QNX Operating System

QED - Fullscreen Editor

For QNX 4

Printed under license by:

QNX Software Systems Co.
175 Terence Matthews Crescent
Kanata, Ontario
K2M 1W8
Canada
Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: `info@qnx.com`
Web: `http://www.qnx.com/`

Electronic edition published 2005.

Technical support options

To obtain technical support for any QNX product, visit the **Technical Support** section in the **Services** area on our website (`www.qnx.com`). You'll find a wide range of support options, including our free web-based **Developer Support Center**.

Contents

About the QED Manual

The **qed** utility is a fullscreen editor for both your console and attached terminals. This documentation has been divided into several major sections.

Tutorial Guide
: This section consists of a conversational introduction to the Full Screen Editor (**qed**). It contains examples which should be attempted on your PC as you read. It is highly recommended that all users of **qed** read this guide. The first several pages contain a complete reference to all the defined function and cursor keys. It will help you correlate the many functions available. Upon completion of the guide you should be capable of performing most editing tasks. For most users, this may be all you need to know about the editor.

Using **qed** on non-QNX Terminal Types
: This section describes how to configure the system to run **qed** from attached terminals.

Reference Manual — Editor Commands
: This section consists of a detailed description of every command supported by the editor. You will discover that the defined function and cursor keys are in fact implemented as one or more of these commands. Anyone who is going to be using the editor on a regular basis should read the preliminary sections up to the **Append** command. The description of the **Substitute** and **Global** commands are also highly recommended. Should you wish to define your own function key operations then it is imperative that you read and understand *all* sections.

Defining Macros
: This section has been written as a tutorial guide in the writing of macros. It should be sufficient to get you started. However, the best way to learn about macros is to experiment.

Error Messages (Appendix A)
: This section details the error messages you may get from **qed**, and explains their meaning.

Quick Reference
: This section is a quick overview of the function keys which relate to word processing. They are also discussed in the tutorial introduction and are repeated here as a quick reference.

Chapter 1

Tutorial Guide

The full screen editor (**qed**) is a program which allows you to type in text, edit it and save it away in a file. The editor itself treats your text as a series of lines consisting of from 0 to 512 characters. It was designed as a program development editor first and a word processor second.

When operating on text from a file, the editor reads a copy of your file into memory (we will call this a buffer). Any changes you make to your file copy (the buffer) will not affect the original file until you issue a write command to save your buffer. Should you change your mind about saving the particular changes you have made, you may exit without saving, or you may reread a copy of the original file.

The maximum number of lines you can edit will depend on how long each line is and the amount of available memory space. The maximum number of lines in a file that may be edited is on the order of 100 000. Larger files must be split and if necessary rejoined after editing.

Getting Started

Assume you want to enter some text and save it away in a file, use:

```
qed
```

The editor will load from disk, clear your screen, then present a screen that consists of a status line at the top:

```
last=0  (0  ,1)  a+ b- c- d- f- i- j- l- m+ n+ s- t+ w+
```

a line below it for command entry, and a large text area below that.

The top line of the screen is the status line and provides information about the size of your file, where in the file you are currently located, and what editing options you have selected. This line is kept up to date by the editor. You may *not* enter text into this area.

The second line of the screen is the command line. It is the area where you may type editor *commands* which are to be executed. On a monochrome display, a solid line appears after this line. On a color display, the command line will appear in a different color than the rest of the screen.

The rest of the screen is for text. If your text is longer than or wider than the screen, then this space represents a window into your text. As you progress, you will quickly learn how to position your window to view and/or modify any part of your file.

If you wish to exit from the editor, you can quit by typing a **q** followed by a carriage return on the command line. If you try this later (after some text has been entered), you may be greeted with a message indicating that you have changed some text but have not saved it away. You can force an exit without saving your text by typing a carriage return to clear the error and a **qq** followed by a carriage return. We mention quitting here to rescue those people who jump in and then get called away before they can read the rest of the manual.

Now that we have our bearings, let’s look a little closer at these three areas.

The Status Line

This line is composed of three parts. On the left, **Last=0** indicates that the last line of your text is line 0. You therefore have an empty buffer (zero lines). Next to this, the numbers in brackets **(0,1)** indicate the current position of the cursor in the file. It is of the form **(***row***,***column***)**. The row indicates your current line and for all but an empty buffer it will range in value between 1 and Last. On attached terminals you may find that the column number does not change.

Next on the line are your current editing options. A **+** (plus) after an option indicates that the option is *on* while a **-** (minus) indicates that it is *off*. Some options which are meant to catch your attention when on will display a *flashing* **+**. These options may be turned on and off at the command line. Some of the options are also tied in with function keys. They are toggled on/off by the indicated key.

Briefly the options have the following meaning:

a - Anchor (Alt – a) This option will be better understood after reading the section on pattern matching. For now it is enough to know that the editor has the ability to search for text strings and leave your cursor at the string matched. Should you specify a search for the string "mouse" with this option *on* (**a+**), your cursor will be anchored to the *start* of the pattern matched (in this case "m"). With (**a-**), the cursor would be positioned at the character *after* the matched string "mouse" (in this case the character after the "e").

b - Blank (Alt – b) When viewing a piece of text on the screen, lines appear to be padded with blanks to the end of the screen. Since the editor reads files with variable sized lines containing 0 (a line containing only a carriage return) to 256 characters, you may wonder whether the blanks at the end of the line are real or not. With **b+**, the editor will allow you to differentiate between real blanks by displaying nulls (non-existent characters) as small centered dots.

c - Command (keypad +)

This option indicates whether your active cursor is in the command area (c+) or text area (**c-**).

d - Dual (Alt – d) This option, like option anchor (**a**), concerns pattern matching. If *off* (**d-**) then a search for "mouse" would match the string "MOUSE", "MoUse", "MOUsE" and so on in your text. If *on* (**d+**), the pattern matcher differentiates between upper and lower case.

f - Fill (Alt – f) This option when *on* (**f+**) will cause automatic filling of input lines at your defined right margin. The default right margin is column 60. Should you attempt to enter a character in this

column then any preceding characters of a word will be moved to the next line. Using this option you may enter text without ever having to type a carriage return to end each line. This is extremely useful when entering documentation and letters.

i - Insert (Ins) — When *on* (**i+**), all characters typed will be inserted before the character at the current cursor position.

j - Justify (Alt – j) — This option is used in conjunction with option fill (**f+**). When both option fill and option justify (**j+**) are *on* then when each line is filled it will also be justified.

l - Limit (Alt – l) — This option will flash (**l+**) when a tagged operation is being limited between a left and right limit.

m - Meta (Alt – m) — When *on* (**m+**), meta characters are enabled during pattern matching. A meta character is a character which has special significance to the pattern matcher. For instance the character dot (**.**) is a meta character that will match *ANY* character, not just a dot. This option will be explained in greater detail later.

n - Newline (F1 or F2 keys)

Whenever a carriage return is entered in the text area while this option is on, a new line will open up after your current line to allow you to type in new text. This can be thought of as line insert mode and is similar to character insert mode (**i+**).

s - Save (Alt – s) — When *on* (**s+**), your buffer will be automatically saved in the file **autosave** after every 20 lines of input. While writing to the disk you may still continue to type up to 256 characters per line, however, your keys will not be echoed (shown on the screen) until the write is complete.

t - Tabs (Alt – t) — This option is similar to option **Blank**. **Tab** characters in your text are expanded into enough spaces to reach the next tab stop. What may appear as 4 spaces may only be one “real” character. Turning this option *on* (**t+**) will display the actual tab character as a right triangle. When option **Blank** is also *on* (**b+**), the spaces which pad the tab to the next tab stop will be displayed as centered dots since they do not exist within the buffer. Tabs are characters which are heavily used within C programs for indentation.

w - Wrap (Alt – w) — This option allows pattern searches to wrap around from bottom to top or top to bottom when *on* (**w+**). If *off*, then pattern matching will stop when it reaches the last or first line of the buffer.

The Command Line

As previously stated, this line is used for entering text to be interpreted by the editor as commands. If you have just entered the editor you should have a cursor on the left margin of your command line. There will also be an inactive cursor (rectangular block) in the text area. Whenever you type a character (excluding cursor and most function keys) it will appear where the active cursor is, and the cursor will move to the right.

