth line, and ***cut-delete*** is pressed. The second column is cut out and the screen looks as follows:

```
----*----1----*----2----*----3----*----|
First      Third
123.45     88.23
999.00     45.06
  2.33       -
```

Now the cursor is moved to the top line, several columns past the word ‘Third’, and ***paste*** is pressed. The result is:

```
----*----1----*----2----*----3----*----|
First      Third     Second
123.45     88.23      76.99
999.00     45.06       8.4
  2.33       -          -
```

Mark-rectangle may be followed by ***rectangle-spaces*** (default **ctrl/s**) instead of ***cut-delete***, ***cut-copy*** or ***delete-marked***. This has the effect of inserting a rectangle of spaces into the file. The operation is carried out by placing the cursor at one corner and pressing ***mark-rectangle***, then moving to the opposite corner and pressing ***rectangle-spaces***. The effect is to insert the same number of spaces into each line at the column marked by the left-hand side of the rectangle so that the character that was previously in that column is now in the first column to the right of the marked rectangle.

Suppose that the lines in the above example were to be indented by five characters. ***Align-line*** cannot be used because it would mis-align the third line of numbers. The cursor is placed at the start of the first line, and ***mark-rectangle*** is pressed. Then the cursor is moved to column six on the last line and ***rectangle-spaces*** is pressed. The result is as follows:

```
----*----1----*----2----*----3----*----|
     First      Third     Second
     123.45     88.23      76.99
     999.00     45.06       8.4
       2.33       -          -
```

3.14 Re-formatting paragraphs

When a paragraph of text is edited it often becomes untidy, in the sense that the lengths of the lines are very variable, and some lines may have become longer than the screen width. Re-formatting, so that each line contains the maximum number of words, is achieved by pressing ***keystring-20***, having previously placed the cursor on the first line to be affected. This need not necessarily be the first line of the paragraph [FORMAT].

Effectively, the lines from the current line to the end of the paragraph are joined into one long line, and the result is then split up so that no line exceeds the right-hand margin [RMARGIN]. Splitting always takes place just after a space character, unless there are no preceding spaces in the line, in which case the split happens exactly at the margin. Disabling the right-hand margin (***keystring-10***) does not affect formatting; the same margin value is still used.

A blank line or a line beginning with a space marks the end of a paragraph and the cursor is left at the beginning of the line following the paragraph that has been formatted. Thus ***keystring-20*** can be pressed several times in succession in order to format several paragraphs.

If the current line is empty when ***keystring-20*** is pressed, the only effect is to move the cursor to the start of the following line. Only non-empty lines are recognized as being part of a paragraph, and so blank lines are preserved by the formatting process.

It is possible to change the rules for the recognition of the beginnings and ends of paragraphs [BEGINPAR, ENDPAR]. This can be useful when editing text that contains, for example, control statements for a text formatting program.

3.15 Help information

Some information about the various keystrokes used by NE is available from within an NE session. Pressing ***keystring-9*** (normally implemented as function key 9 on any keyboard that has function keys) generates a display which lists the current actions of the keystrokes [SHOW KEYS]. The display is in three parts:

(1) The 'control' keystrokes. These are those that are usually generated by holding down the CTRL key and pressing another key with it.

(2) The 'extra' keystrokes. These are those that are usually generated by dedicated keys on the keyboard, such as the DELETE key.

(3) The function keystrokes. These keystrokes are those that are normally generated by function keys on the keyboard, but in some cases are generated by pressing the ESC key followed by another key. Pressing ***keystring-19*** generates a display which lists the actions of the various 'function' keystrokes only.

    Control and function keystrokes that are unset are omitted from the displays. When either ***keystring-9*** or ***keystring-19*** is pressed, the bottom separator line is moved up, and the display appears at the bottom of the screen. For ***keystring-9***, the total display is too long to fit on a typical screen. NE therefore pauses between parts and outputs the message:

    ```
    Press RETURN to continue
    ```

    The default actions of the function keystrokes are in fact to obey particular NE *commands*, that is, they are associated with appropriate function keystrings. The use of commands is described later, but many of the operations shown (such as 'format') should be understandable by someone who has read only this far.

At the end of the display there is a line containing

```
NE>
```

This is a command prompt from NE – the use of commands is covered in the next section. Pressing ***enter*** at this point causes the information display to disappear and the previous contents of the screen to be restored. On all current versions, ***enter*** is implemented as the RETURN key.

3.16 Entering NE command lines

The keystroke ***read-command*** (default **ctrl/g**) is a request to enter a line of NE commands. The character at the cursor position is re-written in inverse video, and the cursor moves out of the text display area and into the command entry line at the bottom of the screen, where the prompt '`NE>`' is shown.

The user must now type a line of commands, terminated by ***enter*** (normally the RETURN key). If the line is empty, NE simply reverts to screen editing. If mistakes are made while entering commands, the line can be edited using the cursor keys, ***delete-previous***, ***delete-here***, ***delete-left***, ***delete-right***, ***delete-line***, ***delete-to-word-left***, ***delete-to-word-right***, ***word-left***, and ***word-right*** before pressing ***enter*** to cause it to be obeyed. If a mistake is detected by NE, an error message is given and another prompt is issued. The previous line of commands can be recalled for editing by means of the ***scroll-up*** key.

The following subset of commands may prove useful to users who are just starting to learn how to use NE. For more details, and for a fuller description of the complete set of commands, see the chapters which follow. Note that the case of letters in command names is not significant.

- **F** /<*string*>/ causes NE to search forwards through the file from the current position until it finds the given character string, which should not contain the character '/'. If the string contains any letters, it does not matter whether they are in upper or lower case (capital or small); NE treats different versions of the same letter as identical in simple searches such as this. Once an **f** command has been obeyed, the same search can be repeated simply by pressing the ***keystring-7*** key. NE can be made to search backwards by using the **bf** command instead of **f**; this is repeated by means of the ***keystring-17*** key. If NE fails to find the string it is searching for, the current position is unchanged.

- **Ge** /<*string1*>/ /<*string2*>/ is a request to *globally exchange* the first string for the second. NE searches forwards in the file until it finds the string, exactly as for the **f** command. When it finds an occurrence, it updates the screen to show the text which was found, in inverse video, and outputs the prompt

  ```
  Change, Skip, Once, Last, All, Finish, Quit
                                            or Error?
  ```

  The user must type the initial letter of one of the displayed words, followed by ***enter***. 'C' (change) causes NE to make the change, then move on to the next occurrence of the string, while 'S' (skip) causes it *not* to make the change, but still to move on to the next occurrence. 'O' (once) and 'L' (last) both make the change, then terminate the command; the difference between them is that 'O' restores the current point to where it was at the start of the command, while 'L' leaves it after the final change. 'A' (all) makes the change, and then continues through the file, changing all occurrences without further prompting.

  'F' (finish), 'Q' (quit) and 'E' (error) all terminate the command without making the change. The difference between them is that 'E' forces an error, which causes NE to abandon any further commands and prompt for a new command line, while after 'F' or 'E', the current point is restored to where it was at the start of the command, but after 'Q' it remains at the point of last match.

  A number of successive replies to the prompt can be given all at once as a string of letters. For example, a reply of

  ```
  ccsccc
  ```

  specifies that the current and next occurence of the sought-for string are to be changed, then one occurrence is to be skipped, and then the next three are to be changed. Digits can be used within the string to save typing. The following is an equivalent reply:

  ```
  2cs3c
  ```

  In addition to the **ge** command, which exchanges one string for another, there are also commands called **ga** and **gb** which operate in the same way, but which cause the second string to be inserted after or before occurrences of the first string, respectively.

  NE contains facilities for more complicated kinds of change, including the use of 'wild card' characters in strings. The chapters entitled *Context matching*, *Character string insertions*, and *Global changes* contain further details.

- **Load** <*file name*> causes NE to load a new file for editing. If the old file has been changed but not saved (see below), the following is output:

  ```
  The contents of buffer 0 have not been saved.
  Continue with LOAD (Y/N)?
  ```

  A reply of **y** or **yes** causes the current editing text to be lost; a reply of **n** or **no** indicates that a mistake has been made. In this case, NE issues a further command prompt.

- **M** <*number*> causes NE to make the line with the given number the current line (**m** is an abbreviation of 'move'). For example,

  ```
  m1234
  ```

  makes line 1234 the current line. Remember that lines are counted as they are read from the original file, and the numbers do not change as lines are edited [RENUMBER]. If the specified line has been deleted, an error occurs and the current line does not change. A line number of zero is taken to mean the start of the file, and an asterisk may be given instead of a line number to move to the end of the file.

- <*Number*> **n** causes NE to advance through the file by the given number of lines (**n** stands for 'next'). For example, after obeying

  ```
  300N
  ```

  the current line is 300 lines further into the file than before.

- <*Number*> **p** causes NE to move backwards through the file by the given number of lines (**p** stands for 'previous').

- **Rmargin** <*n*> specifies a new right-hand margin value. For example, if text is being input and the maximum line length that is wanted is 60 characters, then

  ```
  rmargin 60
  ```

  is appropriate. After this command has been obeyed, the separator lines show a vertical bar in column 61, and any character typed in this column causes the power typing feature to come into effect. In addition, the formatting keystroke (***keystring-20***) now makes paragraphs whose maximum width is 60 characters.

- **Save** causes the current text that is being edited to be written back to the disc file. Before actually writing to the file, NE outputs the prompt

  ```
  Write to <buffer title> (Y/N/TO filename)?
  ```

  to ask for confirmation. A reply of **y** or **yes** allows the writing to go ahead; a reply of **n** or **no** indicates that a mistake has been made, and NE issues a new command prompt. A reply consisting of the word 'to' followed by further text is taken as supplying an alternative file name. When a new name is given, it becomes the default name for future **save** operations and for writing the file at the end of editing.

  If the reply is not in one of the above forms, NE outputs an error message and displays the prompting line again.

  Save is one way of taking safety copies of a file during a long editing session. It can also be used in conjunction with **load** to finish editing one file and start on another without leaving NE.

- **Show wordcount** causes NE to display information about the text it is editing. The numbers of lines, words, and characters are displayed; words in this context are sequences of characters separated by spaces or the ends of lines. The information is displayed at the bottom of the screen, which scrolls up to accommodate it, and a further command prompt is then given. Pressing the RETURN key at this point returns to screen editing.

- **Show commands** causes NE to display a list of all its command words.

- **Stop** causes NE to cease processing immediately, and exit. Any changes to the file being edited are lost. An error code is passed to the operating system.

If a command line (other than one containing **stop**) is successfully obeyed, NE returns immediately to screen editing without any further action from the user. If, on the other hand, an error message is output, NE issues a prompt for a further line of commands.

3.17 Multi-buffer editing

It is possible in NE to edit more than one file at once, switching between two or more buffers as necessary. Details of this facility is given in the chapter entitled *Buffer Handling*.

3.18 Leaving NE

When editing of the file is complete, ***keystring-3*** should be pressed [w].

The first thing that NE does when preparing to end an editing session is to check whether any text has been cut into the cut buffer, but never actually pasted into the file. If this is the case then the prompt

```
The contents of the cut buffer have not been pasted.
Continue with W command (Y/N)?
```

is output as a warning. (The reference to the **w** command occurs because the ***keystring-3*** key works by issuing that command.) If the user replies **y** or **yes** then NE proceeds, and the contents of the cut buffer are lost. If the reply is **n** or **no** then NE stops what it is doing and issues its standard command line prompt, '`NE>`'. Pressing ***enter*** at this point returns NE to screen editing.

Next, NE considers the file that has been edited. If no changes have been made to it, NE outputs the message

```
No changes made to <buffer title>
```

and exits. Otherwise, it outputs the prompt line

```
Write to <buffer title>? (Y/N/TO filename/Discard/STOP)
```

The user must give one of the five possible replies – a null or erroneous reply causes an error message to be output, followed by a repeat of the prompt. The effects of the replies are as follows:

- A reply of **y** or **yes** causes NE to write the edited text to the file whose name is shown, and then to exit.
- A reply of **n** or **no** indicates that the user has made a mistake. NE outputs its prompt line ('`NE>`') so that the user can tell it what to do next. Some possible responses to this are described below.
- A reply of the form **to** <*filename*> is a request to write the contents of the buffer to an alternative file. NE attempts to open this file and write the text to it instead of to the file named in the prompt.
- A reply of **d** or **discard** causes NE not to write the text to the file, but to carry on as if it had. In the simple case of editing only one file, it exits without error.
- A reply of **stop** (no abbreviation) causes NE to stop processing immediately, without doing anything further. It exits with an error code.

If there is an error while attempting to open or write to the file, or if the user replies **n** to the prompt, NE does not exit, but prompts for a line of user commands. Full details of NE's commands are given in later chapters, but a short list of possible responses is given here for the benefit of new users of NE. The command line should be terminated by ***enter***.

- To exit from NE (with an error code), type the command **stop**. The edited text which NE is holding in main memory is then abandoned.
- To attempt to output the edited text again (for example, if a file name was mis-spelled previously, causing an error to occur), type the command **w** (windup). NE then issues the 'Write to' prompt again.
- Entering an empty command line causes NE to restart screen editing.

3.19 Summary

This chapter has described the screen editing facilities of NE, and a subset of the more common commands, using all the default option settings. The facilities covered are sufficient for many straightforward editing jobs. The chapters which follow cover NE's command system in detail. This extends the power of the editor for more complicated situations. Note that there is a summary of the logical keystrokes near the end of this document, and that details of the correspondences between logical control keystrokes and actual keypresses are given in the device-specific chapters.

4. The NE command

The NE command takes the following form:

```
ne [<options>] [<file names>]
```

Without any options or file names, NE starts up with an empty buffer and no pre-set output file. Up to 50 file names can be given; each is read into a separate buffer. The keyword **-from** can optionally precede the list of file names, and the file names, provided none of them starts with a hyphen, can precede the options. Thus, for example:

```
ne /some/file -opt m1234
```

is a valid NE command line. A filename that begins with a hyphen must immediately follow the keyword **-from**.

If any of the input files does not exist, NE generates an error message and exits. It does not create a new file under these circumstances. To create a new file, a command such as

```
ne -to <filename>
```

can be used. Alternatively, the **ne** command can be given with no file names at all. This enters NE and sets up an empty editing buffer; a file name can be supplied when the buffer is to be written out.

The command line options are as follows:

-binary or **-b** invokes the special facility for editing binary files which is described in section 4.3 below.

-from may optionally precede the list of input files.

-help or **-h** requests a display of the syntax of the **ne** command. The main part of NE is not entered.

-id requests a display of the current version number of NE. The main part of NE is not entered.

-line requests that NE operate in line-by-line mode (as opposed to screen mode).

-noinit suppresses the use of any initializing commands. Normally, NE looks for a file whose name is specified in the NERC environment variable. If this is not set, it looks for a file called **.nerc** in the caller's home directory. The file contains NE commands that are obeyed at the start of every run. If NE is started with the **-noinit** option, the initializing action is bypassed.

-notabs disables any special handling of tab characters, and causes them to be treated as data characters. See section 4.2 for further details.

The **-opt** keyword on the **ne** command line is used to supply one or more commands to be obeyed at the start of editing. This can be useful, for example, for moving to a particular point in the file before displaying a screenful of data over a slow communications line. It can also be used to supply a short 'script' of non-interactive editing commands without using file to store them in.

-readonly or **-r** causes the first file that is being edited to be loaded into a read-only buffer [READONLY]. Any attempt to alter the contents is faulted.

The **-stream** option is concerned with the editing of files that are too large to fit into main memory, and its use is described in chapter 21 (*Editing large files*).

-tabs, **-tabin**, and **-tabout** control the handling of tab characters in files that are being edited. Details are given in section 4.2.

If only a single input file name is given, that file is edited and written back to a file of the same name. If a file name is given following the **-to** keyword, the result of editing the first file is written to the **-to** file at the end of editing. (These are defaults; the file name can be changed from within NE by appropriate commands.)

Up to fifty input file names may be given; each is loaded into a separate editing buffer. See chapter 19 (*Buffer handling*) for further details of multiple buffers. The **-to** option, if present, applies to the first named input file only. The default for the other files is to update them in place.

The **-ver** keyword can be used to direct verification and error messages to a specific file. If it is not specified, such messages are sent to the standard output (except when filtering, when the standard error is used).

The **-with** keyword can be used to specify the input file containing editing commands. If it is not specified, commands are read from the standard input (except when filtering – when the standard input contains the text to be edited.).

NE is initialized in interactive mode, unless the **-with** or **-ver** keywords are given or NE is being run as a filter. In interactive mode, if the terminal is suitable for screen editing, screen mode is also selected, unless **-line** is present. Otherwise interactive line-by-line mode is selected.

It is possible to specify a single hyphen as the file name for the **-with** and **-ver** command line options, as well as for **-from** and **-to**. The effect of a hyphen with **-with** is to cause editing commands to be read from the standard input (which is the default), but in line-by-line mode. The effect of a hyphen with **-ver** is likewise to cause verification output to be written to the standard output. (This is also the default – the syntax is really provided just for consistency.)

The preferred method of calling NE from within a Unix shell script to edit a file non-interactively using inline editing commands is:

```
ne somefile -with - <<End
<editing commands>
End
```

NE diagnoses an error if both **-from** and **-with** are specified (explicitly or implicitly) as the standard input. This also happens if **-to** and **-ver** are both specified explicitly as the standard output. If only **-to** is specified as the standard output, the default for **-ver** is changed to the standard error stream.

In non-interactive mode, NE stops if any error occurs, with a non-zero return code. When NE is run interactively, it issues a non-zero return code only when the **stop** or **abandon** command is obeyed. Normal termination always results in a return code of zero, even if there have been errors in the run.

4.1 Using NE as a filter

NE can be run as a filter under by specifying the input file as '–' (a single minus sign). The text to be edited is then read from the standard input and written by default to the standard output. The default command input is switched to the null file, and the default verification output to the standard error stream, but these can be changed by means of the **-with** and **-ver** keywords if necessary. Commands may also be specified using the **-opt** keyword, of course, as in this example:

```
ls | ne - -opt "rmargin 40; format" | more
```

The **-to** keyword can be used to direct the output to a different destination when the input is being read from the standard input, and, conversely, a minus sign can be used with **-to** to direct output to the standard output when input is not from the standard input.

4.2 Tab support

NE's treatment of tab characters in text files is unusual.

By default, tab characters are expanded when input lines are read, assuming tab stops every eight characters. Lines in which tabs have been expanded are marked as such, and when they are output, NE puts back as many tabs as it can, except that it does not use a tab where a single space will do. Note that this does not guarantee that unchanged output lines will be identical to the input lines.

The default action can be changed by an option on the command line, or by setting the environment variable NETABS. The command line option overrides the environment variable. The possible settings are:

- Option **-notabs** or NETABS setting ‘notabs’: tab characters are not treated specially. They will normally be displayed as question marks. Option **-tabs** or NETABS setting ‘tabs’: this restores the default action.
- Option **-tabin** or NETABS setting ‘tabin’: Tabs are expanded on input, but no action is taken when lines are output.
- Option **-tabout** or NETABS setting ‘tabout’: No action on input; all lines have as many tabs as possible inserted when they are output.
- Both options **-tabin** and **-tabout** or NETABS setting ‘tabinout’: Tabs are expanded on input, and all lines have as many tabs as possible inserted when they are output.