Now try to enter a few commands to change your options. Most options are enabled by typing

o{*option_character*}+

and disabled by typing

o{*option_character*}-

or, they can be toggled by typing

o{*option_character*}~

First let's turn on option blank by typing the string **ob+** followed by a carriage return. You can turn it off by typing **ob-** followed by a carriage return. Finally, you can toggle it by typing **ob~** followed by a carriage return. A toggle causes the option to change state. If it was previously *off* it will be turned *on* and if it was previously *on* it will be turned *off*.

Should you make a typing error you may delete characters by pressing the Backspace key or may delete the entire line by pressing the Ctrl and X key simultaneously. Both of these keys work on the line containing the active cursor and are similar to the line editing keys of the command interpreter for the QNX2 operating system which this editor originated in. (The line editing in the QNX 4 shell is considerably different.)

If you type in an unknown command or do something which causes an error, an error message will be displayed on the command line and held until you type a carriage return to clear it. Try typing in "abc" followed by a carriage return to generate an error, then clear it by typing another carriage return.

You may execute any system command from the editor command line by preceding it with an exclamation mark (**!**).
e.g. to invoke the **ls** utility to display the contents of the current working directory:

```
!ls
```

or to invoke another **qed** editor to edit another file:

```
!qed  another_file
```

The second example will invoke another copy of the editor. When you leave you will return to this editing session.

Text Area

It is this area where your text is displayed and may be directly edited. As you have seen by setting **ob+**, the text area is currently empty. We are now going to enter some text and follow through some examples to illustrate the function of the various cursor and function keys. Option blank should be *on* for these exercises (**b+**).

Appending New Text (F1)

To enter the text area with the intention of appending new text, you should press the F1 key. The cursor in the text area will become active (flashing) and the command line will be cross-hatched. You will also see option newline enabled (**n+**) and your last and current line on the line will now be (1,1) not (0,1). The Backspace key and Ctrl – X keys behave in the same manner as they did on the control line.

Type in the following text, typing carriage return at the end of each line. We apologize to J. R. Tolkien for the misquote.

```
Three Rings for the Eleven-kings under the sky,
Nine for Mortal Men doomed to die,
One for the Dark Lord on his light throne
In the Land of Mordor where the shadows lie.
In the Land of Mordor where the shadows lie.
```

After you finish typing you will want to turn off the option newline (**n-**). This can be done by simply pressing the F1 key again. F1 is a toggle key. Note that option newline is disabled (**n-**) and the cursor is positioned at the current line or the last line typed. Please refer to the section entitled “Saving Your Text” if you wish to leave the editor and/or save your text.

Appending or Inserting Lines (F1/F2)

The F1 key appends *after* the current line and the F2 key inserts *before* the current line. Now that you have some text, let’s experiment with the cursor keys. Using the

four arrow keys at the right (←, →, ↑, ↓), you can move the cursor anywhere on your text (note: If the arrow keys generate numbers, type the Scroll Lock key). Your screen will scroll if necessary. You can move off the right of the screen until your status display indicates that you are on column 512. Note that you can't move above the first line, below the last line or to left of the first character.

Using the arrow keys, position yourself on line 2 directly under the "N" of "Nine". Press the F2 key and note that option newline is now on (**n+**). A new line has opened up between line 1 and line 2 and the cursor points at the beginning of this new line. Press F2 again and notice how the line will disappear if you decide that this is not the place where you want to insert a new line. Press F2 again and type in the following:

```
Seven for the Dwarf-lords in their halls of stone,
```

Now press F2 again to turn off option newline (**n-**) and use the cursors to position yourself on line 5 under the "I" of "In". Type F1 (**n+**) to append lines after line 5 and type in the following 2 lines:

```
One Ring to rule them all, One Ring to find them,
One Disk to bring them all and in the darkness bind them
```

Now type F1 to turn off option newline (**n-**). The F1 and F2 keys behave identically when leaving newline mode.

Using the Del and Backspace keys

Position yourself about 10 spaces to the right of the word "throne" and type a letter. The editor will immediately lengthen your line with blanks so it can place your character where requested. You can delete these blanks to restore the text by pressing the backspace key or positioning yourself just past the "e" of "throne" and holding down the Del key. The Del key deletes the character at the cursor causing the line to Shift over.

In general, you will find the Backspace key useful for correcting new text being entered, while the Del key is useful for editing existing text. As an added advantage, the Del key saves each character it deletes. Position yourself at the start of a line and hold down the Del key. To restore the last deleted characters press the Ctrl and the Ins keys simultaneously. By holding this key down you can restore the last 256 characters deleted in this editor session. The characters need not be restored where they were deleted. They will be restored at the active cursor wherever it is, even at the command line. This can be a useful method of moving strings of characters.

Change the word "Eleven" to "Elven" in line 1 by moving the cursor to the second "e" and pressing the Del key. Now move to line 7 and position the cursor under the "D" in the word "Disk" and change this word to "Ring" by simply typing over it. Note that by default, you are always in *replace* mode.

Inserting text using Ins

Now that we can enter new text, replace existing characters, delete characters and re-insert deleted characters, it would be nice to complete our capabilities by inserting new characters.

Move to the word “light” in line 4 and position the cursor under the “l”. Change this word to “dark” by pressing the Del key 5 times, pressing Ins (**i+**), typing the word “dark” and then pressing the Ins key again to disable insert mode (**i-**).

Other Cursor keys which Simplify Editing

There are several more cursor keys which will simplify movement through your text as follows:

1. Pressing Shift and Tab (back-tab), will move the cursor to the start of the current line.
2. Pressing the Ctrl and Tab, will move the cursor to the end of the current line. Always press the Shift or Ctrl key first before Tab. The Tab key by itself will place a tab character in your text. Tab characters inserted accidentally in your text can be deleted with the Del key.
3. A Ctrl and either the left or right arrow (←, →) cursor movement keys will step over words quickly. They will always stop at the end of a line containing text. The keys are symmetrical. If you overshoot, simply go in the opposite direction.
4. A Ctrl and either the up or down arrow (↑, ↓) cursor movement keys will move your cursor up or down four lines.

Saving your Text

At any time you may leave the text area and go to the command area by typing the large + key on the right hand side of the keyboard. Note that the right and left cursor keys, the Del key, and the Ins key will operate on the command line. Other cursor keys refer only to the text area. Write your file away by typing:

w *filename*

where *filename* is any valid pathname, followed by a carriage return.

After your file has been saved, the editor positions you back in the text area where you left off. You can now continue with the rest of the exercises or may leave the editor by typing the large + key again followed by a **q**. If you modify the file, the **q** command will not let you quit without saving your changes. If you do not wish to save your changes type **qq** to force the editor to exit.

Exercise

To test the other cursor keys, a file exceeding the text area must be used. To create a large file named **bigfile** on which you can do some experimentation with the editor, try the following:

```
find /bin /usr/bin | xargs -n10 echo >bigfile
```

Position the active cursor at the command line using the large + key and enter the command:

```
e bigfile
```

The **e** command deletes your current buffer and reads the file into the editor creating a new buffer. Note that the editor checks to make sure any current file you are working on has been saved before proceeding. If you get a warning, type carriage return to clear the error and use the **ee** command to force the editor to load the new file.

```
ee bigfile
```

Your buffer now contains too many lines to display on your screen and some lines which are too wide to display in their entirety. You have a window of *n* lines by *m* characters into your text. Try using your arrow keys to move about the current screen. If you attempt to leave the screen it will automatically scroll. If you want to move faster remember the Ctrl arrow keys step over words and the Shift – Tab and Ctrl – Tab move to the beginning and end of a line.

Two common reference points in a file are its beginning and end. Press the Home key and you will find yourself back at the first line. Press the End key and you will find yourself at the last line of your buffer. A Ctrl and the Home or End key will move you to the first or last line of your current screen respectively. This means that you have a *local* (to the current screenfull) home and end as well as a *global* home and end.

To step through the file a page at a time you can use the PgUp and PgDn keys. These keys lock up at the beginning or end of your file buffer. The cursor will be left at your defined center line which defaults to line 3. If you prefer it to be the first line or another line you can change it by entering command mode (the big + key) and typing the *view center* command:

vc{*number***}**

where *number* is between 1 and (*screen length* - 2). For example, to set the defined center line to line 1 enter:

```
vc1
```

Occasionally you would like to scroll the screen up or down one line without moving the cursor from its current screen line position. This can be accomplished via the Ctrl – PgUp and Ctrl – PgDn keys.

When editing programs, one of the greatest virtues of a screen editor is its ability to give you context via its full screen display. The center “5” key on the numeric keypad is designed to aid you in this respect. If you move the cursor to any line on the screen then press the 5 key (termed the *center* key), your screen will be redisplayed with that line positioned at your defined center line. Try positioning the cursor on one of the lines at the bottom of the screen and press the 5 key.

We have now exhausted the supplied cursor movement keys and it is time to examine the 10 function keys in more detail. QNX Software Systems has found that the odd function keys are in general easier to type and where possible has placed the more commonly used editing keys on them.

More on the F1 and F2 keys

The F1 and F2 keys as we have seen toggle the new line mode. While in newline mode, you may at any time go to the command area and execute any editor command or toggle any of the options. This includes setting option newline *off* instead of using the F1 or F2 keys as toggle switches. This method of entering new lines is different from most editors and much more flexible. You are not locked into an append-only mode.

For example, if you are entering text with option blank off and want to know where all your “real” blanks are, you can zip up to the command area, turn *on* option blank (**b+**) and zip back without leaving the newline mode. This facility allows you to write your file occasionally without exiting from newline mode. The editor always places you where you left off after your file has been written away.

Also, suppose you have 5 lines of text in your buffer and are half way through adding line 6 when you notice a typing error on line 2. You are free to move immediately to line 2 using the cursor movement keys, correct the error, and return again to where you left off on line 6.

In the section on “Appending or Inserting Lines”, an example had you insert a line using the F2 key and turn the newline mode off. You then moved the cursor keys to another position and appended 2 lines using the F1 key. As you will see from the following examples it was not necessary to turn the newline mode off between the two steps. Newline mode has no effect until you type a carriage return.

Try double spacing the first few lines of the **`bigfile`** file. Move to the first line of the file by pressing the Home key. Press F1 and type a carriage return and then move to the start of the second line using the cursor movement keys. Type another carriage return and continue appending a few blank lines in this manner.

You could also try the following:

1. Appending and inserting text lines in the file.
2. Correcting and inserting characters in the text using the Del and Ins key.
3. Combining steps 1 and 2 without leaving newline mode.

When leaving the newline mode using either the F1 or F2 keys, the editor will delete the last blank line. Typically when appending text users will enter their last line followed by a carriage return. This will open up an unwanted hole which the closing F1 (or F2) conveniently removes. However, for your protection this automatic delete is conditional upon the line being empty.

Deleting Lines (F3 key)

Your file now has a few blank lines which you may wish to delete. F3 is the line delete key and will delete the current line. Move the cursor to the start of a blank line and press F3. Continue deleting the blank lines in this manner. Now move to a line containing text and press F3. If you wish very hard and then press Ctrl – F2 your line will reappear.

The editor stores deleted lines in a delete buffer. The Ctrl – F1 will append the last deleted single line *after* the current line and the Ctrl – F2 will insert the line *before* the current line. In this case, we inserted the deleted line because the current cursor line moved down one line when the F3 key was used.

If you delete 5 lines you can restore them one by one using Ctrl – F1 or F2. They can be deleted in one text area and "undeleted" into another area. In most cases, this can be used as a method of moving lines of text; however, a simpler method of moving blocks of lines using the F7 and F8 keys will be explained later.

Filling Lines (F4 key)

When option fill is *on* (**f+**), your lines will be broken on word boundaries based upon your current right margin. If at some time in the future you would like to change your right margin and refill your text you can accomplish this using the F4 function key. It will take the line your cursor is on and all following lines up to:

- a blank line which is assumed to be a paragraph separator, or
- the end of the file,

whichever occurs first.

A line may be marked as a paragraph start by typing an Alt – p. A paragraph symbol will appear at the *end* of the line. The Alt – p acts as a toggle allowing you to mark/unmark a line. Filling will stop at each paragraph start, that line will be indented and filling will then continue until one of the above two conditions is reached. The paragraph start is provided as a means to stop filling between two consecutive lines.

If a group of lines have been tagged as described under the F7 key, then only those tagged lines will be filled. Filling will step over each paragraph stop as described above.

Lines are filled between your left and right margins. If option justify (toggle with Alt – j) is *on* (**j+**), then the text will also be right justified.

Centering Lines (CtrlF4)

This key will center the current line between the left and right margins. Multiple lines may be centered by tagging them as described under the F7 key.

Splitting and Joining Lines (F5/F6 keys)

Sooner or later you are going to want to split a long line or join two short lines together. The F5 key will split a line at the current cursor position into two lines. The F6 key will join the cursor line and the following line together. In practice, you normally split a line on a space and no longer want the space after the split. The F5 key checks if the character under the cursor is a space and if so, deletes it before the split. On a join, F6 checks to see if the cursor line ends in a space and, if not, appends one before joining.

If you wish to keep a space on a split or suppress the append of a space on a join, then use a Ctrl – F5 or a Ctrl – F6. These keys make no assumptions; they simply split and join. Note that the operations of split and join are symmetrical. Try splitting and joining several lines to get used to this function.

Tagging Blocks of Text (F7/F8 keys)

It is often convenient to tag a group of lines, then select an operation to perform on the lines selected. For example, earlier we mentioned a method of deleting a group of lines. Although there are several ways of accomplishing this, the most convenient way is by a tagged delete. The lines to be deleted are tagged and then deleted as a group.

There are two types of tagging.

- Line tagging
- Block (line and column) tagging.

Line Tagging

This is the simpler form and allows you to tag lines in their entirety. The selected operation applies to the whole line. For example, to delete a block of lines you would locate the first line in the series to be deleted and press the F7 key. The line is displayed in reverse-video to indicate that it has been tagged. F7 is another toggle key which sets/removes a tag, hence the line could be “untagged” by pressing F7 again. This is a more complex toggle which is also tied into block tagging, and you should pause about one second before depressing the key again to untag a line.

Now move the cursor and tag the last line to be deleted using F7. You will notice that all of the lines to be deleted in between the tagged lines now show up as inverse-video. When you set two tags, all lines between the two tagged lines are treated as a tagged block. To perform the actual delete, press the F8 key (the tag operation key). This key will prompt you for a command and typing a “d” will delete all the lines as a block. The deleted lines can be restored as a block using the Ctrl – F1 or Ctrl – F2 (probably a Ctrl – F2 since you would want to insert them before the current line).

You will find that if you delete lines one at a time the editor will restore them one at a time and if you delete them as a block they will be restored as a block. This is explained in detail in the Reference section.

The tagging of lines may also be used as a method for moving and copying blocks of lines. The F8 key has five functions:

d (delete, already explained)

m (move)

k (kopy)

s (save)

p (print lines on $lpt)

To try the move option, find a small block of text and tag the first and last lines using the F7 key. Move the cursor to another place in the text, press F8 and respond to the prompt with an "m". The tagged block of text will be appended after the current cursor line.

The kopy option is analogous to the move option. Tag a block of lines using the F7 keys. Move to the desired location, press F8, enter a "k" when prompted and the lines will be copied after the current cursor line. The original tagged block of text remains unchanged.

The save option will save the tagged lines into the file "**/tmp/***group***.***user*", where *group* is your current numerical group ID and *user* is your numerical user ID. The saved text may be restored later using the Shift – F8 key. Selecting a line paste (**l**) will insert the text before the line your cursor is on. You may save and restore text between different edit sessions and/or consoles.

The F4 (fill) and Ctrl – F4 (center) keys will operate on tagged lines if any have been set. If not they default to the range indicated on the section describing them.

Some things to remember when using tags:

- You are allowed a maximum of two tags at any one time. If you set a third tag, then it replaces an existing tag.
- Tag operations apply to all lines between two tagged lines. If only one tag is set, a tag operation will apply to that line only.
- The order in which the tags are set does not matter.
- Tags may be removed from your text by:
  - performing a tag operation function,
  - using the F7 key as a toggle switch, or
  - typing a Ctrl – F7 which removes/resets all tags.