4.3 Editing binary files

Many personal computers have a ‘dump’ command or similar which displays files in hexadecimal and characters in the following sort of layout:

```
00A0  68 74 20 28 ... 55 6E 69 76 65  * ht (c) Unive *
00AC  72 73 69 74 ... 66 20 43 61 6D  * rsity of Cam *
```

Some have programs which allow the user to scroll around such a display, and possibly change its contents. These facilities are useful when working with files containing binary data.

NE contains some simple facilities that allow it to provide a similar function. It works as follows:

If the keyword **-binary** (abbreviation **-b**) is present on the NE command line, NE operates in ‘binary mode’. For example,

```
ne prog.o -b
```

This applies to the whole NE session and to all buffers. It is not possible to have some buffers in binary mode and some in text mode. If NE is entered in binary mode and it is also running in screen mode, screen handling is initialized in overstrike rather than replace mode.

When NE is in binary mode, it reads input files 16 bytes at a time. Each group of 16 bytes is converted into a textual input line in the following format:

```
aaaaaa  dd dd dd dd ... dd dd  * cccccccccccccccc *
```

where aaaaaa is the hexadecimal address within the file of the first byte of the 16, the dd’s are the hexadecimal representations of the characters, and the cccc’s are the character representations, with non-printing characters shown as full stops. The final ‘line’ of a file may represent fewer than 16 bytes.

The majority of the code of NE has no knowledge of binary mode, and it processes these constructed lines as if they were ordinary text lines. The lines may be modified by using any of NE’s repertoire of commands or screen editing facilities. There *is* some special knowledge in the screen driver such that, if the hexadecimal data in a line is changed, the character portion of the line is changed to match, and *vice versa*.

If the file is to be written out again successfully, the format must be preserved sufficiently to satisfy the following algorithm:

Whenever NE is required to output a line in binary mode, it ignores all characters before the first space and also the first asterisk and all characters that follow it. The remainder of the line must consist of pairs of hexadecimal digits, optionally separated by spaces. Each pair of hexadecimal digits is converted into a single byte, which is then written to the file.

Note that the spaces, if present, must fall between pairs of digits, not between the first and second digit of a pair. There may be more or less than 16 pairs, and the letters may be in upper or lower case.

If the format of any line is incorrect, an error message is output which shows the line at error. The data written to the file for that line will contain some bytes of rubbish. A number of error messages may be produced from a single writing operation. If the writing is the result of a W command, and there have been errors, NE does not exit.

5. Lines of NE commands

Lines of NE commands can be obeyed while screen editing as well as when editing line by line. An initial line of NE commands can be included in the system command line that invokes NE, and there is a way of specifying default commands to be obeyed automatically every time that NE is entered (the **.nerc** file). Before describing the individual commands in detail, some general discussion of the syntax is given.

5.1 Format of command lines

An NE command line consists of any number of NE commands, separated by semicolons except in one special circumstance (see below). Each command consists of a command *name*, possibly followed by additional data known as *arguments*. Command names are either a sequence of letters (for example, **rmargin**) or they consist of a single special character (for example, **#**). In the latter case there are never any arguments, and the commands are self-terminating, that is, a semicolon is not needed after such commands. The letters in a command name can be entered in either upper or lower case.

When a command whose name consists of letters is followed by an argument, a space is necessary following the command name if the argument begins with a letter. In other cases, spaces between the command name and the first argument are optional.

A command may be repeated by preceding it by a decimal number. It does not always make sense to repeat commands, but NE does not forbid it. An example of a sensible repetition is

```
132n
```

which has the effect of obeying the **n** command 132 times. A group of one or more commands may be repeated by enclosing it in round brackets and preceding it by a decimal number, for example

```
3(f/abc/; n)
```

Such command groups may be nested up to about 150 deep. They are used in other contexts as well as for repetition.

If two successive backslash characters are encountered in a command line, other than inside a delimited string or a file name, they signifies that the rest of the line is a comment which is to be ignored. NE behaves as if the line ends immediately before the backslash characters. The maximum length of a command line is 512 characters.

5.2 Continuation of command lines

Command lines may be continued onto as many input lines as necessary, provided that the line breaks occur inside brackets and at the end of a command. For example,

```
(ge/s//t/;
 ga/a//b/)
```

Brackets can be used solely for the purpose of introducing line breaks; they need not be preceded by a repetition count.

If NE is running interactively when a command line is continued, it prompts with the text ‘NE+’ for the second and subsequent lines, instead of the normal ‘NE>’.

5.3 Format of common arguments

There are several common kinds of argument that are used in more than one command. They are as follows:

- A *decimal number*, often simply called a *number*, is a sequence of decimal digits. It is terminated by the first non-digit encountered.

- A *word* is a sequence of letters, terminated by the first non-letter. Upper and lower case letters are synonymous in words.
- A *string* is a sequence of characters enclosed in *delimiter characters*. For any particular string, the delimiter character may not itself appear in the string. Only certain characters may be used as delimiters; they are

  ```
  ' " ! . , : + - * /
  ```

  These are the standard English punctuation characters (with the exception of semicolon and question mark) together with the characters normally associated with the four arithmetic operators in programming languages. Some examples of strings follow:

  ```
  /elephant/         .rhinoceros.
  :3.14159:          "The quick brown fox"
  !mighty atom!      +e = mc**2+
  ```

  When a string is the very last thing on a command line, the final delimiter may be omitted.
- A *qualified string* is a string preceded by certain qualifier characters. Details are given in the next chapter.
- A *search expression* is either a single qualified string, or several such strings connected by boolean operators. Details are given in the next chapter.
- Provided that the file name does not contain any semicolon characters or start with two backslash characters, it can be entered straightforwardly. If one or more semicolons are present, or if the first two characters are backslashes, the file name must be quoted using either single or double quote characters.

If a file name begins with a tilde character, it is interpreted in the same way as the shell would interpret it:

- If the name is of the form `~/`<*path*> the contents of the environment variable `HOME` are used in place of the tilde.
- If the name is of the form `~`<*user*>`/`<*path*> the entry in the password file for the given user is looked up, and the contents of its home directory field replaces `~`<*user*> in the file name.

While typing a file name as part of a line of commands while screen editing, the tab character can be used, as it is in some shells, to perform file name completion. However, NE provides only a very simple form of this facility, and only for screen editing.

5.4 Obeying commands while screen editing

There are two ways in which commands can be obeyed while screen editing. The keystroke ***read-command*** is a request to enter a line of commands from the keyboard. The cursor is moved to the last line of the screen, and the prompt '`NE>`' is output. The current position in the text being edited (that is, where the cursor was when ***read-command*** was pressed) is displayed in inverse video.

A line of commands can now be constructed by typing in the normal way. If an error is noticed, the arrow keys can be used to move the cursor back along the line, and correction can be carried out using the normal screen editing facilities. When the command line is complete, pressing ***enter*** causes NE to analyse it and, if there are no syntax errors, to obey it. If a syntax error is found, none of the commands are obeyed.

The second way of causing commands to be obeyed is to execute a keystroke which has been bound to one of the logical keystrokes ***keystring-1*** to ***keystring-60***. These cause the appropriate function keystring to be obeyed as a line of commands, just as if it had been typed by the user in response to the '`NE>`' prompt.

If a keyboard has function keys, these are normally set up by default to cause the keystrings of the same numbers to be obeyed. In addition, other keys are often bound to function keystrings, for example **ctrl/circumflex** and **ctrl/underline** are bound by default to ***keystring-58*** and ***keystring-59***.

The contents of all the non-empty function keystrings can be seen by obeying the command

```
show keystrings
```

and the relationships between actual keystrokes and keystrings can be seen by obeying

```
show keys
```

which shows the current bindings of the 'control-type', 'extra', and 'function-type' keystrokes [SHOW].

Keystrokes that execute function keystrings can also be used when the command prompt ('`NE>`'), or indeed any other prompt, has been output during screen editing. Any characters typed by the user before pressing the key which activates a function keystring are ignored. For example, if a context search in the forward direction fails, causing an error message to be output and leaving NE waiting for a new line of commands, pressing ***keystring-17*** causes a **bf** command to be obeyed; this searches backwards for the same context.

The user can alter the function keystrings [FKEYSTRING]. It is also possible to change the relationship between keystrokes and keystrings [KEY]. 'Function-type' keys can be set up to perform built-in operations (for example, to delete a line) and 'control-type' keys can be associated with function keystrings. That is why there are more function keystrings available than 'function-type' keystrokes.

When a line of commands is obeyed as a result of pressing a key that activates a function keystring, it is displayed at the bottom of the screen exactly as if it had been typed interactively by the user. If the command line executes successfully, this line is deleted. If, however, there is an error, the area at the bottom of the screen expands, leaving the command line still visible.

NE keeps a stack of up to one hundred previous command lines. A command line is added to the stack only if it is different to the previous command line. Lines on the stack can be recalled (one by one) by pressing ***cursor-up*** or ***cursor-down*** when the command prompt is displayed. ***Cursor-down*** cycles through the lines in historical order, starting from the oldest one on the stack, while ***cursor-up*** cycles through them in the reverse order, starting from the most recently obeyed line.

A recalled line can be edited in the usual way before pressing ***enter*** to cause it to be analysed and obeyed. At the start of an editing session the stack contains the text passed to NE via any automatic initialization and the **-opt** argument.

After obeying a command line that produces no output, NE reverts immediately to screen editing.

NE arranges that any output generated as a result of analysing or obeying a command line (of which error messages are a special case) is not wiped off the screen before the user has a chance to read it. If the output is an error message, or if it was caused by the last command on the line, NE outputs a further command prompt. This has the effect of keeping the cursor in the expanded area at the bottom of the screen. Simply pressing ***enter*** causes NE to revert to screen editing and to re-write the screen.

If non-error output is generated by a command which is not the last on a line, NE pauses after this command is finished, and outputs the message

```
Press RETURN to continue
```

When ***enter*** is pressed, processing of the command line proceeds. Any characters that the user may have typed before ***enter*** are ignored.

5.5 Long command lines while screen editing

If, while screen editing, a command line is entered that is too long to fit on the screen, NE scrolls the command input area to the left as necessary. When a command line has been scrolled horizontally, the ***scroll-left*** and ***scroll-right*** keystrokes can be used to view different parts of it, and further scrolling to the left or right happens automatically if an attempt is made to move the cursor past the edge of the screen.

The ***enter*** keystroke which terminates command entry can be typed at any time, wherever the cursor is placed. It does not have to be at the end of the line.

When a previous long command line is recalled for re-use, the last portion is displayed, with the cursor at the end of the line. It can be scrolled for editing in the normal way.

5.6 Refreshing the screen

NE does not normally update the screen display while a line of commands is being obeyed. Once the commands are finished, it inspects the current contents of the buffer and updates the screen as necessary. (An exception to this rule occurs during the processing of interactive global commands, when the screen is automatically updated if the user is to be prompted.)

Sometimes it is necessary to be able to force NE to make the screen display up-to-date in the middle of a line of commands. The command

```
  refresh
```

has this effect. (It is ignored if screen editing is not in operation.) **Refresh** does *not* cause NE to re-draw the screen completely. It simply causes the updating that would have occurred had there been no further commands on the line.

5.7 Summary

Command lines are available while screen editing as well as while editing line by line. They can be entered from the keyboard or stored in function keystrings and obeyed by a single keystroke. Many commands can be entered on one line, separated (in general) by semicolons. Commands may be grouped, and both groups and individual commands may be preceded by a repeat count.

6. Context matching

Searching a file for a particular context is a very common operation when editing. NE provides the means for performing very complicated kinds of search based on *search expressions* and *qualified strings*. These are used in a number of different commands, so they are described separately in this chapter.

The searching mechanism is based on the concept of matching a search argument against a line of text. The result of a matching operation is either a failure to match, or two character positions defining the start and the end of that part of the line which has matched the search argument.

6.1 Qualified strings

The simplest kind of search argument is a *qualified string*, and the simplest kind of qualified string is just a sequence of characters enclosed in delimiters. When such an argument is matched against a line, the match succeeds if the line contains the given sequence of characters. However, if any of the characters involved are letters, the upper and lower case forms are treated as synonymous. Thus, for example, if the search string is

```
  /Milton Keynes/
```

then all the following lines match it:

```
  The town of Milton Keynes is
  the town of milton keynes is
  THE TOWN OF MILTON KEYNES IS
  tHe tOWn Of MiLToN keYnes is
```

The automatic equating of the upper and lower case forms of the same letter can be disabled, either by a qualifier, for a single qualified string (see below), or by a command [CASEMATCH] which changes the default.

Frequently it is useful to apply additional constraints to search arguments. For example, occurrences of the string at the beginning of a line may be the only ones of interest. Such constraints are specified by means of *qualifiers* which precede the string. Most of the available qualifiers take the form of single letters (in either upper or lower case); one qualifier is a number, and one is a number pair. No spaces are necessary between individual qualifiers if more than one is present on a single string. They may be given in any order. Examples of qualified strings are given after the descriptions of all the qualifiers.

6.1.1 The B qualifier

The **b** qualifier specifies that the string is to be matched at the beginning of the line only. Occurrences of the string other than at the beginning of the line do not match. If a column qualifier (see below) is also present, it changes the effect of the **b** qualifier so that the string matches only at the beginning of the specified column, rather than at the beginning of the whole line.

6.1.2 The E qualifier

The **e** qualifier specifies that the string is to be matched at the end of the line only. Occurrences of the string other than at the end of the line do not match. If a column qualifier (see below) is also present, it changes the effect of the **e** qualifier so that the string matches only at the end of the specified column, rather than at the end of the whole line.

6.1.3 The H qualifier

The **h** qualifier specifies that the string is to be matched at the current cursor position ('here') only. This facility is intended mainly for use with conditional commands such as **if** and **while**. **H** can be combined with **p** in order to test the rest of the line from the current cursor position (see below).

6.1.4 The L qualifier

The **l** qualifier specifies that the search for the string in the line is to proceed in the reverse direction to normal, that is, from right to left instead of left to right. This means that, if a match occurs, it is the *last* occurrence on the line that is found. If a column qualifier (see below) is also present, it changes the effect of the **l** qualifier so that, if a match occurs, it is the last occurrence in the column, rather than the line, which is found.

6.1.5 The N qualifier

The **n** qualifier negates the result of the string match. If, taking into account all the other qualifiers, the result of the match is a failure (that is, the line does *not* contain the string) then the **n** qualifier converts it into a success, yielding the beginning and end of the line as pointers to a string that does *not* contain the one searched for. If, on the other hand, the string is found in the line, the **n** qualifier converts the result into a failure.

6.1.6 The P qualifier

If the **p** qualifier is present, the string match succeeds if the line contains *precisely* the given string, that is, the line consists only of the string, with no other characters before or after it.

If a column qualifier (see below) is also present, it changes the effect of the **p** qualifier so that it is the characters within the column which must match the given string precisely.

P can be used with a null string to match blank lines, or with a null string and a column qualifier to search for lines longer than a particular length. It can also be used with the **h** qualifier to cause the given string to be compared with the rest of the line, starting from the current cursor position. This facility can be used to check whether the cursor is at the end of a line by specifying a null string.

6.1.7 The R qualifier

The **r** qualifier specifies that the characters in the string are not to be used for a literal match against the line, but instead are to be interpreted as a *regular expression*. Regular expressions are a powerful way of expressing complicated matching conditions; they are described in section 6.3.

6.1.8 The S qualifier

The **s** qualifier causes NE to ignore leading and trailing spaces when matching strings at the beginnings or ends of lines. The letter ‘s’ is an abbrevation for ‘significant’. This qualifier is normally used in conjunction with the **b**, **e**, or **p** qualifiers.

6.1.9 The U qualifier

The **u** qualifier causes NE to match letters in an *un-cased* manner – that is, upper and lower case versions of the same letter are treated as the same letter. This is, of course, the default action of NE. However, the default can be changed [CASEMATCH], and this qualifier is provided to override the alternative default.

6.1.10 The V qualifier

The **v** qualifier causes NE to match letters in a *verbatim* manner – that is, upper and lower case versions of the same letter are *not* treated as the same letter.

6.1.11 The W qualifier

The **w** qualifier causes NE to match the string as a *word*, in the following sense: if the string is found in the line, the preceding and following characters are examined, if there are any. If either character exists and is found to belong to the set of characters that are permitted in words, then the match at that point in the line is cancelled, and NE carries on searching for another occurrence of the string. The default set of characters for words is the set of letters and digits; this can be changed by means of the **word** command (see *Changing default operations*).

6.1.12 The X qualifier

If the **x** qualifier is present, the string is interpreted as a sequence of pairs of hexadecimal digits representing the characters to be searched for. The characters specified by the hex pairs are always matched in a verbatim manner, independent of the presence of the **u** qualifier or the setting of the case matching default.

6.1.13 The repeat qualifier

A decimal number may appear as a qualifier. It specifies the number of times the string must be found in a line for the match to succeed. For example,

```
3/elephant/
```

matches a line containing at least three occurrences of the string ‘elephant’, and yields pointers to the third occurrence.

6.1.14 The column qualifier

The search for the string can be restricted to certain column positions in the line. The columns in a line are numbered from one. Two numbers, giving the inclusive starting and ending columns, may be given in square brackets, the numbers being separated by a comma. Either number may be omitted. If only one number is given, only a single column is searched. If one number followed by a comma is present, the search continues to the end of the line. If the first number is omitted but the comma is present, the starting column is the beginning of the line. The presence of this qualifier affects the behaviour of the **b**, **e**, **l**, and **p** qualifiers.

6.1.15 Combining qualifiers

Qualifiers may be combined in any sensible combination; nonsensical combinations are forbidden and provoke an error message, as do repeated occurrences of the same qualifier. Only one of **b**, **e**, **l** or **p** may appear on any one qualified string.

6.1.16 Matched strings

When a qualified string matches a line, it normally identifies a portion of the line which matches the string. However, if the **n** qualifier is present, it is the *whole line* which is considered to have been matched.

6.1.17 Examples of qualified strings

Some examples of qualified strings are shown below, together with a description of the characteristics of lines that they match:

`/abcd/`	contains ‘abcd’
`3/abcd/`	contains 3 occurrences of ‘abcd’
`[4,20]/abcd/`	contains ‘abcd’ in columns 4–20
`b/abcd/`	begins with ‘abcd’
`e/xyz/`	ends with ‘xyz’
`n/spqr/`	does not contain ‘spqr’
`nb/spqr/`	does not begin with ‘spqr’
`n3/spqr/`	contains fewer than 3 occurrences of ‘spqr’
`p//`	is an empty line
`[73,]np//`	contains characters after column 72
`sb/abcd/`	begins with optional spaces, then ‘abcd’
`v/Milton/`	contains ‘Milton’, as capitalized
`w/cat/`	contains the word ‘cat’
`x/7E4D/`	contains the two characters specified in hex

6.2 Search expressions

The simplest kind of *search expression* is a single qualified string, but more complicated search expressions can be formed by combining qualified strings using the boolean operators ‘and’ and ‘or’. Such a search expression is always enclosed in round brackets, and if it succeeds, it is the *whole line* which is considered to have been matched.

Within the brackets there are a number of search expressions, separated by one of the characters '&' or '|', meaning 'and' and 'or' respectively. For example,

```
(/cat/ & /dog/)
```

matches a line that contains both the strings 'cat' and 'dog'. Note that the order of the strings inside the brackets does *not* imply that the strings in the line are in the same order. The 'or' operation is inclusive, so that

```
(/mouse/ | /elephant/)
```

matches a line that contains either or both of the strings 'mouse' or 'elephant'.