Block Tagging

Line tagging is sufficient for most editing tasks. It falls short in those cases where you wish to operate on a block of text within a line. This occurs most frequently in the preparation of multi-column text. For example, if you wished to move the block of "C"s

```
AAAAAAA    BBBBBBB    CCCCCCC
AAAAAAA    BBBBBBB    CCCCCCC
AAAAAAA    BBBBBBB    CCCCCCC
AAAAAAA    BBBBBBB    CCCCCCC

DDDDDDD
DDDDDDD
DDDDDDD
DDDDDDD
```

under the block of "B"s you would need to tag only the "C"s, not the "A"s and "B"s. Enter the text above and position yourself on the first character of the first line of "C"s. Depress the F7 (tag key) twice in rapid succession. The first depression will tag the line and the second depression will turn on the block tag feature and set a left limit at your cursor. The characters from the first C to the end of your line will be displayed in inverse video.

Now move your cursor to the C in the lower right corner and depress the F7 key again. You need only depress it once. This will tag the last line of "C"s and set a right limit. The block of "C"s should now be displayed in inverse video. If you wish, you may adjust the line range being tagged or the limits of the tag by moving your cursor and typing the F7 key. Unlike line tagging, typing F7 again will not remove a block tag.

Now move your cursor under the first B on the line containing the first row of "D"s and depress the F8 key. You will be queried with a more extensive list of operations then a simple line delete. They are

d - delete The tagged block will be deleted. Any text to the right will slide over to the left to fill the gap.

e - erase The tagged block will be replaced by blanks. In the special case where there is no text to the right of the block then the text is simply deleted. This option will maintain column integrity.

k - kopy The tagged block will be copied to the current cursor location. The original text is unchanged unless the destination overlaps the tagged block.

m - move and erase

This is a combined **kopy** followed by an **erase**.

M - move and delete

This is a combined **kopy** followed by a **delete**.

s - save The save option will save the tagged text into the file "**/tmp/***group***.***user*", where *group* and *user* will be numbers. The saved text may be restored later using the Shift – F8 key. Selecting a column paste will insert the text at your cursor. You may save and restore text between different edit sessions.

p - print Print tagged block on the printer ($lpt).

Insert Mode and Block Move and Copy

When performing a tagged block MOVE or COPY with option insert *off* (**i-**), the moved text will simply overlay any existing text. If option insert is *on* (**i+**) then the text will be inserted before the cursor. This can add considerable convenience. For example you may move the block of "C"s in front of the block of "D"s by tagging it, enabling option insert, and performing a move to the first character of the first line of "D"s. You should tag extra spaces to the right so the columns line up. After the move, delete the extra block of "C"s. It is recommended that you always check the state of option insert when performing a tagged block move.

Re-setting the Last Tagged Lines or Block

The Ctrl – F7 is a toggle which can be used to retag lines or a block. Depressing it with no tags set will restore the last tags set. Depressing it with tags set will clear all tags.

Tags are set on absolute line and column numbers and inserting or deleting lines and characters may cause your tags to move with respect to your text.

Re-executing Commands (F9/F10 keys)

These two keys are used for re-executing command line commands. The F9 key will re-execute the last command. F10 will redisplay the last command allowing you to edit it, if needed, before typing a carriage return to execute it. These functions will be more useful as you learn more editor commands in the next sections.

Tabs

The editor has tab stops set every four columns with the first tab set on column five. Typing the tab key will enter a tab character at your active cursor. When displayed on your screen the tab character will be expanded into the necessary number of spaces to move to the next tab stop. Try inserting a few tabs into your text. You can display the tab character as a right triangle by turning on option tabs (**ot+**). By also turning on option blank (**ob+**), padding spaces will be displayed as small centered dots since they do not exist within your text.

Tabs are not treated with any special significance internally. They only affect your display and cursor movement on the display. You can not position your cursor on the padding spaces following a tab, only on the tab character itself or the first real character following it.

The use of tabs rather than spaces for indentation in structured languages can save considerable typing and file space storage on your diskette.

You can change your tab settings by using the View Tab command. On the command line type:

Command	gives
vt2	Tabs every 2 columns
vt4	Tabs every 4 columns
vt8	Tabs every 8 columns

Line Drawing Characters

You can redefine the Ctrl and Alt cursor keys into line drawing characters by executing a macro file as follows from the **qed** command line:

```
x /usr/lib/box.macros
```

Experiment by typing each Ctrl – *cursor-key* and Alt – *cursor-key* within the editor. The Home, PgUp, PgDn, End and 5 key should also be used. Some of the keys will move you in the most natural direction for drawing a box, however, the Ctrl – ↓ will not move beyond the last line of your file buffer. You can press carriage return to open up room.

Margins

The editor maintains a left margin which determines the point a carriage return will return to and a right margin which determines the point at which filling will occur.

Moving Your Margins (Shift F1 to F6)

A Shift – F1 will set your left margin at your current cursor position and a Shift – F2 will set your right margin at your current cursor position. Shift – F3 and Shift – F4 will march your left margin left or right while a Shift – F5 and Shift – F6 will march your right margin left or right.

Auto Fill and Your Right Margin

Go to the command area (using the large + key) and enable option fill with the command **of+** or toggle it on with the Alt – f key combination. In the text area, append the following ten lines (taken from “The Princess Bride” - William Goldman) as one long line. Do *not* type a carriage return.

```
The year Buttercup was born, the most beautiful
woman in the world was a French scullery maid named
Annette. Annette worked in Paris for the Duke and
Duchess de Guiche, and it did not escape the Duke’s
notice that someone extraordinary was polishing the
pewter. The Duke’s notice did not escape the notice of
the Duchess either, who was not very beautiful and not
very rich, but plenty smart. The Duchess set about
```

```
studying Annette and shortly found her adversary's
tragic flaw. Chocolate.
```

You will find that the editor will automatically take any partially typed word and move it to the next line when the cursor column exceeds 60, which is your default right margin. This feature allows you to type continuously without having to closely watch the screen. You can change your right margin using the **View** command which will be described shortly.

Auto Justify

Go to the command area and enable option justify with the command **oj+** or toggle it on with the Alt – j key combination. In the text area, append the following lines.

```
Prince Humperdinck was shaped like a barrel. His chest
was a great barrel chest, his thighs mighty barrel
thighs. He was not tall but he weighed close to 250
pounds, brick hard. He walked like a crab, side to side,
and probably if he had wanted to be a ballet dancer, he
would have been doomed to a miserable life of endless
frustration.
```

You will find that the editor automatically justifies each line with your right margin when it fills. This option is only active when option fill is enabled.

Indenting and Your Left Margin

A Ctrl – b will begin an indent of four spaces (increase your left margin by four) and a Ctrl – e will end an indent (decrease your left margin by four). Append the following 6 lines, holding down the Ctrl and typing a b or e for Ctrl – b and Ctrl – e.

```
The year Buttercup turned ten,
Ctrl - bthe most beautiful woman lived in Bengal,
Ctrl - bthe daughter of a successful tea merchant.
This girl's name was Aluthra,
Ctrl - eand her skin was of a dusky perfection
Ctrl - eunseen in India for eighty years.
```

You should note that your left margin only determines the point you will return to on a carriage return. You may use your cursor keys to move to the left of an indent.

Line Flags

Overstrike Flag (Alt – o)

This will flag the end of the current line with a left arrow character. When this line is written, the record separator will be replaced by a carriage return. As a result, when this line is printed, no linefeed will be issued and the following line will *overstrike* this line. This flag is automatically set when reading source lines which terminate in carriage returns rather than record separators. This allows reading and editing the output of **DOC** files which have underlining and/or boldfacing. (**DOC** is a document

markup language which was used for the original QNX 2 manuals and which was also offered as a product by Quantum Software Systems Ltd.)

Continuation Flag (Alt – c)

This will flag the end of the current line with a bidirectional arrow character. When this line is written, the record separator will be suppressed. As a result, the next line will be *continued* (joined) with this one. When the editor reads a line of greater than 512 characters it will automatically split the input line and set this flag on the split line. This allows for the editing of files containing very long lines.

Paragraph Flag (Alt – p)

This will flag the end of the current line with a paragraph symbol. This is used by the fill key (F4).

Some Simple Editor Commands.

You should now be able to create new files, edit existing files (**e** *filename*) and write them away (**w** *filename*). Up until now you have had little reason to leave text mode to go to command mode to execute editor commands. The next sections will introduce you to some useful editor commands.

Learn Mode

If you have a sequence of keys which you enter often, you may wish to learn them once and assign them to a single key. Let's assume that you wish to learn the string "Copyright (c) 1998". The next time you are about to enter it, type a Ctrl – Minus sign on the keypad and enter a Ctrl – a when prompted for the key to learn. Type in your text then signal the end of learn mode by typing a Ctrl – Break. From this point on each time you enter a Ctrl – a it will be replaced by the learned input sequence.

It is possible to learn very long and complex sequences consisting of text, cursor movements and commands.

Absolute Line Positioning

The Home, End, PgUp and PgDn keys allow you a coarse means of moving through a buffer, however, if you want to be in the middle of a thousand line buffer they are very awkward to use. If you go to the command line and type in the number of the line you want to go to (followed by a carriage return to execute) the editor takes this as a command to move your cursor to that line. This gives you the ability to move through the file in absolute terms. For example, if a compiler issues an error for line 458 of a source file then upon reading that source file with the editor you can go right to that line.

Simple Pattern Matching

When editing a file, you often want to be able to say "Find me an occurrence of this string" so you can work on it without knowing precisely where it is. This is especially

true when working from a paper listing in which you know the text string you want to edit, but probably not its line number. In the editor you can find a string simply by enclosing it in slashes on the command line. The command **/son/** will cause a search to be made for the string "son". If option dual is off (**od-**) then the matching will be case insensitive and **/son/** will match "SON", "SoN", "sON" and so on. It will also match a line containing "personal" since it contains an instance of "son".

Searching begins at the character *after* your cursor, resulting in the editor finding the *next* occurrence of your pattern. If the editor searches down to the end of the buffer without finding your pattern and option wrap is *on* (**w+**) it will continue the search at the first line of the buffer and continue until it reaches and tries your current line from behind. If your pattern is not found, an error will be generated.

Assuming you match a line containing your pattern, then that line will become your current line and your cursor will either point to the first character of the string matched (**a+**) or the character after this string (**a-**). The default is option anchor enabled (**a+**); however, if you are adding commas to the end of words you may prefer to switch to **a-**.

Enclosing a pattern in question marks instead of slashes will cause a search to be made backwards through your buffer. Therefore **?son?** will search for the first occurrence of the string "son" starting at the character before the cursor. If the editor searches backward to the beginning of the buffer without finding the pattern and option wrap is *on* (**w+**) it will continue searching backwards from the end of the buffer. Again if your pattern is not found an error will be generated.

The editor is careful always to remember the last pattern you specified. Typing **//** will cause a search for the last pattern specified. This can save typing when looking for multiple occurrences of a long pattern. An even faster method would be to use the F9 key to re-execute the last command.

Up until now we have used the words *string* and *pattern* interchangeably. Although the editor can search for simple strings of characters like **/The quick brown fox/** they are a subset of a more powerful pattern matching facility. This facility is enabled by the option metacharacters (**om+**) when you define a pattern. When enabled the characters **.**, **@**, **$**, **^**,*****, /, **?**, and **[** have a special meaning. For instance, a dot (**.**) is a pattern which matches *any* character. The pattern **/a.c/** would match an occurrence of a string containing an "a" followed by *any* character followed by a "c". The meanings of these characters are explained in detail in the reference manual chapter. Unless you intend to read this you should turn off option metacharacters (**om-**) or prefix all special characters with a backslash character in your pattern. When prefixed by a backslash the special characters mentioned above lose their special meaning. A **/cat\./** will only match the string "cat." regardless of the state of option **m**. If you want to match a backslash you must type two of them. A **/a\\b/** will match the string "a\b". For non-alphanumerics the rule is simply this. If a character is preceded by a \ then remove the backslash and take that character literally. This allows you to specify a slash in your pattern. A **/total\/number/** will match the string "total/number". The slash is protected and not taken as the pattern delimiter.

For alphanumerics the rule is slightly more involved. If the two characters following the backslash are hexadecimal (**0** to **9** or **A** to **F**) then the whole sequence is taken as one character which has the hexadecimal value indicated. A **/\07/** is a pattern consisting of the single character whose hexadecimal value is 07. A **/\z/** is just a "z" while a **/\bc/** is the character whose hexadecimal value is 0xBC. If this seems complex or confusing you probably don't need this ability. Just remember to type two back slashes to get one and that you can match a slash (or a **?** if you scan backwards) by prefixing it with a backslash.

File I/O Commands

As we have already seen, we can read a file into the editors buffer with the **e** command

e *filename*
ee *filename*

and write out our buffer to a file with the **w** command.

w *filename*

Whenever you read a file with the edit command (**e**) three things actually occur:

1 Your current buffer is purged. However, your delete buffer is kept, allowing you to delete from one file and undelete into another.

2 The filename you specified is memorized.

3 The contents of the file are read into your empty buffer.

In each example we have specified the filename that the command should operate upon. This is not always necessary. Should you omit the filename it will default to the filename of the last file you specified with your **e** command. Therefore,

```
e report
```

followed by a

```
w
```

will have the write command write to the file "report". Likewise, an

```
e
```

will simply re-read the last file edited. This is often used when you bungle something and want a fresh copy. Note that in this case you will probably have to issue an

```
ee
```

command to indicate that your unsaved buffer should be overwritten. For your protection a simple **e** will not destroy an unsaved buffer.

If you want to check the name of your current file, you can issue the file command:

```
f
```

which will display it in the command area. You can change it (or define it) by following the command with a filename:

```
f  file1
```

This will define your current filename as **file1**.

It is often useful to read another file into a non-empty buffer after some specified line. This can be accomplished with the **r** command; e.g. to read the file **file2** into the buffer after the current line:

```
r  file2
```

The command may be prefixed by a number (or pattern search) to explicitly indicate a particular line; e.g. to read **file2** after the 10th line in the text buffer:

```
10r  filename
```

Similarly, to read **file2** after the line containing the next occurrence of "end" in the text buffer:

```
/end/r filename
```

The read ("**r**") command does not affect the current filename. If you do not specify a filename after "**r**" the editor will read another copy of the current file into your text buffer.

It is also possible to prefix the write ("**w**") command with a line number or range of lines to write.

Some examples follow.

Write all lines:

w *filename*

Write line 33 only:

33w *filename*

Write lines 1,10:

1,10w *filename*

Write all tagged lines:

#w *filename*

Finally, you can append to a file by specifying the write append command which is of the same form as the write command:

wa *filename*

This is useful when you wish to build a new file based upon lines or blocks of lines from your current file (or many files), for example:

- Initialize file with lines 1 through 4:
  1,4w *file*

- Append lines 24 through 30:
  24,30wa *file*

- Append a group of tagged lines:
  #wa *file*

- Read a new file:
  e *newfile*

- Append first 10 lines:
  1,10wa *file*

The View Command

Users with a color display can change the color of the three display areas using the view attribute command.

va{*area***}{***foregroundcolor***} {***backgroundcolor***}**

where:

{***area***}=	**Indicates:**
1	the status line
2	the command line
3	the text area

and:

{*foregroundcolor***}** or **{***backgroundcolor***}**

are color numbers between 1 and 15 inclusive

Executing System Commands

Any QNX command may be executed from within the editor by preceding it with an exclamation mark (**!**).

For example, to list the contents of the current working directory:

```
!ls
```

or to edit the file **another_file**:

```
!qed  another_file
```

After executing the QNX command, the editor will pause, waiting for the user to type carriage-return before redisplaying the screen. If you type two exclamation marks (**!!**) the screen clear and pause will be suppressed allowing you to invoke commands via macros in a hidden manner.

Note that Ctrl – Z is recognized within the editor and will bring up an interactive shell allowing you to execute multiple commands. To return to the editor type a Ctrl – D or issue the shell **exit** command to terminate the new shell.

Epilogue

At this point you should be able to perform most editing tasks. However, this introduction has hit on only a few of the editor commands and interested users are strongly urged to read the reference section, especially if they are interested in defining their own function key operations.

Chapter 2

Using qed on non-QNX terminal types

Setting Your Terminal Type

When **qed** is invoked, it uses terminal capabilities described in the terminfo entry for the terminal type set in the **TERM** environment variable. The **qed** editor was written specifically for the original QNX console terminal type (**TERM=QNX**) and the IBM PC keyboard. Depending on the console driver in use, your system may be using either ANSI based screen controls (**Dev.ansi**) or QNX screen controls (**Dev.con**). When used on an ANSI-based terminal, some key combinations may not work.