It is also permitted to have just a single qualified string inside brackets, and this behaves differently from the unbracketed case. The results of matching `/xxx/` and `(/xxx/)` (for example) are not the same. The first matches part of a line; the second matches a whole line.

Because the items inside the brackets are search expressions in their own right, they can themselves be enclosed in brackets. Brackets are often necessary when both operators are used, to specify the order of combining the expressions. By default, the 'and' operator is more strongly binding that the 'or' operator, so that an expression such as

```
(/abcd/ & /spqr/ | /xyz/)
```

is equivalent to

```
((/abcd/ & /spqr/) | /xyz/)
```

and it matches a line that either contains both 'abcd' and 'spqr' or contains 'xyz' (or all three, since the 'or' operation is not exclusive).

Some of the qualifiers used in qualified strings can also be applied to complete search expressions by placing them immediately before the opening bracket. The qualifiers that are permitted in this position are:

`N`	negation of match result
`U`	un-cased matching
`V`	verbatim matching
`W`	word matching

When used in this way the **u** and **v** qualifiers apply to any enclosed qualified strings that do not themselves have a **u** or **v** qualifier. The following examples show equivalent search expressions:

```
n(/cat/ & /dog/)    =   (n/cat/ | n/dog/)
uw(/cat/ | v/dog/)  =   (uw/cat/ | vw/dog/)
```

Search expressions permit quite complicated matching conditions to be expressed, but because they impose no order on the different items in the line, there are certain kinds of search for which they cannot be used.

6.3 Regular expressions

Regular expressions permit the expression of complicated kinds of matching condition which are not possible with search expressions using ordinary qualified strings. A regular expression is essentially a pattern or template which is matched to the line. Since a regular expression is itself a kind of qualified string, it can form a component of a search expression if necessary.

The mechanisms for dealing with regular expressions are fairly complicated, because of the generality that is possible. For this reason, regular expressions should only be used when necessary; if the required matching condition can be expressed without their use, fewer resources will be consumed by NE.

When a regular expression is used in a command which searches for a string and replaces it or inserts another string alongside it (for example, the **ge** command), the replacement string can also be subject to special interpretation. Details are given in the next chapter.

The **r** qualifier, when present on a qualified string, causes NE to interpret the given string as a pattern for a regular expression instead of an ordinary text string. Other qualifiers may also be present, and have their usual effect. Within the pattern, certain characters take on a special meaning, and these are called *meta-characters*.

If there are no meta-characters in the string, the effect is as though the **r** qualifier were not present – except that NE consumes more resources. When matching a regular expression to a line, NE always tries to find the longest matching portion of the line that it can.

The current version of NE supports only Perl-like regular expressions, by means of the PCRE library. (Earlier versions supported a private syntax which was compatible with earlier Cambridge editors E and Zed. Now that regular expressions syntax is fairly well standardized, the old format has been dropped.)

The PCRE library must be installed in order to compile NE. A description of the regular expressions that it supports can be found in its documentation.

7. Character string insertions

A number of NE commands cause the insertion of a character string into a line. Such an insertion may replace an existing part of the line, or it may spread the line out to make room for the new material. When part of a line is being replaced, there is no requirement for the new material to contain the same number of characters. Replacement by a null string is one way of removing characters from a line.

In most cases, the characters of the insertion string are not interpreted in any way. However, there are three circumstances where this is not so. In all cases the special treatment of the string must be explicitly requested by means of a string qualifier.

7.1 Wild replacements for regular expressions

When a string insertion forms part of a command that first matches a search expression against the line (for example, the **e** or **ge** commands), and the search expression consists of a single regular expression, the appearance of the qualifier **r** on the insertion string causes it not to be taken as a literal string, but instead to be interpreted specially.

The character '`$`' is a meta-character in the insertion string. It is the only such character. The character following '`$`' is treated specially, as follows:

- If the character following '`$`' is the digit '`0`', the entire matched string is copied into the insertion string at that point, replacing '`%0`'.
- If the character following '`$`' is any other digit, the <*n*>th 'captured substring' is copied into the insertion string at that point.
- If the character following '`$`' is not a digit, the '`$`' is removed from the insertion string, and the following character is not interpreted. In particular, the string '`$$`' is replaced by '`$`'.

Here are some examples to show what can be done using this facility.

```
ge r/0\d*/ r/($0)/
```

This command encloses in brackets any sequence of digits beginning with zero. Thus 0223 would become (0223) and 081 would become (081).

```
ge r/\[(.)(.)(.)\]/ r/[$3$2$1]/
```

This command reverses the order of any three characters that appear in square brackets.

```
ge r/proc\s*([^(]+)\(([^)]*)\)/ r/proc $1($2,ierror)/
```

This command adds an extra argument, '`ierror`', to simple procedure definitions (in some programming language) that have only one argument (and no nested parentheses).

The **r** qualifier may be given on any insertion string. However, if the matching search expression does not consist of a single regular expression, then only the meta-sequence '`$0`' is useful. It is replaced by the portion of the line which was matched, as described in the previous section. For example,

```
e/expression/ r/($0)/
```

has the effect of enclosing the word 'expression' in brackets. If a digit greater than zero follows a dollar sign, nothing is substituted into the string.

7.2 Hexadecimal insertion strings

An insertion string is interpreted as a sequence of pairs of hexadecimal digits if it is preceded by the qualifier **x**. Such strings must contain an even number of characters. For example, the command

```
b// x/4c5d68/
```

inserts at the current position the three characters whose hexadecimal codes are 4C, 5D and 68 respectively.

Both the **x** and the **r** qualifiers may appear simultaneously on an insertion string. In this case the string can consist of a mixture of hexadecimal pairs and meta-sequences beginning with dollar signs.

8. Procedures

When a sequence of NE commands is to be used repeatedly, it is often convenient to define it as a named *procedure*. The name of a procedure can be any sequence of letters and digits, preceded by a full stop. The following are valid procedure names:

```
  .proc1 .x .special .r2d2
```

In the current version of NE, procedures are not permitted to have arguments.

A procedure is defined by means of the **proc** command:

`proc <`*name*`> is <`*command or bracketed command sequence*`>`

For example,

```
  proc .moan is comment /There’s a problem here!/
  proc .next is (f/something/; 5<; 5#; b// /body/)
  proc .uc   is until h/ / do %
```

The syntax for calling a procedure is simply to type its name. For example,

```
  unless /moon/ do .moan
  5.next
  bf/capital/; .uc
```

Procedures may be called recursively; that is, a procedure may call itself, or a number of procedures may be mutually recursive. A nesting limit of 150 is imposed, to catch runaway cases.

A procedure definition can be cancelled by obeying the command

`cproc <`*name*`>`

This is necessary before the procedure name can be re-used.

9. Single-character commands

This chapter contains descriptions of all those NE commands which consist of a single special character. With one exception (the ‘?’ command) they are all concerned with operating on the character at the current point.

9.1 The ? command

The **?** command is a request to NE to verify the current line. Two lines of output are always generated; a third and fourth may be present if the cursor is not at the start of the line, and if the line contains non-printing characters.

The first line of output contains the line’s number. If it has no number because it is an inserted line, four asterisks are output.

The second line of output contains the text of the line. If the line contains only printing characters, it is output in the normal way. If, however, there are any non-printing characters in the line, then two lines of output are generated. The first contains any printing characters, together with the first hexadecimal digits of the codes of any non-printing characters. The second line contains spaces in the positions of printing characters, and the second hexadecimal digits of the codes of any non-printing character.

If the cursor is not at the start of the line, an additional line of output is produced, containing the character ‘>’ in the position before the cursor. Thus, in the following example,

```
451.
first 0 second 1 third
      5        F
              >
```

The current line is number 451, and it contains two non-printing characters whose codes are 05 and 1F in hexadecimal. The cursor is positioned at the second of these characters.

Although the **?** command is mostly used when operating in line-by-line mode with automatic verification suppressed [VERIFY], it can be useful in screen mode for displaying the hexadecimal values of any non-printing characters that a line may contain.

9.2 The > and < commands

These two commands move the current position one character to the right or left in the current line respectively. They never cause another line to become current. If < is obeyed at the start of a line, it has no effect. If > is obeyed at the end of a line it has the effect of moving the current point beyond the end of the line. Neither of these commands is affected by the margin value. They are both faulted if obeyed when the end-of-file pseudo-line is current.

9.3 The # command

The # command has the effect of deleting the character at the cursor position and closing up the rest of the line by moving the remaining characters one place to the left. The cursor position is not moved. A second # command therefore deletes the next character. If a known number of characters are to be deleted, the standard repetition count mechanism can be used, for example

```
25#
```

Such repetition is optimised and is more efficient than obeying the equivalent number of successive # commands.

9.4 The $, % and ~ commands

These commands operate on the case of the character at the cursor position, provided that it is a letter. (Note that the case of whole lines can be forced by the **ucl** and **lcl** commands.) The single-character commands act as follows:

`$`	force lower case (small letter) – Dollar for Down
`%`	force upper case (capital letter) – Percent for uP
`~`	change to the opposite case

If the current character is not a letter, no change is made to it. The cursor is always moved one character position to the right after each of these commands, whether or not the current character was a letter; a second occurrence therefore affects the following character. If a known number of characters are to have their case changed, the standard repetition count mechanism can be used, for example

```
25$
```

Such repetition is optimised and is more efficient than obeying the equivalent number of successive **$**, **%** or **~** commands.

9.5 Single-character commands in line mode

When NE is being used as a line-by-line editor, the single-character commands can conveniently be placed under the verification of the line they are to affect. The following example shows a line's verification followed by a line of single-character commands, and then the verification of the changed line.

```
99.
the quoick BROWN ffox
~>>>>>#>>>>$$$$$>#
99.
The quick brown fox
               >
```

The cursor is left immediately following the '>' character, so subsequent single character commands may be entered immediately. It is in order to make this kind of editing possible that NE does not output its normal 'NE>' prompt in addition to line verification when operating line-by-line.

10. Search commands

Two commands are provided for searching forwards and backwards in the file. The command names are **f** and **bf**, and they take a single search expression as an argument. A third command, **df**, acts like **f**, but deletes the lines between the starting point and the found line. For example,

```
f/mousetrap/
f bs/subroutine/
bf (/if/ | /unless/ | /while/ | /until/)
bf (rv/A#$dZ/ | p//)
df/endproc/
```

10.1 The F command

For the **f** command, the search begins at the current point (including the character at the cursor position) unless one of the following conditions holds:

- The cursor is past the last character in the line.
- The cursor is not at the start of a line and the search expression is a qualified string containing the **b** or **p** qualifiers.
- The cursor is not at the start of a line *and* the search is a *line search*. A line search is one which, if successful, results in the identification of a complete line, rather than a string within a line. If the argument to the **f** command is a search expression in brackets or a qualified string containing the **n** qualifier, the search is a line search.

In all of these cases, the search begins at the first character of the line which follows the current line.

If the search expression is a qualified string containing the **h** qualifier, the search begins at the cursor position. If the initial match fails, subsequent tests are applied at the start of each successive line only. In other words, the **h** qualifier has the same effect as the **b** qualifier except on the first line that is searched.

If the end of the file is reached without a match being found, an error message is generated and the current point is not altered.

When a match is found, the current point is moved to just beyond the string or line which has been identified. For a line search, therefore, the current point ends up just past the end of the line, while for a *string search* (one that finds a string within a line) it is left just after the string. For example, suppose the file contains the following line:

```
The quick brown fox jumps over the lazy dog.
```

Then the command

```
f /fox/
```

leaves the cursor on the space character after ‘fox’, but if the command

```
f n/kangaroo/
```

matches this line (because previous lines do contain ‘kangaroo’) then the cursor is left after the final full stop.

10.2 The BF command

The **bf** command operates exactly as the **f** command, but in the opposite direction. That is, it searches backwards from the current point. The search begins at the character preceding the current point unless one of the following conditions holds:

- The cursor is at the first character in the line.
- The cursor is not past the end of a line *and* the search is a line search.

In both of these cases, the search begins at the last character of the line before the current line.

If the start of the file is reached without a match being found, an error message is generated and the current point is not altered.

When a match is found, the current point is moved to the start of the string or line which has been identified. For a line search, therefore, the current point ends up at the beginning of the line, while for a string search it is left on the first character of the string. For example, suppose the file contains the following line:

```
The quick brown fox jumps over the lazy dog.
```

Then the command

```
bf /fox/
```

leaves the cursor on the first character of ‘fox’, but if the command

```
bf n/kangaroo/
```

matches this line then the cursor is left at the beginning of the line.

10.3 The DF command

The **df** command operates exactly like the **f** command, except that all lines from the starting line (inclusive) until the line before the new current line are deleted. If **df** fails to match, then no lines are deleted.

10.4 Repeating search commands

An **f**, **bf**, or a **df** command can be given without an argument, in which case it repeats the most recent search (which may have been an **f**, **bf**, or a **df** command) in the appropriate direction. When screen editing, the ***keystring-7*** and ***keystring-17*** keystrokes (in their default settings) cause such an **f** or **bf** command to be obeyed, respectively. If a successful **f** command is immediately followed by **bf** without an argument (or *vice versa*) the effect is to move the current point to the other end of the string or line which has just been found.

The general command repetition facility can also usefully be used with the **f** and **bf** commands. A command such as

```
4f/white rabbits/
```

finds the fourth occurrence of the string ‘white rabbits’, starting from the current position.

11. Current point movement

There are, in addition to the search commands, a number of other commands which move the current point about the file, and they are described in this chapter.

11.1 The > and < commands

These single-character commands for moving the current point within the current line have already been described in chapter 9.

11.2 The BACK command

NE keeps a record of the twenty most recent regions of each buffer where modifications have taken place. Roughly speaking, a 'region' corresponds to a screenful of lines. The command **back** returns to the most recent region. By default, the keystroke ***keystring-59*** (default **ctrl/underline**) obeys a **back** command.

The following actions cause NE to start a new region:

- Any of the keystrokes ***scroll-up***, ***scroll-down***, ***scroll-top***, or ***scroll-bottom***.
- Scrolling caused by moving the cursor off the top or bottom of the screen;
- Obeying any of the commands **f**, **bf**, **df**, **i**, **m**, **n**, or **p**.

If a new region is selected, but no changes are then made to the file, then that region is not remembered. In this state, obeying **back** once returns to the most recently changed region, and pressing it a second time returns to the region before that. A succession of **back** commands (with no intervening changes to the file) then cycles round the twenty most recently changed regions.

Once changes have been made in a region, it becomes eligible for remembering. If **back** is obeyed in this state, the current region is remembered, and the cursor is moved to the previous region. Obeying **back** a second time returns to the region where it was first pressed. A succession of **back** commands (with no intervening changes to the file) then cycles round the changed regions.

Special action occurs when the **i** command is obeyed, either to insert a file or to insert lines directly from the command input. Both the top and bottom of the inserted material are remembered as separate regions. The cursor is left at the bottom of the inserted material; obeying **back** moves it to the top of the inserted material. A second **back** moves to the bottom again, and subsequent **back** commands then move to previously remembered regions.

Special action also occurs after a global command (**ga**, **gb** or **ge**), when **back** has the effect of moving to the last position in the file that was altered by the global command. Previously altered positions are not remembered.

The following types of use are envisaged for **back**:

- While making changes at one point in a file, other parts of the file are often consulted. A single keystroke brings the cursor back to the original area of editing.
- While editing a file, if ***scroll-top*** or ***scroll-bottom*** is pressed accidentally (which is quite easy to do on some keyboards) simply pressing ***keystring-59*** is enough to return to the point of editing.
- A part of a file can be cut out, moved elsewhere, and then editing resumed at the cutting point.
- While writing a program, it is easy to switch between the head of a procedure (for declarations) and its body.

11.3 The M command

The **m** command is used to move to a particular line by reference to its line number. When the file that is to be edited is read, the lines are counted, and each line is allocated the appropriate line number. Lines retain these numbers for the duration of the editing session, unless the **renumber** command is obeyed. Deleting a line does not result in a re-numbering of the subsequent lines, as happens in some other editors.

New lines that are inserted into the file are un-numbered, and cannot become current via the **m** command (unless the whole set of lines is subsequently renumbered with the **renumber** command). Lines which are moved about in the file by cut-and-paste operations lose their numbers and behave as inserted lines.

A single argument is required for the **m** command. It is either a string of decimal digits, or the single character `*`, which is used conventionally to refer to the end of the file. There need not be a space between the command and its argument. For example,

```
m1234
m9
m*
m0
```

The lines in the file are numbered starting from one. However, the number zero is permitted as an argument to the **m** command. It causes NE to move to the first line in the buffer, without regard to its line number.

If an **m** command succeeds, the current position is moved to the first character of the appropriate line; if it fails, the current position is not changed.

When screen editing, the keystrokes ***keystring-8*** and ***keystring-18*** are set up to obey the commands **m*** and **m0** respectively. The keystrokes ***scroll-bottom*** and ***scroll-top*** perform the same functions, except that they do not alter the horizontal position of the current point. However, not all keyboards have dedicated keys that can be bound to these functions.

11.4 The N command

The **n** command moves the current position to the first character of the next line in the file. It can be used with a repeat count to move forwards by a given number of lines, for example

```
150n
```

If it is obeyed when the current line is the last in the file, the current point is moved to the start of the end-of-file pseudo-line. This line becomes a real line only if characters are inserted into it (at which point it ceases to be the end-of-file line). If **n** is obeyed while the end-of-file line is current, an error is caused.

11.5 The P command

The **p** command moves the current position to the first character of the previous line in the file. It can be used with a repeat count to move backwards by a given number of lines, for example

```
150p
```

If it is obeyed when the current line is the first in the file, an error is caused.

11.6 The PA and PB commands

The **pa** and **pb** commands move the cursor by context within the current line. The names of the commands are abbreviations for ‘point before’ and ‘point after’. Each takes a search expression as an argument. The current line, starting from the current point, is matched against the search expression. If there is no match, an error occurs. Otherwise the current point is moved to the first character that matched (**pb**) or just after the last character that matched (**pa**). For example, if the current line is

```
  Scherzo in G Op. 6
```

then the command

```
  pa/zo/
```

places the cursor just after the word ‘Scherzo’.

11.7 The PLL and PLR commands

These two commands, which take no arguments, move the cursor to the extreme left or extreme right of the current line, respectively. The command names are abbreviations for ‘pointer to line left’ and ‘pointer to line right’, respectively. The keystrokes ***keystring-6*** and ***keystring-16*** are set up by default to obey **pll** and **plr**.

When the current line is not wider than the screen, these commands are synonymous with the ***first-char*** and ***last-char*** keystrokes respectively. When editing long lines, however, they move to the true beginning and end of the line respectively, causing a horizontal scroll if necessary.

11.8 The TOPLINE command

The **topline** command has no effect unless NE is in screen editing mode. When this is the case, **topline** causes the screen to be re-displayed with the current line at the top. The position of the current point is not altered.

The keystroke ***keystring-58*** (default **ctrl/circumflex**) is set up to call **topline** by default. Placing the cursor on a particular line and pressing ***keystring-58*** therefore has the effect of moving the line where the cursor is to the top of the screen. **Topline** can also be useful in command lines such as

```
  F/procedure/;topline
```

If **topline** appears in the middle of a command line, and subsequent commands on the same line change which line is current, then the effect of **topline** may be lost. What happens when **topline** is obeyed is that NE remembers the line which is current at that time. When the screen is subsequently re-drawn, if the current line (which might have been changed as the result of other commands) will be visible with the remembered line at the top of the screen, then the screen is written like that. Otherwise, the information saved by **topline** is ignored. Thus, the following two command lines are *not* equivalent:

```
  m23; topline; p
  m23; p; topline
```

For this reason, **topline** should normally be the last command in a line.