Most of the basic keycodes needed by **qed** are described in extra terminfo capabilities that are included in the terminfo files for the terminal components shipped with QNX; this includes **Dev.ansi** (terminal type **qansi**) and **xterm** (terminal type **xterm**). For other terminal types you may need to add capabilities to make **qed** function acceptably well.

The following is a table of the mappings for the terminfo capabilities used by **qed**:

qnxterm.h #def	kbd value 16/8	terminfo name	curses #def	full name
K_DOWN	ffa9 251	kcud1	KEY_DOWN	down arrow
K_UP	ffa1 241	kcuu1	KEY_UP	up arrow
K_LEFT	ffa4 244	kcub1	KEY_LEFT	left arrow
K_RIGHT	ffa6 246	kcuf1	KEY_RIGHT	right arrow
K_HOME	ffa0 240	khome	KEY_HOME	home
		kbs	KEY_BACKSPACE	backspace
		kf0	KEY_F(0)	f0
K_F1	ff81 201	kf1	KEY_F(1)	f1
K_F2	ff82 202	kf2	KEY_F(2)	f2
K_F3	ff83 203	kf3	KEY_F(3)	f3
K_F4	ff84 204	kf4	KEY_F(4)	f4
K_F5	ff85 205	kf5	KEY_F(5)	f5
K_F6	ff86 206	kf6	KEY_F(6)	f6
K_F7	ff87 207	kf7	KEY_F(7)	f7
K_F8	ff88 210	kf8	KEY_F(8)	f8
K_F9	ff89 211	kf9	KEY_F(9)	f9
K_F10	ff8a 212	kf10	KEY_F(10)	f10
K_F11	ffae 256	kf11	KEY_F(11)	f11
K_F12	ffaf 257	kf12	KEY_F(12)	f12

continued...

qnxterm.h #def	kbd value 16/8	terminfo name	curses #def	full name
K_SHF_F1	ff8b 213	kf13	KEY_F(13)	f13
K_SHF_F2	ff8c 214	kf14	KEY_F(14)	f14
K_SHF_F3	ff8d 215	kf15	KEY_F(15)	f15
K_SHF_F4	ff8e 216	kf16	KEY_F(16)	f16
K_SHF_F5	ff8f 217	kf17	KEY_F(17)	f17
K_SHF_F6	ff90 220	kf18	KEY_F(18)	f18
K_SHF_F7	ff91 221	kf19	KEY_F(19)	f19
K_SHF_F8	ff92 222	kf20	KEY_F(20)	f20
K_SHF_F9	ff93 223	kf21	KEY_F(21)	f21
K_SHF_F10	ff94 224	kf22	KEY_F(22)	f22
K_SHF_F11	ffdb 333	kf23	KEY_F(23)	f23
K_SHF_F12	ffdc 334	kf24	KEY_F(24)	f24
K_CTL_F1	ff95 225	kf25	KEY_F(25)	f25
K_CTL_F2	ff96 226	kf26	KEY_F(26)	f26
K_CTL_F3	ff97 227	kf27	KEY_F(27)	f27
K_CTL_F4	ff98 230	kf28	KEY_F(28)	f28
K_CTL_F5	ff99 231	kf29	KEY_F(29)	f29
K_CTL_F6	ff9a 232	kf30	KEY_F(30)	f30
K_CTL_F7	ff9b 233	kf31	KEY_F(31)	f31
K_CTL_F8	ff9c 234	kf32	KEY_F(32)	f32
K_CTL_F9	ff9d 235	kf33	KEY_F(33)	f33
K_CTL_F10	ff9e 236	kf34	KEY_F(34)	f34
K_CTL_F11	ffbe 276	kf35	KEY_F(35)	f35
K_CTL_F12	ffbf 277	kf36	KEY_F(36)	f36
K_ALT_F1	ffd1 321	kf37	KEY_F(37)	f37
K_ALT_F2	ffd2 322	kf38	KEY_F(38)	f38
K_ALT_F3	ffd3 323	kf39	KEY_F(39)	f39

continued...

qnxterm.h #def	kbd value 16/8	terminfo name	curses #def	full name
K_ALT_F4	ffd4 324	kf40	KEY_F(40)	f40
K_ALT_F5	ffd5 325	kf41	KEY_F(41)	f41
K_ALT_F6	ffd6 326	kf42	KEY_F(42)	f42
K_ALT_F7	ffd7 327	kf43	KEY_F(43)	f43
K_ALT_F8	ffd8 330	kf44	KEY_F(44)	f44
K_ALT_F9	ffd9 331	kf45	KEY_F(45)	f45
K_ALT_F10	ffda 332	kf46	KEY_F(46)	f46
K_ALT_F11	ffce 316	kf47	KEY_F(47)	f47
K_ALT_F12	ffcf 317	kf48	KEY_F(48)	f48
		kf49	KEY_F(49)	f49
		kf50	KEY_F(50)	f50
		kf51	KEY_F(51)	f51
		kf52	KEY_F(52)	f52
		kf53	KEY_F(53)	f53
		kf54	KEY_F(54)	f54
		kf55	KEY_F(55)	f55
		kf56	KEY_F(56)	f56
		kf57	KEY_F(57)	f57
		kf58	KEY_F(58)	f58
		kf69	KEY_F(59)	f59
		kf10	KEY_F(60)	f60
		kf61	KEY_F(61)	f61
		kf62	KEY_F(62)	f62
		kf63	KEY_F(63)	f63
K_CTL_DELETE	ffbc 274	kdl1	KEY_DL	delete line key
K_CTL_INSERT	ffbb 273	kil1	KEY_IL	insert line
K_DELETE	ffac 254	kdch1	KEY_DC	delete character key
K_INSERT	ffab 253	kich1	KEY_IC	ins char/enter ins mode key

continued...

qnxterm.h #def	kbd value 16/8	terminfo name	curses #def	full name
K_ALT_INSERT	ffcb 313	krmir	KEY_EIC	rmir or smir in insert mode
K_ALT_A	ffe1 341	kclr	KEY_CLEAR	clear screen or erase key
K_ALT_DELETE	ffcc 314	ked	KEY_EOS	clear-to-end-of-screen key
K_ALT_END	ffc8 310	kel	KEY_EOL	clear-to-end-of-line key
K_CTL_UP	ffb1 261	kind	KEY_SF	scroll-forward/down key
K_CTL_DOWN	ffb9 271	kri	KEY_SR	scroll-backward/up key
K_PGDN	ffaa 252	knp	KEY_NPAGE	next-page key
K_PGUP	ffa2 242	kpp	KEY_PPAGE	previous-page key
K_ALT_B	ffe2 342	khts	KEY_STAB	set-tab key
K_CTL_TAB	ff9f 237	kctab	KEY_CTAB	clear-tab key
K_ALT_D	ffe4 344	ktbc	KEY_CATAB	clear-all-tabs key
K_CTL_ENTER	ffd0 320	kent	KEY_ENTER	Enter/send (unreliable)
K_PRTSC	ffad 255	kprt	KEY_PRINT	print or copy
		kll	KEY_LL	home-down key
		ka1	KEY_A1	Upper left of keypad
		ka3	KEY_A3	Upper right of keypad
		kb2	KEY_B2	Center of keypad
		kc1	KEY_C1	Lower left of keypad
		kc3	KEY_C3	Lower right of keypad
K_BACKTAB	ff80 200	kcbt	KEY_BTAB	Back tab key
K_ALT_HOME	ffc0 300	kbeg	KEY_BEG	beg(inning) key
K_KPDMINUS	ffa3 243	kcan	KEY_CANCEL	cancel key
K_ALT_C	ffe3 343	kclo	KEY_CLOSE	close key
K_KPDFIVE	ffa5 245	kcmd	KEY_COMMAND	cmd (command) key
K_CTL_KPDFIVE	ffb5 265	kcpy	KEY_COPY	copy key
K_ALT_KPDFIVE	ffc5 305	kcrt	KEY_CREATE	create key

continued...

qnxterm.h #def	kbd value 16/8	terminfo name	curses #def	full name
K_END	ffa8 250	kend	KEY_END	end key
K_CTL_END	ffb8 270	kext	KEY_EXIT	exit key
K_ALT_F	ffe6 346	kfnd	KEY_FIND	find key
K_ALT_H	ffe8 350	khlp	KEY_HELP	help key
K_ALT_M	ffed 355	kmrk	KEY_MARK	mark key
K_ALT_E	ffe5 345	kmsg	KEY_MESSAGE	message key
K_ALT_I	ffe9 351	kmov	KEY_MOVE	move key
K_ALT_PGDN	ffca 312	knxt	KEY_NEXT	next object key
K_ALT_O	ffef 357	kopn	KEY_OPEN	open key
K_ALT_K	ffeb 353	kopt	KEY_OPTIONS	options key
K_ALT_PGUP	ffc2 302	kprv	KEY_PREVIOUS	previous object key
K_CTL_RUBOUT	ffde 336	krdo	KEY_REDO	redo key
K_ALT_L	ffec 354	kref	KEY_REFERENCE	ref(erence) key
K_ALT_G	ffe7 347	krfr	KEY_REFRESH	refresh key
K_ALT_R	fff2 362	krpl	KEY_REPLACE	replace key
K_ALT_J	ffea 352	krst	KEY_RESTART	restart key
K_ALT_P	fff0 360	kres	KEY_RESUME	resume key
K_ALT_Q	fff1 361	ksav	KEY_SAVE	save key
K_ALT_N	ffee 356	kBEG	KEY_SBEG	shifted beginning key
K_CTL_KPDMINUS	ffb3 263	kCAN	KEY_SCANCEL	shifted cancel key
K_CTL_KPDPLUS	ffb7 267	kCMD	KEY_SCOMMAND	shifted command key
K_ALT_S	fff3 363	kCPY	KEY_SCOPY	shifted copy key
K_ALT_T	fff4 364	kCRT	KEY_SCREATE	shifted create key
		kDC	KEY_SDC	shifted delete char key
K_ALT_V	fff6 366	kDL	KEY_SDL	shifted delete line key
K_KBDPLUS	ffa7 247	kslt	KEY_SELECT	select key
K_ALT_UP	ffc1 301	kEND	KEY_SEND	shifted end key
K_ALT_DOWN	ffc9 311	kEOL	KEY_SEOL	shifted clear line key
K_ALT_W	fff7 367	kEXT	KEY_SEXIT	shifted exit key
K_ALT_X	fff8 370	kFND	KEY_SFIND	shifted find key

continued...

qnxterm.h #def	kbd value 16/8	terminfo name	curses #def	full name
K_ALT_Y	fff9 371	kHLP	KEY_SHELP	shifted help key
K_CTL_HOME	ffb0 260	kHOM	KEY_SHOME	shifted home key
K_SHF_ENTER	ffe0 340	kIC	KEY_SIC	shifted input key
K_CTL_LEFT	ffb4 264	kLFT	KEY_SLEFT	shifted left arrow key
K_ALT_LEFT	ffc4 304	kMSG	KEY_SMESSAGE	shifted message key
K_ALT_RIGHT	ffc6 306	kMOV	KEY_SMOVE	shifted move key
K_CTL_PGDN	ffba 272	kNXT	KEY_SNEXT	shifted next key
K_ALT_Z	fffa 372	kOPT	KEY_SOPTIONS	shifted options key
K_CTL_PGUP	ffb2 262	kPRV	KEY_SPREVIOUS	shifted prev key
K_CTL_PRTSC	ffbd 275	kPRT	KEY_SPRINT	shifted print key
K_ALT_PRTSC	ffcd 315	kRDO	KEY_SREDO	shifted redo key
		kRPL	KEY_SREPLACE	shifted replace key
K_CTL_RIGHT	ffb6 266	kRIT	KEY_SRIGHT	shifted right arrow
		kRES	KEY_SRSUME	shifted resume key
K_ALT_KPDPLUS	ffc7 307	kSAV	KEY_SSAVE	shifted save key
K_ALT_KPDMINUS	ffc3 303	kSPD	KEY_SSUSPEND	shifted suspend key
		kUND	KEY_SUNDO	shifted undo key
		kspd	KEY_SUSPEND	suspend key
K_ALT_U	fff5 365	kund	KEY_UNDO	undo key
K_RUBOUT	007f 177	del		
K_ERASE	0018 030	^x		
K_TAB	0009	tab 011		

Required Terminal Capabilities

Any terminal which is to be used with **qed** must support the following capabilities:

- Direct Cursor Addressing
- Screen Clearing
- Zero width escape sequences

The last point requires further explanation. On some terminals, an escape sequence to turn on inverse video takes up a character position on the line. As a result your

standard 80 column line will be reduced to 78 columns (one character to turn on, and one character to turn off) The **qed** editor does *not* work properly on this type of terminal. Escape sequences must *not* take any physical room on the screen.

Although not required, the following capabilities are recommended

- Clear to end of line.
- Insert and delete line.
- Inverse video (and to a lesser extent underline).
- Up, down, left and right cursor keys.

The first two will speed up display updates while the third is necessary for displaying tagged blocks of text. If your terminal supports Highlighting or underline, but not inverse video, then **qed** will attempt to display tagged areas of text using these capabilities. Without cursor keys, the editor may be painful to use.

Screen Output

qed will adjust its output to conform to your terminal's screen size. Display updates will be determined by the baud rate if **qed** is being run from an attached terminal. This will be considerably slower than running **qed** on a console, xterm etc. Non-ASCII characters (0x00-0x1f and 0x7e-0xff) will be displayed as a question (**?**) mark. They are saved and manipulated as the characters they really represent. It is only the display which prints them as question marks.

Note: on attached terminals, the column position is NOT updated as you move your cursor. To force an update you must type the **<**SHOW**>** key, which on a PC keyboard, is the center key (5) on the numeric keypad when Num Lock is turned off.

Keyboard Input

It is keyboard input which really differentiates using **qed** on a QNX terminal type from **qed** on a non-QNX terminal. The console keyboard is rich in both function and cursor keys. Unfortunately, most terminals are rather limited in the special keys that they provide. To overcome this it is often necessary to enter a two or three character sequence to simulate a single console key. The terminal database defines keys according to their function. For example, the large + key on the console keyboard's numeric keypad is called the "select" key. It is used by **qed** to leave some form of input mode and return back to a command state. This key is very seldom found on a terminal, so you will have to add a kslt capability to your terminal type. Note that this should be something sent by one keystroke unless you have very fast fingers – the timeout between characters of multi-character keycodes is quite quick.

There is no provision for executing any of the Alt letter keys such as Alt – b to turn on option blank. You will have to go to the command line and type the option command directly. (**ob+** *or* **ob-**)

You may examine the escape sequences in effect for your terminal by using the **infocmp** utility.

Chapter 3

Reference Manual

In this chapter. . .

The Syntax of Editor Commands

Editor commands are given in the form:

[*line_range***]***C***[***argument***]**

where:

line_range	indicates which lines to act on
C	is a single character indicating the command to execute
argument	is command specific information

Line Range

The line range specifies which lines the command should operate on. It can consist of zero, one or two line addresses. If no range is specified then it will usually default to the current line. The current line is the line your cursor is on in the text area and will typically be updated by each command to reflect the last line it operated on. Check the section on each command for the exact behavior.

The following line range forms are allowed:

<*None***>**	No line address — this causes the command to choose a default line address. If not stated otherwise in the documentation for a command, the default for will be the current line.
line	The command will operate only on the single line number *line*
*line1***,***line2*	The command will operate on all lines between *line1* and *line2*, inclusive.
*line1***;***line2*	The command will set the current line to *line1*, then operate on all lines between *line1* and *line2*, inclusive.
*****	This form is a synonym for **1,$**, which means line 1 to the last line in the file. The command will operate on all lines.
#	The command will operate on the tagged lines in the buffer.

The elements of a line range are line addresses *line* and are composed of:

number	This is an ordinary line number referring to the *number*th line of the buffer.
$	This special character refers to the last line of the buffer.
.	This refers to the current line of the buffer.