12. Changing the current line

This chapter describes a number of commands for making changes to the current line. In addition, there are a number of single-character commands which affect the contents of the current line. These are described in chapter 9.

12.1 The A, B and E commands

These three commands make contextual changes to the current line. Each takes two arguments: a search expression that defines which part of the current line is to be altered, and an insertion string to be put into the line. The difference between the commands is that

- The **a** command inserts the string *after* the text which matches the search expression;
- The **b** command inserts the string *before* the text which matches the search expression;
- The **e** command *exchanges* the text which matches the search expression for the insertion string.

The search for the given context in the current line starts at the cursor position. After the command is obeyed, the cursor is moved so that it follows the last affected character in the line. For example, if the current line is

```
In describing the experiences of the
```

with the cursor at the start of the line, then after obeying the command

```
a/describing/ / briefly/
```

the line becomes

```
In describing briefly the experiences of the
```

and the cursor is left following the word ‘briefly’. If the commands

```
b/experiences/ /novel /; e/the/ /a/
```

are now obeyed, the line becomes

```
In describing briefly the novel experiences of a
```

Note that is is the second occurrence of the word ‘the’ which is changed, because the cursor is left after ‘experiences’ by the **b** command.

If an **a**, **b** or **e** command has a first argument which specifies a line search, that is, it contains the **n** qualifier or is enclosed in brackets, then the current position must be at the start of a line. If this is not the case, an error occurs.

The **a**, **b** and **e** commands can be given without any arguments, in which case they operate by re-using the arguments for the most recent such command.

12.2 The DTA and DTB commands

These two commands delete characters from the line, starting at the cursor position, and ending either after or before a matched qualified string, respectively. For example,

```
dta w/foxes/
```

There is an error if the qualified string does not match on the current line.

12.3 The LCL and UCL commands

These two commands have the effect of converting any letters in the current line at and following the cursor position to lower case or upper case respectively. The cursor is moved to just beyond the end of the line, unless it was already further to the right.

13. Inserting text into the file

There are a number of ways of inserting text into a file. The *cut and paste* facilities are described in the chapter on screen editing, and the equivalent commands are described in the chapter entitled *Keystroke commands*. The use of the **i** command for inserting whole files is described in the chapter entitled *File operations*. The remaining commands are described here.

13.1 The I command

The **i** command can be used to insert a whole file into the text, and this is described in the chapter on file operations. It can also be used to insert lines of text from the command stream. If **i** is obeyed with no argument, it causes successive lines of command input to be inserted into the edited file *before* the current line, until a line containing only the letter ‘z’ (in upper or lower case) is encountered. This line serves to terminate the insertion; it is not itself inserted into the file. For example, to insert some lines before line 345:

```
m345; i
Here are some inserted lines
to go before line 345.
z
```

The cursor position is not changed by the **i** command. This form of the **i** command is normally used when editing line-by-line, but it can be used while screen editing, in which case the input takes place in the command area at the bottom of the screen.

If NE is running interactively, it prompts for each successive line of insertion only if screen editing is in progress. The prompt text is the string ‘NE<’ instead of the normal command prompt ‘NE>’. No prompts are given in line-by-line mode.

13.2 The ICURRENT command

The **icurrent** command has no arguments. It has the effect of inserting a copy of the current line immediately before the current line. The cursor position is not changed. The command repetition facility can be used with **icurrent** to replicate a line any number of times.

13.3 The ILINE command

The **iline** command can be used to insert a single line of text before the current line. It takes a delimited string as an argument – this is the text of the line to be inserted. The cursor position is not changed. The argument of **iline** may be given with the **x** qualifier, in order to specify a new line as a string of hexadecimal pairs representing the characters of the line.

14. Splitting and joining lines

When screen editing, lines can be split and joined using the ***split-line*** and ***concatenate*** keystrokes, as described in chapter 3 above. There are also the following commands for performing these operations.

14.1 The SA and SB commands

These commands split the current line after or before a given context, respectively. They take a single search expression as an argument. For example, if the current line is

```
The cow jumped over the moon
```

then the command

```
sa/jumped/
```

turns it into the following two lines:

```
The cow jumped
 over the moon
```

The search for the given context takes place in the current line only, starting from the cursor position.

By default, the second part of a split line is made into a new line with its first character in column one. It is, however, possible to arrange that this line is given the same indent as the previous one, by using the **autoalign** command to set *autoalign mode*. For details, see the chapter entitled *Changing default operations* below.

14.2 The CL command

The **cl** command concatenates the current line with the following line. It may be followed by an optional string argument, which defines a string of characters to be inserted between the two lines. The cursor is left at the start of what was the second line. If the cursor is beyond the end of a line when **cl** is obeyed, spaces are added to the line to lengthen it to the cursor position before the second line is joined on.

The **cl** command can take a hexadecimal string as its (optional) argument, for example:

```
cl x/0d/
```

15. Deleting parts of the file

Several ways of deleting text exist. The **dmarked** command is an exact analogue of the screen editing keystroke ***delete-marked***, and is described in the chapter entitled *Keystroke commands*. The **df** command is described in the section 10.3.

15.1 The DLINE command

The **dline** command, which takes no arguments, deletes the current line from the file and moves the cursor to the first character of the following line. The normal command repetition facilities can be used to delete a number of lines. For example

```
  23dline
```

deletes twenty-three lines, starting from the current line.

15.2 The DREST command

The **drest** command, which takes no arguments, deletes the current line and the remainder of the file which follows it. The use of **drest** speeds up NE’s processing, because it does not need to read from the disc those lines which it has not yet got in main memory.

16. Restoring deleted text

The **undelete** command, which by default is bound to ***keystring-4***, retrieves deleted characters and lines. Up to 100 deleted lines are remembered, in reverse order. Deleted characters are held in a packed form, and up to 64 consecutive characters can be held in one ‘line’ for this purpose.

Undeleted characters are inserted back into the text at the current point, and NE remembers whether they were deleted forwards or backwards, so that a sequence of character deletes followed by a sequence of undeletes should put the text back as it was. Undeleted lines are treated as text to be inserted at the current point.

Text deleted with commands such as **e/a//b** is also added to the undelete stack, as is text cut out with the **delete** command (the **ctrl/q** keystroke). Text cut or copied to the cut buffer is *not* added to the undelete stack.

Note that this does *not* provide a general ‘undo’ facility.

17. Formatting commands

When new text is being entered into the file while screen editing, the position of the right-hand margin controls the maximum width of line that can be entered without special action. The user can type input continuously, without ever pressing the ***split-line*** (i.e. **return**) key – this is known as *power typing*. When the right-hand margin is passed, the line is automatically split at the previous space character, which is itself deleted from the text. If there is no previous space character on the line, it is split immediately after the character in the right-hand margin position, leaving a line that is full right up to the margin.

The second half of a split line is made into a new line with its first character in column one. It is possible, however, to arrange for it to have the same indent as the previous line, by using the **autoalign** command. For details, see the chapter entitled *Changing default operations* below.

17.1 The CENTRE command

The **centre** command centres the current line, ignoring leading spaces, within the current margin value.

17.2 The RMARGIN command

The default margin value is one less than the width of the screen (typically 79). If NE is being run from a window-based terminal emulator such as **xterm**, then the width of the window is used. The margin can be altered by means of the **rmargin** command, which takes three forms:

- If given without an argument, it disables the margin without forgetting its position or, if the margin is already disabled, it enables it at the previously-remembered position. This form of the **rmargin** command is used by the ***keystring-10*** keystroke. On the screen display, the position of the first character beyond the maximum line width is shown by a vertical bar or a backslash in the separator lines, depending on whether the margin is enabled or disabled. When the margin is disabled, no automatic line splitting takes place.
- **Rmargin** can also be followed by one of the words **on** or **off** which have the effect of enabling or disabling the margin, respectively, without changing its position.
- If **rmargin** is followed by a number, it sets the margin position and enables the margin, whatever its previous state. The margin position need not coincide with the edge of the screen. The number specifies the maximum line width required.

The margin value affects only two features of NE:

(1) It always controls the width of lines produced by the **format** command, whether it is enabled or not.

(2) When it is enabled, it defines the column at which a data character causes automatic line splitting (the ‘power typing’ column).

Characters typed beyond the margin do not cause line splitting.

17.3 The BEGINPAR and ENDPAR commands

The **beginpar** command is used to specify an alternative definition of the beginning of a paragraph for use by the **format** command. It takes a single search expression as its argument; lines which match the search expression are valid paragraph beginnings. The default state is equivalent to

```
  beginpar nps//
```

That is, a line must contain a character other than a space to be recognized as the start of a paragraph, unless **beginpar** has been used to specify otherwise.

Similarly, the definition of the end of a paragraph can be specified by the user by means of the **endpar** command. Any lines that match the search expression are taken by **format** as being ‘end of paragraph' lines; the preceding paragraph is terminated when such a line is reached. The default state is equivalent to the command

```
  endpar (p// | b/ /)
```

That is, a completely empty line or a line beginning with a space terminates a paragraph unless **endpar** has been used to specify otherwise.

As an example of the use of **beginpar** and **endpar**, consider the editing of a file that is input to the SGCAL formatting program. Changes to the text may result in lines that contain only a few words, or lines that are longer than the width of the screen, and it is often convenient to reformat paragraphs while editing, even though this is not strictly necessary. However, lines which begin with a full stop are not part of the text, and should never be re-formatted. The commands

```
  beginpar (nps// & nb/./); endpar (p// | b/ / | b/./)
```

are therefore appropriate in these circumstances.

17.4 The FORMAT command

As well as controlling the length of input lines on the screen, the right-hand margin interacts with the **format** command. This command, which has no arguments, causes the rest of the current paragraph of text to be reformatted so that its lines are of maximum length within the current margin. (The ***keystring-20*** keystroke issues a **format** command.) Disabling the right-hand margin does not affect the operation of the **format** command; it still uses the current margin setting.

If the current line is the end-of-file line when **format** is obeyed, no action is taken. Otherwise the current line is checked to see whether it is valid as the start of a paragraph. By default, any line that is not completely empty starts a paragraph.

If **format** is obeyed when the current line does not match the beginning of paragraph search expression, the only action taken is to move the cursor to the start of the following line.

When the current line does match, the re-formatting operation begins with the current line, and ends with the line preceding the next 'end of paragraph' line (which is defined below). The effect is as if the entire paragraph is made into one long line and then split up again so as to fit the maximum number of words into each resulting line, none of which can be longer than the margin value. A word, in this context, is any sequence of characters not including a space.

When two lines are joined together, a single space is inserted between them, unless there is already a space at the end of the first line or the beginning of the second, or unless either line is empty. When lines are split, space characters at the splitting position are removed. Apart from this, no changes are made to the paragraph. Sequences of multiple spaces, for example, remain. After **format** has been obeyed, the current point is left at the start of the line following the re-formatted paragraph.

The **format** command works automatically on paragraphs that start with a tag on each line, for example, paragraphs like:

```
  # This paragraph
  # has a # at the start
  # of each line.
```

A tag is recognized if it consists of 10 or fewer characters from the following set: # % * + = | ~ < > and space. If a paragraph consists of only one line (before formatting), such a tag is always recognized. If there is more than one line, the first two must start with identical tags, and the paragraph will end at a non-matching line.

If a tag begins with a space, which is the case with indented paragraphs, this feature does not work with the default settings of **beginpar** and **endpar** because an indented line is treated as ending the paragraph. However, this can be changed, and if you want this to be the default, you can put appropriate settings in your **.nerc** file.

You have to set **endpar** not to treat line starting with a space as a paragraph end. Also, you must use the **h** instead of the **b** qualifier for both commands if you want to be able to detect paragraph starts and ends among a sequence of flagged lines (for instance, while replying to an email and wanting to format quoted paragraphs). For example you could use:

```
  beginpar(nphs//); endpar(phs//)
```

Because **format** leaves the current point on the line following the paragraph, a number of paragraphs can be formatted at once by a command such as

```
  10format
```

or by pressing ***keystring-20*** a number of times in succession while screen editing. Note that the count must allow for any blank lines between paragraphs.

18. Global changes

It is often necessary to make systematic changes in a file, for example to change the spelling or capitalization of a particular word. Three commands which operate in a similar way are provided for this purpose.

18.1 The GA, GB, and GE commands

Each command takes two arguments, a search expression and an insertion string. The only difference between the commands is the way in which each change is made:

- The **ga** command inserts the string *after* each occurrence of text which matches the search expression;
- The **gb** command inserts the string *before* each occurrence of text which matches the search expression;
- The **ge** command *exchanges* the text which matches the search expression for the insertion string.

For example, to change the word ‘bit’ to the word ‘piece’ throughout a file, the command

```
ge w/bit/ /piece/
```

would be used. The qualifier ‘w’ (word) ensures that words such as ‘bite’ do not get changed to ‘piecee’.

If a search expression enclosed in brackets is used as the first argument, it is a complete line which is matched. The command

```
ga (np// & n/ /) / **/
```

adds two asterisks to any line that is not empty, but contains no spaces.

If a regular expression is used as the first argument, then the second argument (the insertion string) can be interpreted specially, by use of the **r** qualifier. In this case the character ‘%’ is used to trigger the insertion of wild strings encountered during the matching process. The insertion string may also be given in hexadecimal, if the **x** qualifier is used. Details of both these facilities are given in chapter 7.

A global command can be given with no arguments, in which case it re-uses the arguments of the most recent global command.

A global command searches forwards in the file from the current point until the end of the file or the *global marker* (see below) is reached. The search is carried out in exactly the same way as for the **f** command. When a global command finishes normally, the current point is restored to what it was at the start. Therefore, a repeated global command or a number of global commands on the same command line all act from the same point in the file.

If no match is found, an error occurs if editing is interactive. When NE is running non-interactively, the absence of any matches is not an error. The file is unchanged and the current position unaltered.

When a match is found, the subsequent action depends on whether NE is running interactively or not.

18.2 Interactive global commands

When NE is running interactively, the global commands interact with the user, making it possible to change some of the matched strings and not others. When a match is found, NE behaves as follows:

If screen editing is taking place, the screen is updated to show the new current position, with the characters which matched shown in inverse video. When it is necessary to re-draw the screen to do this, the line containing the matched characters is placed near the top, in an attempt to minimize the amount of screen updating during the global command. In line-by-line mode the line which matched is verified. Then the prompt

```
Change, Skip, Once, Last, All, Finish, Quit or Error?
```

is displayed, and NE waits for the user to reply. The reply must be the first letter of one of the words in the prompt string, followed by ***enter***; the possible actions are as follows:

- Change: the line is changed, and NE searches for the next match, starting from the character following the matched or inserted text, whichever is the rightmost in the line.
- Skip: the line is not changed; NE searches for the next occurrence of the search expression starting from the character following the matched text.
- Once: the line is changed, and the global command then terminates. The current point is put back to where it was at the start of the command.
- Last: the line is changed, and the global command then terminates. The current point remains immediately after the last change.
- All: the line is changed, and NE searches for the next match. All subsequent matching strings are changed without prompting. When the end of file (or global marker – see below) is reached, the command terminates and the current point is put back to where it was at the start.
- Finish: the line is not changed, but the global command terminates. The current point is put back to where it was at the start.
- Quit: the line is not changed, but the global command terminates. The current point is *not* put back to where it was at the start.
- Error: the line is not changed, but the global command terminates abnormally (though no error message is given). This means that any further commands that were given on the same command line are not obeyed, and NE awaits a new line of commands. The current point is put back to where it was at the start of the global command.

It is possible to give several responses in a single reply to a global interaction prompt. For example,

```
cccssc
```

means ‘change three occurrences, skip two, then change one more’. Furthermore, repeat counts can be embedded in the string. The following response has the same meaning:

```
3c2sc
```

Spaces are allowed in the reply string and are ignored. If the response string contains anything other than spaces, digits, or the allowed response letters, it is rejected before any action is taken, and the prompt is re-issued.

After one prompt has been given, reaching the end of the file or the global marker is no longer an error. If it happens, the message ‘no more’ is output, and the command terminates.

At the end of an interactive global command, NE displays the number of matches and the number of changes made at the bottom of the screen, and the cursor is reset to where it was before the command began executing, except when either the ‘`q`’ or ‘`l`’ response was used to terminate it. This happens whether or not the command was successful.

18.3 Non-interactive global commands

When NE is running non-interactively, a global command always affects all occurences of any text which matches the search expression, between the current point and the end of the file or the global marker. If no occurrence is found, however, no error occurs. At the end of the command the cursor is reset to where it was before the command began executing.

If a **ga**, **gb** or **ge** command is obeyed from inside a command file called by the **c**, **cbuffer**, or **cdbuffer** commands, it behaves in non-interactive fashion, and does not prompt for confirmation before making changes.

18.4 Continuing after a match

When a global command resumes searching the file after a match, it starts at the first character beyond the string that it has just matched. This avoids any problems with accidental recursion. However, it can sometimes lead to unexpected results. For example, the following command might be expected to replace all occurrences of a double space between words by a single space:

```
ge r/$l  $l/ r/%1 %2/
```

However, if a word consisting of only one letter has double spaces on either side of it, this command changes only the first pair of spaces. If it is required, as in this example, to re-scan previous text, then combinations of other NE commands must be used instead of a global command.

18.5 Null strings in global commands

The use of a null string as the first argument in a global command is forbidden, unless one of the qualifiers **b**, **e** or **p** appears. For example,

```
gb b// /*/
ga e// /  !!/
```

When such a global is obeyed in screen mode, and a prompt is issued, a single character is displayed in inverse video to indicate the position of the null string. It is the character following the string's position.

18.6 Globals with margins or long lines

The right-hand margin setting does not affect the operation of the global commands; the entire text of each line is searched, as for the **f** command. The margin is automatically disabled for the duration of each global command.

Automatic scrolling of the window to left or right occurs as necessary to show the matched text. For a **ga** command the window is adjusted so as to show the end of the text; for the other commands it is the beginning of the text which is always shown.

18.7 The global marker

Global commands normally operate from the current point to the end of the file.

However, an endpoint can be set earlier in the file. This takes the form of a marker similar to those used for block and multi-line operations. It is set and unset when screen editing by the keystroke ***mark-global***, (default **ctrl/n**) exactly like the other marks, and the message

```
Global limit set
```

appears at the bottom of the screen when it is set [MARK]. Only one kind of mark can be set at one time.

The global limit marks the end of the search for a match when a global command is being obeyed. If the current point is past the global mark when the search is started, the limit has no effect. Global commands always work forwards through the file.

19. Buffer handling

An NE *buffer* is a separate area of memory which can hold text for editing. As well as its contents (lines of text), a buffer has an associated file name, a title, and a right-hand margin. The title is by default the same as the file name, and is displayed in the bottom separator line when screen editing. Each buffer has a number, starting from zero. Buffer zero is automatically created when NE is entered. Other buffers may also be created during initialization if more than one input file name is given on the command line.

As well as the numbered buffers, there is one special buffer known as the *cut buffer* which is used for holding text which has been cut or copied from the file. Its use is described in section 3.11 (*Cutting, pasting and block deletion*).

The description of NE so far has assumed that a single file is being edited at any one time. In fact, NE makes it possible to edit more than one file at once, and to move material between files as necessary.

19.1 The TITLE command

The **title** command can be used to change a buffer's title string. The title is the string which is used in all displays and prompts to identify the buffer. By default it is the same as the name of the file associated with the buffer; the **makebuffer**, **newbuffer**, **load**, and **name** commands, which set up new file names, also set the title to be the same as the file name. The **save** command does the same when it changes a buffer's file name.

The argument to **title** is in the same format as a file name, that is, it can be an undelimited string, terminated only by end of line or semicolon. However, the standard file name delimiters can be used if required, and are necessary if the string contains a semicolon or begins with a backslash.

19.2 The RENUMBER command

The **renumber** command causes all the lines in the current buffer to be renumbered, starting from one. Whichever line is current remains current after the renumbering, though its number may be changed.

19.3 The DETRAIL command

NE normally takes no special action over trailing spaces in text lines. They are treated like any other character. However, the **detrail** command is provided to vary this action. It can be used in two ways:

- If **detrail** appears with no arguments, it causes all trailing spaces in the current buffer to be deleted.
- If **detrail** appears followed by the word 'output', then trailing spaces are removed from each line when it is output.

19.4 The NEWBUFFER or NE command

The **newbuffer** command is used to create additional buffers. The **ne** command is a synonym for **newbuffer**. There is no limit to the number of buffers that may exist, other than that imposed by the amount of available memory. If **newbuffer** is obeyed without an argument, it creates an empty buffer, with no associated file name. If the command name is followed by a file name, that name becomes associated with the new buffer and the contents of the file are read into the new buffer for editing. The title of the new buffer is set to the file name. If the name of a non-existent file is given, some operating systems may create an empty file (possibly only if an appropriate option is given); others may force an error. The right-hand margin is copied from the current buffer for newly-created buffers.

If the current buffer is marked read-only, then the newly-created buffer is also so marked.

When a **newbuffer** command is successfully obeyed, NE automatically selects the new buffer as the current editing buffer. In the screen display, an additional item appears in the bottom separator line when more than one buffer exists. It is the number of the displayed buffer, enclosed in square brackets, following the file name. Any commands which follow **newbuffer** on the same line are applied to the new buffer; for example

```
  newbuffer another.file; f/something/
```

creates a new buffer, reads the file `another.file` into it, and searches for the word ‘something’ in the new file.

19.5 The MAKEBUFFER command

This command operates like **newbuffer**, but allows (indeed, requires) the user to specify the number of the buffer which is being created, for example:

```
  makebuffer 19 somefile
```

If a buffer of that number already exists, an error occurs. The newly-created buffer is *not* made the current buffer, in contrast to the behaviour of **newbuffer**. This command is useful in NE command files where the number of the buffer is needed for use in other commands.

19.6 The BUFFER command

The **buffer** command is used to switch between existing buffers. If given without an argument, it cycles round the buffers in reverse order of creation. However, an explicit buffer number can be given as an argument if required. The ***keystring-1*** keystroke is set up by default to obey a **buffer** command with no argument.

19.7 The PBUFFER command

The **pbuffer** command (‘previous buffer’) operates like **buffer**, but cycles through the buffers in the opposite order. By default, ***keystring-11*** is set up to obey **pbuffer**.

19.8 The NAME command

The **name** command is used to change the file name associated with a buffer (or to give a name to a buffer that was created without one); it takes a file name as an argument. This command also sets the title of the buffer to the new file name. Changing the file name of a buffer has the effect of marking the buffer ‘changed’, so that when the NE run ends normally, the contents of the buffer are written to the named file (subject to user confirmation if interactive).

19.9 The DCUT command

The **dcut** command deletes the contents of the cut buffer. This command is only necessary when the cutting style is set to ‘append’ [CUTSTYLE], as otherwise each cutting operation overwrites what is already in the buffer. The **dcut** command does not prompt if the cut buffer has not been pasted; it can therefore sometimes be of use in command sequences as a means of suppressing this prompt.

19.10 The DBUFFER command

The **dbuffer** command deletes all the lines in a buffer, closes any associated files, and then deletes the buffer itself, unless it is the only buffer. **Dbuffer** can take a buffer number as an argument, or can be given on its own, in which case it refers to the current buffer.

If the object of **dbuffer** is the current buffer and there are no other buffers, then the effect is simply to delete all the lines in the buffer and close any associated files. If there are other buffers, then the effect is as if a **buffer** command had been obeyed first (i.e. the previous buffer is selected), and then the object buffer is completely destroyed.

If the contents of the object buffer have not been saved, NE prompts and asks for confirmation before proceeding with a **dbuffer** command when running interactively, unless prompting has been suppressed by means of the **prompt** or **warn** commands.

19.11 Copying between buffers

Copying or moving data between buffers can be carried out using the standard cut-and-paste operations. There is only one cut buffer, so the process is as follows when editing on the screen:

- The source buffer is selected, and the text or rectangle is cut or copied to the cut buffer in the usual way, using the keystrokes ***mark-text***, ***mark-rectangle***, ***cut-copy*** and ***cut-delete*** as appropriate.
- The destination buffer is selected, the cursor is moved to the insertion position, and the text or rectangle pasted in using the keystroke ***paste***.

When using one buffer solely as a source of text for insertion in another, it is better to use the ***cut-copy*** keystroke rather than ***cut-delete*** because this avoids marking the source buffer as 'changed'.

An alternative way of inserting a complete file is to use the **i** command, which is described in the next chapter.

20. File operations

This chapter contains descriptions of those NE commands that operate on whole files. The format of file names is system-dependent, and details are given in the chapters describing each system's interface.

20.1 Loading a new file

The **load** command takes a file name as an argument. It causes all the lines in the current buffer to be deleted and any associated files to be closed. The named file is then opened for input, and its contents read into the current buffer. The title of the buffer is set to the file name.

If the contents of the current buffer have not been saved, NE prompts and asks for confirmation before proceeding with a **load** command when running interactively, unless prompting has been disabled by means of the **prompt** or **warn** commands.

20.2 Inserting files

To insert the entire contents of a file into the text which is being edited, the **i** command is used, taking as its argument the name of the file to be inserted. The **i** command can be used in another form for inserting in-line material; this is described in chapter 13 above. An example of the use of **i** for inserting a whole file is

```
  i myfile
```

The lines of the file are inserted immediately prior to the current line. The actual position of the current point in the current line is not relevant, but it is not altered.

If a **back** command is obeyed immediately after an **i** command, it moves the current point to the top of the inserted material.

The newly inserted lines do not have line numbers, as they are not part of the original file which is being edited. It is therefore not possible to move around in the new text by means of the **m** command, unless all the lines in the buffer are renumbered [RENUMBER].

An alternative way of merging files is to use a second editing buffer and cut-and-paste operations. In this case it is possible to use the **m** command to move around the second file.

20.3 Saving files

The **save** command is used to cause the contents of the current buffer to be written to a file, without leaving NE. The command can be given with or without a file name as an argument.

If **save** is given without an argument, the file name associated with the current buffer is used by default. (The **name** command can be used to change the file name associated with the current buffer.) If NE is running interactively when **save** is obeyed without an argument, it issues the prompt

```
  Write to <buffer title>? (Y/N/TO filename)
```

unless there is no name associated with the buffer, in which case the prompt is

```
  Write? (N/TO filename)
```

If prompting for the buffer has been disabled by means of the **prompt** command, then a prompt is issued only when there is no associated file name.

When a prompt is issued, if the reply is **y** or **yes**, the contents of the current buffer are written to the file whose name is displayed. The buffer is marked 'not changed' so that if no further changes are made to the contents, NE will not attempt to write them again when it is finishing. The **y** response is only valid for the first form of the prompt.

If the reply is **n** or **no** it is assumed that a mistake has been made; NE abandons the current line of commands and issues its standard prompt for another command line.

A reply of the form **to** <*filename*> is a request to write the contents of the buffer to an alternative file. NE attempts to open this file and write the text to it instead of to the file named in the prompt. Provided the file is successfully opened, the buffer is marked 'not changed' and its name and title are changed to the new file name.

If the **save** command is given with a file name as an argument there is no prompting; the effect the same as when a new file name is given in response to the prompt.

20.4 The WRITE command

The **write** command is used to write the contents of the current buffer to a file, without changing the name of the buffer (*cf* **save**). It is always followed by a file name as an argument.

If a line mark has been set (by means of ***mark-line*** on the screen, or via the **mark** command) then only those lines between the marked line and the current line, inclusive, are written to the file. Otherwise the entire contents of the current buffer are written.

The mark may precede, follow, or be on the current line. It is automatically removed by the **write** command unless it has been 'held' by pressing ***mark-line*** twice.

20.5 The BACKUP command

NE does not at present contain any facilities for automatically backup up the state of the editing session.

However, there is a fairly widely established convention for renaming files that are about to be over-written. If the command

```
  backup files on
```

is obeyed, then, when an output file is to be opened for the first time in an editing session, any existing file of that name is renamed by inserting a tilde at the start of its name.

The word 'on' can be replaced by 'off' to turn this facility off during an editing session. If the command is given with neither 'on' nor 'off', then the state of the option is inverted.

21. Editing large files

Normally, NE loads complete files into main memory for editing, making it possible to move to any point in the file at will. On some systems the amount of main memory is limited, and so large files cannot be edited in this manner.

NE contains some simple support for handling such files. A buffer can be marked as a *stream* buffer, in which case it is permitted to contain no more than a certain number of lines. When the number of lines exceeds the maximum, earlier lines are written to the output file. There is no support for rewinding; this is a one-pass scheme.

A command of the form

```
ne <input-file> -to <output-file> -stream [<n>]
```

is used to edit a file in stream mode. The **-to** option is mandatory if **-stream** is given. The number after **-stream** is the maximum number of lines to be held in the buffer at any one time. If omitted, it defaults to 1000. The minimum permitted value is 500.

From within NE, an existing non-stream buffer can be converted to a stream buffer by the command

```
stream <output-file>
```

This sets up the output file for the stream, and writes out any initial lines if the buffer is already over the limit. A stream buffer cannot be turned back into a normal buffer.

The **streammax** command can be used to change the line number limit on an existing stream buffer. It also sets the limit for any buffers that are subsequently made into stream buffers.

The **show buffers** command outputs ‘stream’ instead of giving a line count for stream buffers.

A number of commands are not permitted in stream buffers (for example, **renmumber**). The **save** command is permitted with no argument; it writes the rest of the file to the output, and then deletes the buffer. This makes it possible to load and process files in stream mode without exiting from NE.

When the **-stream** option is used on the command line, the output file is opened early, before any commands have been obeyed. Therefore the use of the **backup** command in an initialization string or the **-opt** option does not affect it. If it is desired to apply the **backup files** option to a stream output file, the **stream** command should be used instead of the **-stream** option.

22. Conditional commands

The commands **if** and **unless** can be used to control whether other commands are obeyed according to the result of certain tests. The syntax is

```
if     <condition> then <command> [[;] else <command>]
unless <condition>   do <command> [[;] else <command>]
```

where <*command*> is either a single NE command or a group of commands enclosed in round brackets. The command or command group may be empty. The **else** part of these commands is optional, and may optionally be preceded by a semicolon. The keywords **then** and **do** are synonymous, and can be used interchangeably.

The conditional commands can be continued over several lines of input, provided that the line breaks occur within round brackets, and between commands. If the entire command is enclosed in round brackets, or if it is being read from a non-interactive source of input, then a line break immediately before **else** is also permitted.

The <*condition*> can take one of the following forms:

- If <*condition*> is a search expression, then this is tested against the current line. If it matches, then <*condition*> is 'true'; otherwise it is 'false'.
- If <*condition*> is the word **eof** then <*condition*> is 'true' if the current line is the end-of-file line.
- If <*condition*> is the word **sof** then <*condition*> is 'true' if the current line is the first line of the file, and the cursor is at its start.
- If <*condition*> is the word **sol** then <*condition*> is 'true' if the cursor is at the start of the current line.
- If <*condition*> is the word **eol** then <*condition*> is 'true' if the cursor is at the end of the current line.
- If <*condition*> is the word **mark** then <*condition*> is 'true' if the current line is marked by the line (***mark-line***) mark.
- If <*condition*> is of the form

  ```
  prompt <delimited string>
  ```

  and NE is running interactively, the user is prompted with the given string, and must reply with **yes**, **y**, **no** or **n**. If the answer is affirmative, then <*condition*> is 'true'; otherwise it is 'false'. If NE is not running interactively, <*condition*> is always 'true'.

When <*condition*> is 'true' the **then** part of an **if** command is obeyed; when it is 'false' the **else** part, if any, is obeyed. For an **unless** command, the opposite occurs.

Here are some examples of conditional commands:

```
if mark then break;

if eof then (comment/Reached EOF/; stop)

if /cat/ then (
  comment/It’s a cat/
  ) else (
  comment/It’s not a cat/
  )

unless (/pig/ & n/brown/) do dline

(if prompt/Shall I?/ then e/something//other thing/
  else comment/OK, I haven’t/)
```

The brackets surrounding the final example would be needed only if it were typed in interactively, to prevent the entire **if** comment being terminated by the line break before **else**. In non-interactive input, these brackets could be omitted, because NE looks ahead to check for **else** before terminating an **if** command at the end of a line.

Note that if more than one command is to be conditionally obeyed, they *must* be enclosed in round brackets. If the brackets were omitted in the second example above, then the **stop** command would always be obeyed as it would not be part of the **if** command.

23. Looping commands

NE contains several commands which can be used to generate loops and change the flow of control within them.

23.1 The WHILE and UNTIL commands

These two commands set up a loop which is obeyed while or until a given condition is true, respectively. Their syntax is

```
while <condition> do <command>
until <condition> do <command>
```

where <*condition*> and <*command*> take exactly the same form as for the conditional commands are described in the previous chapter, except that <*command*> may not be empty.

Because it is such a common case, the pseudo-command **uteof** is provided as an abbreviation for

```
until eof do
```

Thus the following two commands are equivalent:

```
until eof do (f/cat/; b///tle/)
uteof (f/cat/; b///tle)
```

When a loop has **eof** as its ending condition, the loop ends when the current line is the end-of-file line at the testing point (that is, when control is at the first or only command), or when an attempt is made to move beyond the end-of-file line by means of the **f**, **df**, **n** or **cl** commands, or if one of the commands <, >, #, **$**, **%**, ~, **a**, **b**, or **e** is attempted on the end-of-file line. In the examples immediately above, for instance, the end of file is reached during the processing of the **f** command. When this happens, the loop terminates normally, without error.

23.2 The REPEAT command

The **repeat** command takes a single command or a command group as its argument. This is repeatedly obeyed until an error occurs or the loop is explicitly terminated by a **loop** or **break** command.

23.3 The LOOP and BREAK commands

The **loop** command causes a jump to the repeat point of the textually enclosing looping command. The **break** command causes a jump to just beyond the end of the textually enclosing looping command. For example,

```
repeat (n;
        if /last/ then break;
        if /skip/ then loop;
        t1)
```

when applied to a file containing the lines

```
The quick brown fox
of literary repute
was skipping
over the trees
when last seen.
```

causes the following output to be generated:

```
of literary repute
over the trees
```

Both the **loop** and the **break** commands can be followed by a numerical argument which specifies the number of nested loops to be considered. Thus

```
break 2
```

jumps out of two nested loops, for example, while

```
loop 3
```

resumes execution at the repeat point of the third textually enclosing loop. Supplying no argument to either of these commands is equivalent to supplying the argument 1.

24. Information displays

The **show** command in NE is provided for the purpose of displaying information about the current state of the editor. It must be followed by a word indicating the information required. During screen editing, the output appears in the expanded message area at the bottom of the screen. In order to give the user time to read it, NE does not immediately return to screen editing, but outputs either a command prompt, if the command was the last on a command line, or the prompt

```
Press RETURN to continue
```

otherwise. In the latter case, pressing ***enter*** causes NE to proceed to the next command on the line.

24.1 Which are ‘word’ characters?

The command

```
show wordchars
```

causes NE to display the list of characters that are set as being ‘word’ characters.

24.2 Switch settings

The command

```
show settings
```

causes NE to display the state of a number of option settings, such as the eightbit character display option. The values for the readonly and prompt options apply to the current buffer only.

24.3 Contents of a buffer

The command

```
show wordcount
```

displays a line count, word count, and character count for the current buffer. A word in this context is any sequence of characters delimited by one or more spaces or the end of a line. The character count does not include the line separator characters (carriage return and/or linefeed) which are used on some systems to separate lines in files. The same file should produce the same output on all implementations.

24.4 Information about buffers

The command

```
show buffers
```

causes a summary of the current contents of NE’s editing buffers to be output. For each non-stream buffer a line of the form

```
Buffer <n> <m> lines [(modified)] <title>
```

is output, where <*n*> is the buffer number, <*m*> is the number of lines in the buffer, and <*title*> is the title attached to the buffer, which defaults to the file name. The text ‘(modified)’ is included if the contents of the buffer have been changed since it was last saved. Otherwise spaces appear in this position. If there is no name attached to the buffer, the text ‘<unnamed>’ is output in the title position.

If the number of lines is followed by a plus sign, as for example in

```
Buffer 0  22+ lines (modified) &t
```

it means that not all of the file has yet been read into main storage. The number given is the number of lines read so far.

For stream buffers, the information given is

```
Buffer <n>       stream [(modified)] <title>
```

since the total number of lines is not known because only part of the file is held in main memory at any time.

If the cut buffer is not empty, a line of information about it is also output in the format

```
Cut buffer <n> lines [(pasted)]  <type>
```

where <*n*> is the number of lines and <*type*> is one of the strings '<text>' or '<rectangle>'. The text '(pasted)' is included if the cut buffer has been pasted at least once. Otherwise blanks appear in this field.

24.5 Command information

The command

```
show commands
```

causes a multi-column list of NE's command names to be output.

24.6 Keystroke information

The command

```
show keys
```

which is obeyed when ***keystring-9*** is pressed, causes NE to display the definitions of three sets of keystrokes: control keys, extra keys, and function keys. It is equivalent to the three commands

```
show ckeys
show xkeys
show fkeys
```

which are described below. Because the amount of information is typically too great to fit on the screen, NE pauses between each set and outputs the message

```
Press RETURN to continue
```

to give the user a chance to read the output.

24.7 Control keystrokes

The command

```
show ckeys
```

causes NE to display the definitions of all the 'control-type' keystrokes that have any effect. (See *Changing keystroke defaults* for how to change the definitions of certain keystrokes.) These are the keystrokes that are implemented by means of the CTRL key in many cases. In the default state, a short phrase describing the action of each keystroke is given.

24.8 Extra keystrokes

The command

```
show xkeys
```

causes NE to display details of usable keystrokes that are neither 'control-type' nor 'function-type'. Some of these (for example, the DELETE keystroke) may be configurable by the user, while others may be fixed in meaning, depending on the particular implementation of NE.

24.9 Function keystrokes

The command

```
  show fkeys
```

which is obeyed when ***keystring-19*** is pressed, causes NE to display the definitions of all the 'function-type' keystrokes that have any effect. These are the keystrokes that are implemented by function keys in many cases. In the default state, each function keystroke with a number no greater than 20 causes the corresponding function keystring to be obeyed as a line of commands. These strings are shown in double-quote marks in the output of this command.

24.10 Keystroke actions

The command

```
  show keyactions
```

causes NE to display a list of action abbreviations which can be bound to keystrokes by means of the **key** command. A short description is given with each one.

24.11 Function keystrings

The command

```
  show keystrings
```

causes NE to display the definitions of all the function keystrings that are set. Function keystrings should not be confused with 'function-type' key*strokes* – they are explained in detail in the chapter entitled *Changing keystroke defaults*.

25. Keyboard interruptions

NE can normally be interrupted from the keyboard during its operation by the conventional **ctrl/c** keystroke. The logical keystroke name ***interrupt*** is used as a general name for whatever is needed to generate an interruption.

25.1 The effect of an interruption

If NE is running non-interactively, a keyboard interruption causes the NE run to be abandoned with an error return code.

If NE is accepting screen-editing keystrokes the interruption has no effect.

If NE is in the middle of reading a logical line of interactive commands which consists of several lines of input (using round brackets to indicate a continuation), a keyboard interruption can be used to abandon the entire logical line, once the first actual line has been read. Before the first line is complete, the usual keystrokes for cancelling an input line can be used to abandon it.

If NE is obeying a line of commands, it interrupts its processing at the end of the current command and outputs the message

```
** Keyboard interrupt
```

It then waits for a new line of commands to be typed in.

25.2 Interruptable commands

Normally, NE completes the current command before checking for a keyboard interruption. However, there are some commands that can be interrupted in the middle of their processing. They are

f bf df ga gb ge i repeat t tl until while

Keyboard interrupts are also recognized while scanning through large files as a result of the ***scroll-bottom*** keystroke, and also while obeying the command `M*` on a partially-read input file.

The looping commands stop as soon as one of the commands in the argument group is halted by an interruption. The others stop after a complete cycle of their action. For an interactive **i** command, a keyboard interruption is equivalent to a terminator line. After an interrupted **f**, **df**, or **bf** command, the current point in the file is unchanged.

The behaviour of NE in regard to keyboard interruptions when it is running another program as a result of a command line beginning with ∗ is system-dependent. However, it is normally the case that unless the called program deals with the interruption itself, it is aborted, and control returns to NE.

25.3 The ATTN command

When setting up procedures or sequences of commands to be obeyed in an NE session it is sometimes desirable to inhibit the taking of interruptions throughout a particular group of commands. For example, if there is a command sequence which updates two files, it may not be desirable for it to be aborted when only one file has been updated.

The NE command **attn off** suppresses NE's checks for keyboard interruptions until cancelled by the command **attn on**. ('Attn' is an abbreviation for 'attention'.) If an interruption occurs in the disabled state it is not forgotten, and will be taken immediately the next **attn on** command is obeyed. Thus, 'critical sections' of NE commands should be bracketed with **attn** commands:

```
attn off
<update first file>
<update second file>
attn on
```

However, to give a means of escape from erroneous command sequences, NE always notices a *second* interruption, even when **attn off** has been obeyed.

26. Leaving NE

The **stop** command causes an immediate abandonment of the NE session. No files are written, and the contents of all the editing buffers are lost. A return code indicating an error exit is passed back to the operating system.

It is easy to forget that you are editing more than one thing at once, and using **stop** to give up on one file can have the effect of losing data from another. For this reason, when **stop** is obeyed interactively, a check is made to see if any buffers other than the current one have been modified but not saved. If one such buffer is found, a message of the form

```
  Buffer <n> (<title>) has been modified but not saved.
```

is output, while if more than one is found, the message is

```
  Some buffers have been modified but not saved.
```

This is followed by the prompt

```
  Continue with STOP command (Y/N)?
```

This check is disabled if the **warn** command has been used to disable warnings. In addition, there is a second command, **abandon**, which acts like **stop**, but which bypasses the checks and never prompts.

The **w** (windup) command is the normal way to terminate NE. (The ***keystring-3*** keystroke causes a **w** command to be obeyed.) It causes NE to scan through the editing buffers in turn, checking for any that are marked 'changed'. Each such buffer is a candidate for being written back to its associated file. However, if NE is running interactively, a prompt is output and the user is given an opportunity to prevent this writing back, or to specify an alternate file name.

Before scanning the buffers, NE checks to see whether there is text in the cut buffer that has never been pasted. If such unpasted text is found, the message

```
  The contents of the cut buffer have not been pasted.
```

is output, followed by up to three lines from the start of the cut buffer. If NE is running interactively, this is followed by the prompt

```
  Continue with W command (Y/N)?
```

If the user's reply is **y** or **yes**, the **w** command proceeds and the contents of the cut buffer are lost. If the reply is **n** or **no**, the **w** command is abandoned, and NE issues a new command prompt.

When NE is running non-interactively, the warning message is output, but of course there is no opportunity of saving the contents of the cut buffer.

The current buffer is considered first. If it has been changed (since the last **save** command) and NE is running interactively, the prompt

```
  Write [buffer <n>] to <buffer title>?
                          (Y/N/TO filename/Discard/STOP)
```

is output, where <*buffer title*> is the title associated with the buffer (defaulting to the file name). The text 'buffer <*n*>' is omitted if only one buffer exists and its number is zero (the very simplest case).

If there is no file name associated with the buffer, the prompt takes the form

```
  Write [buffer <n>]? (N/TO filename/Discard/STOP)
```

The user must give one of the five possible replies, terminated by ***enter***. If an empty or erroneous line is entered, an error message is output, and the prompt is repeated.

If the reply is **y** or **yes** then the contents of the buffer are written to the named file. This reply is only valid for the first form of the prompt.

If the reply is **n** or **no**, a mistake is assumed to have occurred. NE abandons the **w** command at this point, and issues a prompt for a new line of commands.

A reply of the form **to** <*filename*> is a request to write the contents of the buffer to an alternative file. NE attempts to open this file and write the contents of the buffer to it instead of to the file named in the prompt. If the file is successfully opened, it becomes the associated file for the buffer. This is only relevant, of course, if there is a subsequent error that prevents the **w** command from completing.

If the reply is **d** or **discard** then the contents of the buffer are not written, but no error occurs. NE proceeds normally to consider the next buffer, if any.

If the reply is **stop** NE does not write the current buffer. It abandons the **w** command, and obeys a **stop** command instead (see the description of **stop** above).

The prompting sequence that has just been described can be suppressed by obeying the command **prompt off** while the buffer is current. In this case, provided there is a file name associated with the buffer, its contents are always written out without comment (assuming they have changed, of course). However, if the buffer has no file name, prompting always takes place.

If the current buffer is not marked 'changed', then one of the following messages is output, as appropriate:

```
No changes made to <buffer title>
No changes made to <buffer title> since last SAVE
```

If the buffer has no title, then the text 'buffer <*number*>' is used instead. NE then considers all the other editing buffers, in turn. For any that are marked 'changed', the prompt

```
Write buffer <n>
                      to <buffer title>? (Y/N/TO filename/Discard/STOP)
```

is output, and the responses are as described above. If there is no file name associated with the buffer, the prompt is changed in the same manner as for the current buffer. No output is produced for any buffers that are not marked 'changed'.

If NE completes its scan of all the editing buffers without any error or negative responses to the prompts, it returns control to the operating system with a return code indicating a successful run.

If there is an error while attempting to open or write to a file, or if the user replies **no** to any prompt, NE does not exit, but prompts for a line of user commands. At this point, the buffer which was being considered at the time of the error has become the current buffer. Any NE commands can be entered; in particular, if the error was due to a mis-spelled file name, the **w** command can be retried. Prompting restarts with the new current buffer; any buffers that were successfully written out before the error are no longer marked 'changed' and so do not appear in this second prompt sequence.

27. Changing default operations

A number of commands are provided for changing the way in which NE performs certain operations.

27.1 The AUTOALIGN command

The **autoalign** command is used to specify how split lines are to be aligned.

```
  autoalign on
```

specifies that whenever a line is split (whether by means of the **sa** or **sb** commands, or by the ***split-line*** keystroke, or as a result of power typing), the newly created line must be aligned with the previous line. For example, if the line

```
        pack my box with five dozen liquor jugs
```

is split after the word ‘five’, the result is

```
        pack my box with five
        dozen liquor jugs
```

In the default state, which can be restored by the command

```
  autoalign off
```

the result of such a splitting would be

```
        pack my box with five
  dozen liquor jugs
```

Autoalign may also appear without an argument, in which case NE switches to the opposite state to that which is current. When automatic alignment is enabled, the letter ‘A’ is added to the three mode letters that are displayed in the lower separator line on the screen.

27.2 The CASEMATCH command

By default NE performs character comparison operations in a case-independent manner when matching a qualified string to a line. That is, upper-case (capital) letters are considered to be equal to their lower-case (small) equivalents. This action can be overridden by the use of the **v** (verbatim) qualifier on an individual qualified string, but it is also possible to change the default by means of the **casematch** command.

```
  casematch on
```

specifies that the cases of letters must be identical for a match to take place, while

```
  casematch off
```

restores the initial situation. The current state of the case matching switch is displayed in the lower separator line on an NE screen, as one of the letters ‘U’ or ‘V’. **Casematch** without an argument switches to the other state from whatever state is current.

When case matching has been selected, it is possible to overrule it for an individual qualified string by the use of the **u** (uncased) qualifier. The **v** or **u** qualifier can always be used to specify exactly what is wanted for a particular matching operation, overriding whichever default is selected by **casematch**.

Neither the case matching default, nor the **v** or **u** qualifiers have any effect on characters that are specified in hexadecimal, whether by means of the **x** qualifier or by the use of **$h** in a regular expression.

27.3 The CUTSTYLE command

When a portion of text or a rectangle is cut or copied to the cut buffer (see *Cutting and pasting*) it normally replaces any previous data in the buffer. Sometimes it is convenient to arrange for the new data to be added to the end of the cut buffer instead. This is specified by the command

```
  cutstyle append
```

When this option is in effect, there is no warning when additional material is added to an unpasted cut buffer. The default action can be restored by the command

```
  cutstyle replace
```

Cutstyle without an argument selects the opposite action from whatever is currently selected. The current state of this switch is displayed in the lower separator line on an NE screen, as one of the letters ‘A’ or ‘R’.

When new data is being appended, an explicit command, **dcut**, must be obeyed in order to empty the cut buffer. This command can also be obeyed in the replacement state, though it is not normally necessary.

If several rectangles are joined together by cutting them when in the appending state, they are placed one below the other, and the resulting rectangle has the width of the widest of its constituents. Its depth is the sum of the depths of the constituents.

When text blocks and rectangles are joined together, the type of the result is the type of the last item cut or copied. If, for example, a rectangle is appended to a text block, any subsequent paste operation pastes the data as though it were a rectangle. The width is the greater of the width of the rectangle and the longest line in the text.

27.4 The EIGHTBIT command

By default, NE is very conservative in choosing which characters it is prepared to display in screen mode. It has stuck to the printable ASCII characters with codes in the range 32–126, and displayed any other data characters as question marks. In some environments, however, so-called ‘top-bit-set’ or ‘eightbit’ characters with codes in the range 160–255 are meaningful, and are used for additional graphics.

The **eightbit** command controls whether NE displays these additional characters or not. It can be called with either of the argument words ON or OFF; if called with no argument, it changes to the opposite state. The default setting is OFF.

27.5 The OVERSTRIKE command

NE normally operates in ‘insert mode’ when screen editing is taking place. This means that any character typed by the user is inserted into the current line at the cursor position, and the remainder of the characters in the line are moved to the right by one position for each character so inserted.

Occasionally it is useful to be able to operate in ‘overstrike mode’, where characters typed by the user *replace* whatever is already in the line at the cursor position. The **overstrike** command is used to select this state; its argument can be one of the words **on** or **off**, or it can be given without an argument, in which case it changes to the opposite state. The current state is displayed in the separator line at the bottom of the screen as one of the letters ‘I’ (for insert) or ‘O’ (for overstrike).

The keystroke **ctrl/o** is by default set up to obey the **overstrike** command, via function keystring 60, in most implementations of NE. Pressing this key has the effect of swapping between insert and overstrike modes.

When operating in overstrike mode, all control keystrokes have their usual effects – ***split-line*** still splits the line, ***delete-here*** still closes up the line, and so on.

27.6 The PROMPT command

The **prompt** command sets and resets a flag in the current buffer which causes the prompting that normally happens when a **dbuffer**, **load**, **save** or **w** command is obeyed to be suppressed. It takes as its argument one of the words ‘on’ or ‘off’. When prompting is switched off, the buffer is always written out as if an affirmative reply to the prompt had been given.

In the case of the **save** and **w** commands, if the current buffer does not have an associated file name then prompting is not suppressed, even if **prompt off** has been obeyed.

This facility is useful when constructing command procedures which use NE as a data input mechanism. It is tedious for the user to have to reply to NE’s request to write to some temporary file (used internally by the procedure, and therefore not of significance to the user) as well as to the command procedure’s own prompt.

If there is any error while attempting to write the buffer to a file (non-existent file, overlong lines, etc.) then the prompting flag is automatically turned on so that prompting will occur if an attempt is made to write the file again.

27.7 The READONLY command

The **readonly** command sets and resets a flag in the current buffer which prevents any updating of the buffer. It takes as its argument one of the words ‘on’ or ‘off’; if called without an argument the state of the flag is inverted. Any new buffer that is created when a read-only buffer is current is also marked read-only. The **-readonly** command line option can be used to set the read-only flag for the first buffer.

When NE is operating as a screen editor, the first status letter in the lower separator line is set to ‘R’ for read-only buffers (as opposed to ‘I’ for insert mode or ‘O’ for overstrike mode).

27.8 The SET command

The **set** command is used to change the values of parameters that control the way NE behaves. The following are available:

Set autovscroll <*n*> sets the number of lines of vertical scrolling that occur when the cursor hits the top or bottom of the screen. The default value is system-dependent, but is normally one.

Set splitscrollrow <*n*> controls NE’s behaviour when splitting lines on the screen. When a line is split as a result of user input, there is a choice between scrolling the upper part of the screen upwards, or the lower part downwards. By default, NE always scrolls downwards except when the cursor is within five lines of the bottom of the screen. The argument <*n*> is a number which specifies the number of lines from the *bottom* of the screen at which the change of scrolling is to take place. The default is equivalent to

```
  set splitscrollrow 5
```

The argument cannot be less than one, and it if is greater than the number of lines on the screen, the change takes place on the second row.

Set oldcommentstyle causes NE to recognize a single backslash character as introducing comments in command lines, as did its prececessor, the E editor. **Set newcommentstyle** restores the default, which requires two successive backslashes. Changing the style does not take effect until the following line of commands is read.

27.9 The WARN command

The **warn** command disables and enables various NE warning messages. It can be followed by one of the words **on** or **off**; if it appears with no argument it causes NE to change to the opposite state.

By default, warnings are enabled. **Warn off** suppresses warnings that are otherwise given in the following circumstances:

- When a **cut** operation is attempted and there is unpasted material in the cut buffer.
- When termination of NE is attempted and there is unpasted material in the cut buffer.
- When a **load** command is obeyed and the contents of the current buffer have been changed and not yet saved.
- When a **dbuffer** command is obeyed and the contents of the current buffer have been changed and not yet saved.

With respect to buffers, **warn off** acts as a global **prompt off** command. Wherever NE would have prompted for confirmation (e.g. before saving a file or before obeying **stop** with some unsaved buffers), it behaves as if an affirmative response to the prompt has been given.

27.10 The WORD command

The **word** command allows the user to change the definition of what constitutes a ‘word’, as used in qualified strings via the **w** qualifier and by the ***word-left*** and ***word-right*** keystrokes. It takes a single string as an argument. The string contains those characters that are to be considered as forming ‘words’. Ranges of letters or digits can be specified by using the hyphen character. If a hyphen itself is required in a word, then the character must be preceded by a double quote character in the string. This rule also applies to the double quote character itself. The default state is equivalent to

```
word /a-zA-Z0-9/
```

Note that it is necessary to specify upper and lower case letters explicitly.

28. Keystroke commands

There are a number of commands which perform the same actions as the keystrokes that are available when screen editing. The commands are provided for use when editing non-interactively, or for building up complicated sequences in function keystrings or procedures. Full details of the actions of the associated keystrokes are given in chapter 3, and are not repeated here.

28.1 The MARK command

The **mark** command must be followed by one of the words **limit**, **line** (or **lines**), **text**, **rectangle** or **unset**. It has the effect of setting the appropriate mark (**limit** sets the 'global mark') at the current position, or unsetting it if **unset** is given. However, if a mark of any sort is already set, an error occurs.

After **mark line** the word **hold** can appear. This sets the line mark in such a way that it is not automatically deleted after being used in a bulk line operation other than 'delete line'. It is equivalent to pressing ***mark-line*** twice while screen editing.

28.2 The CSU and CSD commands

These commands have the effect of moving the current point (cursor) up or down one line, respectively, without changing its horizontal position in the line.

28.3 The ALIGN command

This command operates on the current line, and has exactly the same effect as the equivalent keystroke; it aligns the line so that the first printing character is at the position of the current point. If a line mark is set, the command operates on the sequence of lines delimited by the mark and the current line, inclusive.

28.4 The CLOSEBACK and CLOSEUP commands

These commands operate on the current line, and have exactly the same effect as the equivalent keystrokes:

- **Closeback** closes up the line by removing spaces immediately prior to the current point.
- **Closeup** closes up the line by removing spaces at the current point.

If a line mark is set, these commands operate on the sequence of lines delimited by the mark and the current line, inclusive.

28.5 The DLEFT and DRIGHT commands

These commands operate on the current line, and have exactly the same effect as the equivalent keystrokes:

- **Dleft** deletes all characters in the line to the left of the current point.
- **Dright** deletes the character at the current point and all those to the right of it in the current line.

If a line mark is set, these commands operate on the sequence of lines delimited by the mark and the current line, inclusive.

28.6 The DLINE command

This command deletes the current line, or a sequence of lines from the line mark to the current line. The current point is left at the start of the line following the last deleted line. In this respect it differs from the ***delete-line*** keystroke.

28.7 The DTWL and DTWR commands

The **dtwl** command deletes characters to the left in the current line, starting at the character preceding the cursor position, and ending with the character at which a ***word-left*** cursor-moving operation would stop. In other words, it deletes back to the start of the previous word.

The **dtwr** command deletes characters to the right in the current line, starting from the character at the cursor, and ending at the last character before the next position in which a ***word-right*** cursor-moving operation would stop. In other words, it deletes everthing from the current position to the start of the next word.

Note that, unlike the ***word-left*** and ***word-right*** keystrokes, these operations never move to a different current line. If issued at the start of end of a line (respectively), they have no effect.

28.8 The CUT and COPY commands

These commands perform the same operations as the equivalent keystrokes.

28.9 The PASTE command

When given without an argument, this command performs the same operation as the ***paste*** keystroke, that is, the contents of the cut buffer are pasted into the current buffer at the cursor position. However, **paste** may optionally be followed by a buffer number, in which case the insertion of the data takes place in the buffer of that number instead of the current buffer.

28.10 The DMARKED command

This command deletes the text or rectangle delimited by the text or rectangular block marker and the current point.

28.11 The ISPACE command

This command inserts a rectangle of spaces into the text. The rectangle is defined by the rectangular mark and the current point.

29. Changing keystroke defaults

The screen editing operations of NE have been described in terms of logical keystrokes such as ***delete-previous*** and ***scroll-bottom***. The relationship between these keystrokes and actual keypresses is a two-stage one.

The main part of NE, which is independent of any particular terminal or operating system, relates the logical keystrokes to idealized ‘actual control keystrokes’. These are divided into three groups:

(1) ‘Control-type’ keystrokes are those that are commonly produced with the aid of the CTRL key; there are 31 of these, named **control-a** to **control-z**, **control-[**, **control-**, **control-]**, **control-^**, and **control-_**.

(2) ‘Function-type’ keystrokes are those that are commonly produced with the aid of function keys.

(3) The remaining ‘actual control keystrokes’ are produced by dedicated keys or are artifacts of the terminal handlers. (An example of the latter is a ‘keystroke’ that is manufactured if a character is typed at the right-hand margin.)

The relationship between ‘actual control keystrokes’ and keypresses on the terminal’s keyboard is a function of the particular terminal driver which is being used. The terminal-specific chapters of this document give details for the many different cases.

The relationships between ‘actual control keystrokes’ and logical keystrokes are not all fixed within NE, and most of them can be changed by the user to taste. Two different kinds of action can be specified for an individual ‘actual control keystroke’.

- A single, primitive editing action, which is built into NE, may be performed. Examples are moving the cursor to the left of the screen, or deleting a line. This occurs when the ‘actual control keystroke’ is bound to a logical keystroke other than ***keystring-1*** – ***keystring-60***.
- An arbitrary line of NE commands may be obeyed. This is achieved by binding to one of ***keystring-1*** – ***keystring-60***, and thereby associating the ‘actual control keystroke’ with one of sixty *function keystrings* that are stored by NE.

29.1 Function keystrings

The contents of any of the function keystrings strings can be changed by the command

```
fkeystring <n> <string>
```

where <*n*> is a number between 1 and 60 inclusive. Because this command is frequently used in initialization sequences for NE, which are limited in length by some operating systems, the abbreviation **fks** is provided. For example, to set up keystring 15 to create a new editing buffer, the command

```
fkeystring 15 "newbuffer"
```

could be used. If no text string is supplied, the keystring becomes unset. The default contents of the function keystrings are shown in the following table. Those that are not listed are empty.

1	"buffer"	16	"plr"
3	"w"	17	"bf"
4	"undelete"	18	"m0"
6	"pll"	19	"show fkeys"
7	"f"	20	"format"
8	"m*"	58	"topline"
9	"show keys"	59	"back"
10	"rmargin"	60	"overstrike"

The command

```
show keystrings
```

can be used to display the contents of those function keystrings that are set. The line of commands for each keystring is displayed inside quotes.

29.2 Keystroke binding

The **key** command is used to specify keystroke bindings, that is, to determine the relationship between ‘actual control keystrokes’ and logical control keystrokes.

The command is followed by one or more definitions, separated by commas. Each definition consists of a key identification and a key definition, separated by an equals sign. The key identification is either

(1) A number in the range 1–30, signifying one of the function-type keystrokes.

(2) A single letter or one of the characters ‘`^`’, ‘`\`’, ’`]`’ or ’`_`’, signifying one of the control-type keystrokes.

(3) One of the following names, signifying the corresponding special keystroke:

`up`	up arrow key
`down`	down arrow key
`left`	left arrow key
`right`	right arrow key
`delete`	delete (or del) key
`del`	delete (or del) key
`backspace`	backspace key
`bsp`	backspace key
`return`	return key
`ret`	return key
`tab`	tab key
`insert`	insert key
`ins`	insert key
`home`	home key
`pageup`	page up key
`pup`	page up key
`pagedown`	page down key
`pdown`	page down key
`end`	end key
`copy`	copy key

(4) One of the above names preceded by either or both of ‘`s/`’ (signifying ‘shift’) or ‘`c/`’ (signifying ‘ctrl’), for example

`s/up`	up arrow key with SHIFT
`s/c/ins`	insert key with SHIFT and CTRL

Only a subset of the special key names is available in any given implementation of NE, depending on the terminal driver which is being used. In particular, some systems do not recognize the use of SHIFT and CTRL in conjunction with the arrow keys.

These key names refer to keys that are available *in addition* to the keys on the keyboard used with CTRL. In some versions of NE, for example, **return** cannot be distinguished from **ctrl/m**, and in these cases the key names ‘`ret`’ and ‘`return`’ are not available.

Changing the meaning of the **return** key, either as a synonym of **ctrl/m** or as a separate key, applies only while editing the main text window on the screen. When a line of NE commands is being entered, **return** always terminates it, that is, it acts as the logical ***enter*** key.

The second part of each definition, which defines which logical keystroke is to be bound, is either

(1) A number in the range 1–60, signifying the corresponding ***keystring-n*** logical keystroke; *or*

(2) One of the following mnemonics, signifying the corresponding logical keystroke.

mnemonic	*logical key*	*action*
`al`	***align-line***	align with cursor
`alp`	***align-previous***	align with previous line
`cat`	***concatenate***	concatenate with previous line
`cl`	***close-up***	close up to the right
`clb`	***close-back***	close up to the left
`co`	***cut-copy***	copy to cut buffer
`csd`	***cursor-down***	cursor down
`csl`	***cursor-left***	cursor left
`csle`	***end-line***	cursor to true line end
`csls`	***start-line***	cursor to true line start
`csnl`	***newline***	cursor to next line (start)
`csr`	***cursor-right***	cursor right
`cssbr`	***bottom-right***	cursor to screen bottom right
`cssl`	***screen-left***	cursor to screen left
`csstl`	***top-left***	cursor to screen top left
`csptb`	***previous-tab***	cursor to previous tab stop
`cstb`	***next-tab***	cursor to next tab stop
`cstl`	***first-char***	cursor to text left on screen
`cstr`	***last-char***	cursor to text right on screen
`csu`	***cursor-up***	cursor up
`cswl`	***word-left***	cursor move left by a word
`cswr`	***word-right***	cursor move right by a word
`cu`	***cut-delete***	cut to cut buffer
`dal`	***delete-left***	delete all chars to left
`dar`	***delete-right***	delete all chars to right
`dc`	***delete-here***	delete character
`de`	***delete-marked***	delete marked text
`dl`	***delete-line***	delete line
`dp`	***delete-previous***	delete previous character
`dtwl`	***delete-to-word-left***	delete to word left
`dtwr`	***delete-to-word-right***	delete to word right
`gm`	***mark-global***	global mark
`lb`	***mark-line***	line(s) begin (mark lines)
`pa`	***paste***	paste
`rb`	***mark-rectangle***	rectangle begin
`rc`	***read-command***	read command line
`rf`	***refresh***	refresh screen
`rs`	***rectangle-spaces***	rectangle space insert
`sb`	***scroll-bottom***	scroll to bottom of file
`sd`	***scroll-down***	scroll down
`sl`	***scroll-left***	scroll left
`sp`	***split-line***	split line
`sr`	***scroll-right***	scroll right
`st`	***scroll-top***	scroll to top of file
`su`	***scroll-up***	scroll up
`tb`	***mark-text***	text begin (mark text)

The **cswl** and **cswr** actions (move left or right by one word) make use of the user-settable definition of a ‘word’ [WORD].

The ‘concatenate’ action causes the current line to be joined on to the previous line, and places the cursor at the first character after the join. This action is not normally bound to anything, since the same action is obtained by pressing ***delete-previous*** when the cursor is at the start of a line.

Some examples of changing keystroke functions are now given. The command

```
key a=dl,c=dc,p=cssl
```

sets up **control-a** as ***delete-line***, **control-c** as ***delete-here***, and **control-p** as ***screen-left***. The previous actions of these keystrokes become unavailable. However, if any other keystroke was set up with one of these actions, it retains its setting. That is, it is possible to have the same action performed by more than one keystroke. The next example,

```
key f=7,g=17
```

causes the keystrokes **control-f** and **control-g** to result in NE's obeying the function keystrings 7 and 17 respectively, so with the default setting for these, **control-f** would perform the 'find next' operation and **control-g** the 'find previous' operation.

Some users prefer to use the **return** key to move to the start of the next line instead of splitting the current line, and others are used to systems where the **delete** key deletes the character at the cursor, instead of the one before it. The next example shows how these requirements can be accomodated:

```
key m=csnl,del=dc,h=dp,4=sp
```

sets **control-m** (equivalent to **return** on many terminals) to ***newline***, the **delete** key to ***delete-here***, **control-h** (**backspace** on many terminals) to ***delete-previous***, and function key 4 to ***split-line***.

The current definitions of the control-type keystrokes, any 'extra' keystrokes, and the function-type keystrokes can be displayed by the commands

```
show ckeys
show xkeys
show fkeys
```

There is also a composite command

```
show keys
```

which is equivalent to obeying each of the previous three in turn. Primitive actions are displayed as short descriptive phrases, while lines of NE commands are displayed in double quotes. When a control-type key is bound to a function keystring, or when a function-type key is bound to a function keystring of a different number, the number of the keystring is shown in brackets before its contents.

30. Calling other programs

When NE is running interactively, it is possible to call other programs without leaving NE. A command line beginning with an asterisk indicates that the remainder of the line is to be handed to a shell for interpretation as an operating system command where possible.

If screen editing is in progress, NE exits from its screen-handling mode, the command line is reflected, and line-by-line mode is temporarily selected while the command is being obeyed. This means that, when reading a subsequent line of NE commands afterwards, the normal line-by-line input conventions are in force instead of those used when in screen mode, and NE's command line history mechanism is not available. To indicate this, the prompt that is given in these circumstances is `NE:` instead of `NE>`.

When control returns to NE, a command prompt is output if editing is interactive and automatic verification is not on. Otherwise the current line is verified in the normal way. Any line of NE commands, or a further system command line (beginning with an asterisk) may now be entered. An empty line causes NE to revert to normal running, and to re-display the screen if screen editing was in progress before the first system command line was entered.

However, if NE is in the middle of obeying a command file as a result of obeying a **c**, **cbuffer**, or **cdbuffer** command when the system command is encountered, the prompt

```
  Press RETURN to continue
```

is issued instead of a command prompt. Pressing RETURN causes execution of the command file to continue.

When screen editing is in progress, a command line consisting of an asterisk only (that is, a null system command) has the effect of putting NE temporarily into line-by-line mode, but without any program to run, so it just issues a command prompt. In environments where a 'scrolling screen' is preserved over a screen-editing session, this is an easy way of referring back to the scrolling screen. Pressing RETURN returns NE to screen editing.

31. Error handling

Most errors are detected while NE is interpreting or obeying a command. Typical examples are a mis-spelt command name or the failure to find a match for a search command.

NE reads an entire command line and converts it into an internal format before obeying it. Therefore, if there is a syntax error in any command in the line, none of its commands are obeyed. When diagnosing a syntax error, NE indicates the point it has reached in scanning the line by printing a > character underneath the line of commands.

If an error in a command line is detected during screen editing, the error message appears in the message area at the bottom of the screen, which expands to accommodate it, thus leaving the command line still visible. After outputting the message, NE displays its standard prompt for reading a new line of commands. This has the effect of keeping the message lines on the screen for the user to read. The user can either

- Press ***enter*** (i.e. enter a null command line) to revert to screen editing;
- Enter a new line of commands;
- Press ***cursor-up*** to recall the previous command line into the input area, edit it, and then press ***enter*** to obey the revised line.

A few errors are so serious that NE cannot continue processing. These include a number of internal consistency failures and any kind of system-detected crash. In these circumstances, NE attempts to save the data from the editing session by writing the contents of all buffers that are marked ‘changed’ to an *emergency file*, whose name is **NEcrash**. The message

```
** Attempting to write data to <file name>
```

is output, followed by one of the following messages for each buffer:

```
** <n> lines written from the cut buffer
** <n> lines written from buffer <n> (<buffer title>)
** No changes made to buffer <n> (<buffer title>)
```

The contents of the cut buffer are written out only if it has not been pasted. The text ‘from buffer <*n*>...’ is omitted if there is only one buffer in existence, and the phrase ‘since last **save**’ may be added to the last message if relevant. The data is written to the emergency file with no separators between the lines of the various buffers.

NE also writes error messages to a log file called **NEcrashlog** when it crashes, to help in tracking down the cause of the crash.

32. Line-by-line editing

NE can be run as a line-by-line editor, both interactively and non-interactively. The majority of this document assumes that screen editing is used when NE is run interactively. However, interactive line-by-line editing is necessary if, for example, the terminal being used does not support full-screen handling. In this circumstance, NE should automatically start up in line-by-line mode. The arrangements for this are dependent on the operating system.

32.1 Interactive line-by-line editing

To force NE to operate line-by-line, the option **-line** can be given on the NE command line.

When operating interactively line-by-line, NE normally verifies the current line before reading the next line of commands, and does not output any prompt string. This makes it easy to use the single character commands, as each single character typed in the command line is exactly below the character of the current line which it will affect. More details are given in chapter 9 above.

32.2 Non-interactive line-by-line editing

Non-interactive use of NE is always in line-by-line mode. In this case, verification of the current line before each line of commands is not the default. A non-interactive run is normally initiated by specifying a **-with** item on the NE command line, giving the name of a file of NE commands to be obeyed. For example,

```
  ne myfile -with myedits
```

When there is no interactive input, the default right-hand margin is 79 – this affects the **format** command only. Reaching the end of the command file is equivalent to obeying a **w** command in NE. It causes any modified buffers to be output, and NE to exit normally.

If any error occurs when NE is running non-interactively, an error message is output to the verification file and the run is abandoned with an error return code.

32.3 Verification output

Verification output and error messages are normally sent to the terminal (even in non-interactive runs). A **-ver** keyword is available on the command line to direct this output elsewhere, for example,

```
  ne myfile -with myedits -ver verification
```

NE contains a switch which determines whether the current line is to be verified on the verification file before each new line of commands is read. The initial state of this switch is ‘on’ for interactive runs, and ‘off’ for non-interactive runs. It can be changed at any time by means of the **verify** command, which is followed by one of the words ‘on’ or ’off’. If **verify** is given without an argument, then the verification switch is changed to the opposite state from the one it is in.

There is also a command to cause verification of the current line only. This is the single-character command consisting of a question mark. It may be used at any time, and is described in detail in chapter 9.

32.4 Format of verification output

The format of the output produced by the **?** command and by the automatic verification facility is described with the full description of **?** (see chapter 9). When NE is running interactively and automatic verification is on, it does not output a command prompt in addition to the verification. This is to make it easy to make use of the single-character editing commands.

32.5 The T and TL commands

Automatic verification and the **?** command show only a single line at a time (the current line). The **t** and **tl** commands enable a number of lines to be output at once. Each takes a single number for an argument, or an asterisk character. If a number is given, it is the number of lines which are to be verified; an asterisk means ‘type until the end of the file’.

The output begins with the current line and continues for the number of lines specified, or until the end of file is reached. The **t** command simply outputs the data in each line, while the **tl** command outputs line numbers at the left-hand side as well. The data is output in the same format as is used for automatic verification and the **?** command – a single line if there are no non-printing characters, two lines otherwise.

The current point is not altered by the **t** and **tl** commands. After obeying one of them in an interactive line-by-line session with automatic verification on, the current line is verified before the next line of commands is read.

These commands can occasionally be useful in screen mode, for showing lines containing non-printing characters. The message

```
Press RETURN to continue
```

is output afterwards to enable the user to read what has been output before returning to screen editing.

32.6 The COMMENT command

The **comment** command takes a string as an argument, and its only effect is to output the string to the verification file.

32.7 The C command

The **c** command takes a file name as an argument. The file is opened for input, and NE reads and obeys lines of commands from it until it is exhausted, or until an error occurs. **C** commands may be nested.

If a file of NE commands is obeyed via a **c** command from an interactive run of NE, any global commands (**ga**, **gb** or **ge**) in it are obeyed in non-interactive fashion, that is, no prompting takes place. However, if it contains any **if** commands with the **prompt** option, prompting still takes place.

If any command in a file obeyed via **c** generates verification output (for example, **comment**, **show**), and the **c** command was issued from a screen editing run of NE, then after such output has appeared in the message area at the bottom of the screen, NE pauses and outputs the message

```
Press RETURN to continue
```

Pressing ***enter*** causes NE to continue with the next command. At the end of the command file, verification output is lost as screen editing resumes.

32.8 The CBUFFER and CDBUFFER commands

These two commands take a buffer number as an argument, and they cause NE to obey the contents of the buffer as a command file. **Cbuffer** leaves the buffer intact, marking it ‘not changed’, so that it can subsequently be overwritten without complaint, while **cdbuffer** deletes the buffer after the commands have been obeyed.

See the **makebuffer** and **paste** commands for details of how to create a buffer with a given number and paste material into it.

33. More details about the Unix interface

This chapter contains nitty-gritty that is probably not of interest to most users.

33.1 Running in screen mode

For NE to work in screen mode, it requires a terminal with sufficient functionality for screen editing to be defined via the *terminfo* or *termcap* mechanism (which of these is used is a compile-time option).

When NE is run in a windowing system under a version of Unix that supports the SIGWINCH signal (most modern systems do) it notices immediately if the size of its window is altered, and adjusts its display accordingly.

33.2 Environment variables

When it starts up, unless the command argument **-noinit** is present, NE searches its environment for a variable with the name `NERC` (in capital letters). The contents of the variable are taken to be the name of a file of NE commands which are obeyed before the **-opt** string. If `NERC` is not defined, NE looks for the file **.nerc** in the directory defined by the environment variable `HOME`.

The environment variable `HOME` is also used in the interpretation of file names that begin with a tilde character.

The environment variable `TERM` is used in determining the terminal type. Details are given in the section on terminal types below.

The environment variable `NETABS` can be used to set up default tab handling options, as described in section 4.2 above.

33.3 Exit codes

The following exit codes are used by NE:

`0`	normal exit
`4`	warning
`8`	errors detected, or **stop** obeyed
`12`	NE ran out of memory
`16`	an internal error was detected
`24`	NE crashed

The only time the warning exit occurs is when a non-interactive run of NE terminates with an unpasted cut buffer in existence.

When NE is run interactively, exit code 8 is generated only as a result of the **stop** command, and exit code 4 is never given.

33.4 Shell commands

The NE '`*`' mechanism can be used to cause shell commands to be executed from within NE. It is a Unix convention that an empty shell escape starts a new interactive shell. However, it is an NE convention that an empty '`*`' command puts NE temporarily into line mode. This is particularly useful on terminals that re-display the scrolling screen when returning to line mode. Therefore a command line such as

```
*/bin/bash
```

is necessary in order to create a new interactive shell.

33.5 Interruptions

In line editing mode, special keystrokes such as the interrupt, suspend and quit characters are handled by Unix, so they have their normal effects.

In screen editing mode, NE traps the interrupt signal generated by the user's interrupt character (typically **ctrl/c**) while obeying NE commands, so this keystroke can be used to interrupt loops or long searches, etc. It also terminates an interactive **i** (insert) command.

During screen editing itself, the keystroke is available as a normal editing keystroke, except when entering lines of commands following ***read-command***, when it can be used to abandon command entry (useful for multi-line commands). This means that, if NE's keystrokes are reconfigured so that the interrupt character is bound to one of the editing functions that is recognized during command entry (e.g. ***delete-here***) then this function is not available during command entry. The default assignment, to ***close-up***, is not used during command entry.

If NE is interrupted in a loop producing output, quite a lot of output can be buffered up and this is not thrown away on receipt of an interruption, so it sometimes takes a while before NE appears to notice an interruption.

33.6 Terminal types

NE supports terminals described in the Unix *terminfo* or *termcap* databases. Which one of these is used is controlled by a compile-time parameter.

If the **-line** keyword is not present on the NE command line, and neither **-with** nor **-ver** is present, NE attempts to start up in screen editing mode. To do this it needs to know what kind of terminal is being used.

- NE interrogates the Unix *terminfo* or *termcap* database to find out the characteristics of the terminal described by the string in the environment variable `TERM`. If the terminal is capable of sustaining a screen editing session, NE initializes its *terminfo/termcap* driver.
- Otherwise a message is output, and NE enters line editing mode.

The configuration required for the terminal or terminal emulator may depend on the communications route being used.

33.7 Terminal capabilities

The following minimum capabilities are required of a terminal in order to support screen editing using NE. These are listed with both their *termcap* and *terminfo* names.

`cm`	`cup`	move to (x,y) on screen
`kd`	`kcud1`	cursor down
`kl`	`kcub1`	cursor left
`kr`	`kcuf1`	cursor right
`ku`	`kcuu1`	cursor up

If the 'cm' or 'cup' string contains '%.' it means that binary values are used for cursor positioning. It is not desirable to generate zero in these circumstances, since it is likely to get swallowed *en route* to the terminal. NE avoids generating binary zeroes by requiring the availability of the 'up' or 'cuu1' control string (cursor up) and either a backspace or the 'bc' or 'cub1' string in this case.

The following optional capabilities are used if present:

`al`	`il1`	add (i.e. insert) line
`ce`	`ed`	clear to end of line
`cl`	`clear`	clear screen
`cs`	`csr`	set up scrolling region
`dc`	`dch1`	delete character – but not if in 'delete mode'
`dl`	`dl1`	delete line
`F1-F9`	`kf11-kf19`	function keys 11-19
`FA-FK`	`kf20-kf30`	function keys 20-30

`ic`	`ich1`	insert character – but not if in ‘insert mode’
`k0-k9`	`kf0-kf9`	function keys 0-9
`k;`	`kf10`	function key 10
`ke`	`rmkx`	end ‘keypad’ mode
`ks`	`smkx`	start ‘keypad’ mode
`se`	`rmso`	end standout mode
`sf`	`ind`	scroll text up
`so`	`smso`	begin standout mode
`sr`	`ri`	scroll text down
`te`	`rmcup`	end use of screen management
`ti`	`smcup`	initiate use of screen management

As well as using the generalised terminal information in *terminfo* or *termcap*, NE contains built-in code to make use of the special capabilities of certain terminals which cannot be described by the *terminfo/termcap* mechanism. Descriptions of these features are given separately below for each relevant terminal. The only relevant terminal that is now supported in this way is:

- An xterm session running under the X windowing system.

The *terminfo/termcap* driver for NE is designed to be usable with a minimal terminal containing only the standard ASCII keys and four arrow keys. The default control keystrokes are set up as shown in the following table.

ctrl/a	*align-line*
ctrl/b	*mark-line*
ctrl/c	*close-up*
ctrl/d	*refresh*
ctrl/e	*cut-copy*
ctrl/f	<*unset*>
ctrl/g	*read-command*
ctrl/h	*scroll-left*
ctrl/i	*next-tab*
ctrl/j	*scroll-down*
ctrl/k	*scroll-up*
ctrl/l	*scroll-right*
ctrl/m	*split-line*
ctrl/n	*mark-global*
ctrl/o	*keystring-60* i.e. "`overstrike`"
ctrl/p	*paste*
ctrl/q	*delete-marked* (but see below)
ctrl/r	*mark-rectangle*
ctrl/s	*rectangle-spaces* (but see below)
ctrl/t	*mark-text*
ctrl/u	*delete-line*
ctrl/v	*delete-right*
ctrl/w	*cut-delete*
ctrl/x	*delete-left*
ctrl/y	*delete-here*
ctrl/z	*align-previous*
ctrl/[	<*escape*>
ctrl/\	*screen-left*
ctrl/]	*unset*
ctrl/^	*keystring-58* i.e. "`top`"
ctrl/_	*keystring-59* i.e. "`back`"

On some communications routes **ctrl/s** and **ctrl/q** are used for flow control and so cannot be passed through to NE. A fudge using **esc** is implemented to get round this – see below.

The **return** key is synonymous with **ctrl/m**. **Delete** provides the *delete-previous* function and the *concatenate* function when used at the start of a line. The cursor keys are used to move around the screen.

If the terminal has function keys described by *terminfo/termcap*, then these are used by NE, with **f0** corresponding to NE's ***keystring-10***. Not all terminals have function keys, and so this facility, together with all other keyboard functions required by NE is also implemented using a keystroke sequence beginning with **esc**.

Many terminals use character sequences starting with **esc** for their special keys, so there is in principle some danger that these may clash with NE's usage. It is hoped that the danger has been minimised in the following choices. When NE receives a sequence of characters starting with **esc** it first searches the list of terminal-specific strings for the functions in which it is interested. Only if the sequence is not in this list does it test for its own built-in interpretations:

<**esc**>**1**–<**esc**>**9**	functions 1–9
<**esc**>**0**	function 10
<**esc**><**esc**>**1**–<**esc**><**esc**>**9**	functions 11–19
<**esc**><**esc**>**0**	function 20
<**esc**>**delete**	ignored (a way to cancel **esc**)
<**esc**>**return**	repaint the screen
<**esc**>**tab**	***previous-tab***
<**esc**><**esc**><*ch*>	enter control char as data
<**esc**>**s**	simulate **ctrl/s**
<**esc**>**q**	simulate **ctrl/q**

If a sequence of two escapes is followed by **del** or a character whose code value is less than 32, that character is interpreted as a data character. Otherwise, unless the character is a digit, the two bits with values 32 and 64 are forced to zero, and the resulting character is handled as a data character. Thus, for example, the data character with binary value one can be inserted into a file by typing **esc** twice followed by 'A' or 'a'.

A keystroke whose value is greater than 127 is always treaded as a data character. Such characters are by default displayed as question marks, but if the **eightbit** command has been obeyed, they are treated as ordinary characters and written to the screen unchanged.

The default assignments to NE's logical keystrokes are shown in the following table. Not all the keystrokes are available on every terminal, and not all of them are definable via the *termcap* or *terminfo* mechanisms.

align-line	**ctrl/a**
align-previous	**ctrl/z**
bottom-right	<*unset*>
close-back	**shift/delete** and **shift/backspace**
close-up	**ctrl/c**
concatenate	**delete** at start of line
cursor-down	**down-arrow**
cursor-left	**left-arrow**
cursor-right	**right-arrow**
cursor-up	**up-arrow**
cut-copy	**ctrl/e**
cut-delete	**ctrl/w**
delete-here	**ctrl/y**
delete-left	**ctrl/x**, **ctrl/del**, and **ctrl/backspace**
delete-line	**ctrl/u**
delete-marked	<**esc**>**q** or **ctrl/q** if not flow control
delete-previous	**delete** and **backspace**
delete-right	**ctrl/v**
end-line	**shift/ctrl/right**
enter	**return**
first-char	**ctrl/left**
interrupt	**ctrl/c** when obeying commands
keystring-n	<**esc**>**1** – <**esc**>**0** for 1–10
	<**esc**><**esc**>**1** – <**esc**><**esc**>**0**
last-char	**ctrl/right**

mark-global	**ctrl/n**
mark-line	**ctrl/b**
mark-rectangle	**ctrl/r**
mark-text	**ctrl/t**
newline	<*unset*>
next-tab	**tab**
paste	**ctrl/p**
previous-tab	<**esc**>**tab** and **ctrl/tab**
read-command	**ctrl/g**
rectangle-spaces	<**esc**>**s** or **ctrl/s** if not flow control
refresh	**ctrl/d**
screen-left	**ctrl/\**
scroll-bottom	**ctrl/down**
scroll-down	**ctrl/j** and **shift/down**
scroll-left	**ctrl/h** and **shift/left**
scroll-right	**ctrl/l** and **shift/right**
scroll-top	**ctrl/up**
scroll-up	**ctrl/k** and **shift/up**
split-line	**return**
start-line	**shift/ctrl/left**
top-left	<*unset*>
word-left	<*unset*>
word-right	<*unset*>

33.8 Keyboard interruptions

While screen editing is occurring, most keystrokes are taken over by NE and used for controlling the editing process, as described above. The only exceptions are any flow control characters that may be in use (typically **ctrl/s** and **ctrl/q**).

However, while NE is obeying a line of commands entered from screen mode, the interrupt keystroke (typically **ctrl/c**) changes its meaning, and if pressed causes an interrupt to be sent to NE. It can thus be used to interrupt a command loop or a long search command.

The many other special keystrokes (kill, literal-next, etc.) are never available during screen editing, the relevant keystrokes retaining their NE meaning. If these facilities are wanted they can be accessed by leaving screen mode, by obeying a command line containing only an asterisk. In the line editing state NE runs with ‘cooked’ terminal input and so all the special keystrokes are available.

33.9 xterm

NE has some built-in knowledge of the xterm terminal emulator which is part of the X windowing system. When the value of the `TERM` variable is ‘xterm’, the internal tables in NE give access to the following additional keystrokes over and above those defined by *termcap* or *terminfo*:

shift/left	*scroll-left*
shift/right	*scroll-right*
shift/up	*scroll-up*
shift/down	*scrown-down*

In addition, the following are also available if the terminal emulator is configured appropriately:

shift/delete	*close-back*
ctrl/delete	*delete-left*
ctrl/left	*first-char*
ctrl/right	*last-char*
ctrl/up	*scroll-top*
ctrl/down	*scroll-bottom*
ctrl/tab	*previous-tab*

The **backspace** key is distinguishable from **ctrl/h**, and it is initially defined to have the same effect as the **delete** key.

The function keys correspond to NE's function keystrokes, and SHIFT may be used with any of them to add 10 to its value.

34. Logical keystroke summary

This chapter lists the logical screen editing keystrokes recognized by NE, preceded by the mnemonics used to specify them in the **key** command (where relevant), and followed by the equivalent NE commands for performing the same operations.

`al`	*align-line*	**align**
`alp`	*align-previous*	**p; pb s//; csd; align**
`cssbr`	*bottom-right*	no equivalent
`clb`	*close-back*	**closeback**
`cl`	*close-up*	**closeup**
`co`	*concatenate*	no direct equivalent; use **cl** on *previous* line
`csd`	*cursor-down*	**csd**
`csl`	*cursor-left*	<
`csr`	*cursor-right*	>
`csu`	*cursor-up*	**csu**
`co`	*cut-copy*	**copy**
`cut`	*cut-delete*	**dcut**
`dc`	*delete-here*	**#**
`dal`	*delete-left*	**dleft**
`dl`	*delete-line*	**dline**
`de`	*delete-marked*	**dmarked**
`dp`	*delete-previous*	no direct equivalent; use **<;#**
`dar`	*delete-right*	**dright**
`dtwl`	*delete-to-word-left*	**dtwl**
`dtwr`	*delete-to-word-right*	**dtwr**
`csle`	*end-line*	**plr**
	enter	not relevant
`cstl`	*first-char*	no equivalent
	interrupt	not relevant
	keystring-n	no equivalent
`cstr`	*last-char*	no equivalent
`gm`	*mark-global*	**mark global**
`lb`	*mark-line*	**mark lines**
`rb`	*mark-rectangle*	**mark rectangle**
`tb`	*mark-text*	**mark text**
`csnl`	*newline*	**n**
`cstb`	*next-tab*	no equivalent
`pa`	*paste*	**paste**
`csptb`	*previous-tab*	no equivalent
`rc`	*read-command*	not relevant
`rs`	*rectangle-spaces*	**ispace**
`rf`	*refresh*	not relevant
`cssl`	*screen-left*	not relevant
`sb`	*scroll-bottom*	**m***
`sd`	*scroll-down*	no direct equivalent; use **n** and **m**
`sl`	*scroll-left*	not relevant
`sr`	*scroll-right*	not relevant
`st`	*scroll-top*	**m0**
`su`	*scroll-up*	no direct equivalent; use **p** and **m**
`sl`	*split-line*	**sa** or **sb**
`cstl`	*start-line*	**pll**
`csstl`	*top-left*	no equivalent
`cswl`	*word-left*	no direct equivalent
`cswr`	*word-right*	no direct equivalent

35. Command summary

∗<*text*>	pass command line to operating system
?	verify current line
<	move cursor one place left
>	move cursor one place right
#	delete character at cursor
$	lowercase character at cursor
%	uppercase character at cursor
~	flip case of character at cursor
a <*se*> <*qstring*>	after <*se*> insert <*qstring*>
align	align line(s) with cursor
attn on	permit keyboard interruptions
attn off	suspend keyboard interruptions
autoalign	flip autoalignment on/off
autoalign on	enable autoalignment for split lines
autoalign off	disable autoalignment for split lines
b <*se*> <*qstring*>	before <*se*> insert <*qstring*>
back	move back to previous change place
backup files	flip output file renaming
backup files on	enable output file renaming
backup files off	disable output file renaming
beginpar <*se*>	define paragraph beginning
bf <*se*>	find backwards
break [<*n*>]	break out of loop
buffer [<*n*>]	select buffer [<*n*>]
c <*file name*>	obey commands from file
casematch on	match letter cases by default
casematch off	do not match letter cases by default
casematch	flip case matching state
cbuffer <*n*>	obey commands from buffer <*n*>
cdbuffer <*n*>	as **cbuffer**, then delete the buffer
centre	centre the current line
cl [<*string*>]	concatenate line with next
closeback	close up line(s) before cursor position
closeup	close up line(s) at cursor position
comment <*string*>	output comment text
copy	copy marked block to cut buffer
cproc <*proc*>	cancel procedure
csd	cursor down one line
csu	cursor up one line
cut	cut marked block to cut buffer
cutstyle append	append copied and cut data to cut buffer
cutstyle replace	replace data in cut buffer each cut or copy
cutstyle	flip append/replace state
dbuffer [<*n*>]	delete buffer [<*n*>]
dcut	delete contents of cut buffer
detrail	remove trailing spaces in current buffer
detrail output	remove trailing spaces on output
df <*se*>	delete lines forwards until <*se*> is found
dleft	delete to the left of the cursor
dline	delete current line
dmarked	delete marked text
drest	delete rest of file
dright	delete to the right of the cursor
dta <*qstring*>	delete till after <*qstring*>

dtb <*qstring*>	delete till before <*qstring*>
dtwl	delete to word left
dtwr	delete to word right
e <*se*> <*qstring*>	exchange <*se*> for <*qstring*>
eightbit	control display of eight-bit characters
else <*cg*>	follows **if** or **unless**
endpar <*se*>	define paragraph end
f <*se*>	find forwards
fkeystring <*n*> <*string*>	set function keystring
fks <*n*> <*string*>	abbreviation for **fkeystring**
format	re-format rest of current paragraph
ga <*se*> <*qstring*>	globally after <*se*> insert <*qstring*>
gb <*se*> <*qstring*>	globally before <*se*> insert <*qstring*>
ge <*se*> <*qstring*>	globally exchange <*se*> for <*qstring*>
i	insert in-line text
i <*file name*>	insert named file
icurrent	insert copy of current line
if <*cond*> **then** <*cg*>	conditional command
iline <*string*>	insert single line before current
ispace	insert rectangle of spaces
key <*data*>	specify key binding(s)
lcl	lower case current line
load <*file name*>	load file to current buffer
loop	restart current command loop
m <*n*>	move to line n (zero means ‘start of file’)
m*	move to end of file
makebuffer <*n*> [<*file name*>]	create new buffer <*n*> [for <*file name*>]
mark limit	set global limit mark
mark line	set line block mark
mark text	set text block mark
mark rectangle	set rectangular block mark
n	move to next line
name <*string*>	set file name
newbuffer [<*file name*>]	create new buffer [for <*file name*>]
overstrike on	overstrike data characters
overstrike off	insert data characters (default)
overstrike	flip overstriking state
p	move to previous line
pa <*se*>	point after context in current line
paste	paste cut buffer in current buffer
paste <*n*>	paste cut buffer in buffer <*n*>
pb <*se*>	point before context in current line
pbuffer	select previous buffer
pll	point to line left
plr	point to line right
proc <*name*> is <*cg*>	define procedure
prompt on	enable prompting for current buffer
prompt off	disable prompting for current buffer
readonly on	make current buffer read-only
readonly off	make current buffer read-write
readonly	invert read-only state of current buffer
refresh	update current screen
renumber	renumber lines in current buffer
repeat <*cg*>	loop of indefinite duration
rmargin	flip right margin on/off
rmargin on	set right margin on
rmargin off	set right margin off

rmargin <*n*>	set margin on, with new value
sa <*se*>	split current line after context
save [<*file name*>]	[rename and] write buffer
sb <*se*>	split current line before context
set autovscroll <*n*>	set automatic vertical scroll amount
set newcommentstyle	double backslash for comments
set oldcommentstyle	single backslash for comments
set splitscrollrow <*n*>	set up/down scroll boundary
show ckeys	display **ctrl** keystrokes
show commands	display command names
show fkeys	display function keystrokes
show keyactions	display key action mnemonics
show keystrings	display function keystrings
show wordcount	show line, word & char count
stop	stop immediately (error return code)
stream <*file name*>	convert to stream buffer
streammax <*n*>	set line limit for stream buffers
t <*n*>	type <*n*> lines
title <*string*>	set title for buffer
tl <*n*>	type <*n*> lines with line numbers
topline	current line to top of screen
ucl	uppercase current line
undelete	restore deleted character or line
unixregexp	interpret regular expressions using Unix syntax
unless <*cond*> **do** <*cg*>	conditional command control
until <*cond*> **do** <*cg*>	loop control
uteof <*cg*>	same as **until eof do** <*cg*>
verify	flip automatic verification state
verify on	enable automatic verification
verify off	disable automatic verification
w	windup (normal exit)
warn	flip warning state
warn on	enable warnings (default)
warn off	disable warnings
while <*cond*> **do** <*cg*>	loop control
word <*string*>	define ‘word’ for **w** qualifier
write <*file name*>	write buffer to <*file name*>

36. Argument format summary

This chapter contains summaries of the formats for the various different kinds of argument that are used by NE commands.

36.1 String delimiters

The available delimiters for string arguments are

```
’ " ! . , : + - * /
```

For file names the delimiter set is a subset of the above, with those characters that are allowed in file names removed. This varies according to the operating system.

36.2 String qualifiers

The following qualifiers are available for modifying the effect of string matching:

B	match at the beginning of the line only
C	match the line’s control character (MVS only)
E	match at the end of the line only
H	match ‘here’ (at current cursor position) only
L	match leftwards in the line
N	negate the result of the match
P	match the line precisely
R	interpret the string as a regular expression
S	ignore leading and trailing spaces
U	match letters in an uncased manner
V	match letters verbatim
W	match the string as a word
X	the string is in hexadecimal
<*n*>	the string must match <*n*> times in the line
[<*n*>,<*m*>]	match between columns <*n*> and <*m*>, inclusive

36.3 Regular expressions

The characters of a qualified string are interpreted as a regular expression if the qualifier **r** is present.

36.4 Regular expression replacements

When a text replacement command has a regular expression as its first argument, the insertion string can be qualified with the **r** qualifier, in which case the character ‘`$`’ in the replacement string is interpreted specially.

`$0`	inserts the entire matched string
`$<`*n*`>`	inserts wild string <*n*>
`$$`	inserts a single percent character
`$<`*x*`>`	inserts the character <*x*>

Any number of wild strings can be matched by a regular expression, but only the first nine can be inserted in this way.

36.5 Hexadecimal insertions

An insertion string can be specified in hexadecimal by qualifying it with the **x** qualifier. This can be in addition to the **r** qualifier for regular expression replacements.

Index