@	This refers to the line occupying the currently defined center line.
&	This refers to the top line of your currently displayed screen.
%	This refers to the current line if you are in the text area and line zero if you are on the command line. It is often used in macros by the **Zap** (**z**) command to operate on the command line.
*/*_pattern_**/**	This is the address of the line which contains an instance of the specified pattern. The search for *pattern* will begin at the character after the cursor and will continue to the end of the buffer. If no match has been found by that time, and option wrap is *on* (**w+**), the search will wrap around to the beginning of the buffer and continue looking for *pattern* from line one. If no match is found in the entire buffer then an error will be issued. This line search sets the condition register TRUE if a match is found and FALSE if a match is not found.
?_pattern_**?**	This serves as the line address of the line which contains an instance of the specified pattern. The search for *pattern* will begin at the character before the cursor and will go backwards through the buffer wrapping around from the first to last line if necessary. If no match is found an error will be issued as above. This line search also sets the condition register to TRUE on a match and FALSE on no match.

Each line address above may be combined with other line addresses using the **+** and **-** characters to form expressions. For example:

.-5,.+5	will specify the five lines before and after the current line.
&;.+23	will specify all lines on the current screen.
?begin?,/end/	will specify all lines between the lines containing the previous **begin** and the next **end**.

If you specify a line address which is outside the buffer you will get an error and the command will *not* be executed. The special character "**|**" can be used to limit the preceding line addresses to lie within the buffer (between one and **$**). This is very useful in defining macros. The **|** operator sets the condition register FALSE if the line address falls outside the buffer and needs to be limited. For example:

```
&;.+23|
```

is a safer form of the example shown above.

Command Specification Character

Each major command consists of a single character which has been chosen to reflect its nature. For example, the character **d** was chosen for the delete command and the character **w** was chosen for the write command. This character must be in lower case.

If a command line is entered which contains a *line_range* but no command

```
44
```

then the current line is set to the last line address specified. In this case the cursor will move to line 44.

Right Arguments

Some commands require extra information to specify their operation. For example, the **Move** (**m**) command requires the specification of the destination line address:

{*line1***},{***line2***}m{***line3***}**

Other commands like the **Zap** (**z**) command represent a class of commands which are specified by sub-command characters. For example:

```
zcd
```

is a zap cursor delete command. The **cd** is a subcommand of the zap command and will not be interpreted as the major commands **c** and **d**. This form of subcommand is used by several editor major commands.

Placing Multiple Commands On A Line

The editor allows you to place more than one command on a line. Each command on the line is executed sequentially from left to right. For example:

```
ob+ot+
```

will turn on option blank followed by option tab. The command:

```
1,4d$d
```

will delete lines one through four and then delete the last line. Note that the line range only applies to the command that it immediately precedes. Should an error occur on any command then execution will be halted and any following commands will *not* be executed.

Some editor commands consume all characters until the end of the line collecting their right argument. They must therefore be the last command on any line on which they occur. For example

e *filename*

consumes all characters until the end of the line collecting the filename.

Special Characters

The Full Screen Editor treats a small number of characters as special in certain situations. These characters are described in the following subsections. The only character you cannot save in your text is an ASCII NUL (hex 00).

The Linefeed Character (hex 0A)

The line separator character in QNX4 is a linefeed (hex 0A). Source files separate lines by a single linefeed character, not a carriage return. On input, whenever you enter a carriage return (hex 0D) it is mapped into a linefeed character.

When the editor reads a file it collects characters up until a linefeed, replaces the newline with a null (hex 00) and saves the collected characters as a line in your buffer. The point to note is that the linefeed is *not* saved. It is stripped on a read and added to the end of each line when the file is written.

In the definition of complex macros containing several lines, the lines may be separated by either a carriage return *or* a linefeed. The supplied macro file has adopted the convention of using the linefeed separator.

The NUL Character (hex 00)

The NUL character (hex 00) is used internally by the editor to delimit strings. It is therefore not possible to save this character in your buffer. Should you attempt to enter this character, the line (text or command) will be truncated at that point.

The Meta Characters (@$^^&.*[)

When option meta-characters is *on* (**m+**), then these characters have a very special meaning when used within patterns (they are special only within patterns). The period (**.**) for example will match *any* character, not just a period. The meaning of these characters is explained in the section on pattern matching.

The Backslash Character (\)

The escape character on the command line is the backslash. When it precedes a meta character in a pattern it causes that character to be taken literally. That character loses any special significance it might have normally had.

Following a backslash by two hexadecimal characters in a pattern or translate string results in a single character with the hexadecimal value specified. For example **\0A** is the single character whose hexadecimal value is 0A (a linefeed character).

The Tab Character (hex 09)

When displayed on your screen this character will be expanded into the necessary number of spaces to move to the next tab stop. Tab stops are fixed at every four columns with the first stop set on column five. You can display tabs by turning on option tabs display (**ot+**).

Tabs are not treated with any special significance internally. They only affect your display and your cursor movement on the display. You can not position your cursor on the expanded spaces following the tab, only the tab character itself or the real character following it.

The Command Character (hex FF)

On input, the character with hexadecimal value FF will cause all characters up until the next record separator (newline) to be collected (no echo) in a hidden buffer, then executed as a command. Any current text on the command line is *not* affected. This character is used heavily by the translate command when defining macros for the various cursor and function command keys.

This character is only special on input. You can place a hexadecimal FF character in your text by using the substitute command and a **\ff** escape. e.g. to replace "C" with the hexadecimal character FF:

```
s/C/\ff/
```

The Recall Character (hex FE)

On input, the character with hexadecimal value FE will recall to the command line the last command typed. This character is used by the F9 and F10 function keys.

You can place this character in your text by using the substitute command as above.

The Keyboard Input Character (hex FD)

When this character is encountered in a macro, the editor will accept a character from the keyboard. If several characters occur in a row, a maximum of that number of characters will be accepted. Entering a carriage return will always terminate input (skipping any remaining FD's) and the carriage return will be discarded.

The Macro Disable Character (hex A3)

On input, the character with hexadecimal value A3 will prevent the next character from being expanded should a translate be in effect for it. For example, the Home key has a hexadecimal value of A0, but is translated on input into the three character string:

```
<command char>1<newline>
```

If you would like to prevent this expansion (to enter the key's value) then you should proceed it with the - key on the numeric keypad which generates the code for the Macro Disable character. You can of course enter the Macro Disable character itself by typing the - key twice.

The Condition Register

The editor maintains a special register called the condition register which is set to TRUE or FALSE by some of the editor commands. This register can be tested by the **Branch** (**b**) command and the **Until** (**u**) command to perform conditional execution of editor commands. These commands are commonly used in macros.

Delete Buffers

The Editor maintains a buffer for deleted characters and another buffer for deleted lines.

The Character Delete Buffer

The character delete buffer is arranged as a stack 256 characters long. Adding a character to a full buffer will cause the oldest character to be lost. In this manner the most recent 256 characters are kept.

The editor maintains primitive commands for:

- adding a character to the buffer.
- removing the last character which was added to the buffer.
- purging the buffer.

These primitives are provided by subcommands of the **Zap** (**z**) command.

The saving of the last deleted character via the Del key is performed by a macro which saves the character under the cursor in the character delete buffer before deleting it. Likewise, the restoration of a deleted character via the Ctrl – Ins key combination is based upon a macro which inserts the last character placed in the delete buffer before the current cursor position.

The Line Delete Buffer

The editor maintains another buffer in parallel with your text buffer called the line delete buffer. This buffer has the same structure as your text buffer, however, it can not be displayed or directly operated on by the editor's many commands. Each time you delete a line via the **Delete** (**d**) command it is moved from your text buffer into your line delete buffer. You can restore deleted lines using the special forms of the **Append** (**a**) and **Insert** (**i**) commands which can move lines from the delete buffer back to your text buffer.

The moving of lines between the two buffers is slightly more complicated than is indicated above and is best explained by an example.

If you were to delete 5 lines, one at a time (say via the F3 key) it would be nice if you could undelete them one at a time so that the last line deleted was the first line restored. This is particularly nice when you delete one line too many and just want to restore the last one, not all of them. Conversely, if you were to delete a group of 100

lines as a block (say via a tagged delete) you do not want to have to restore them one at a time but want them restored as a block as well. The above two scenarios describe, from a user's point of view, the editor's implementation of the line delete buffer. Associated with the buffer is a flag which indicates whether the buffer contains a series of single line deletes or one block delete. To avoid confusion, the editor will purge the line delete buffer before adding in the following circumstances.

- You perform a single line delete and the line buffer contains a block delete.
- You perform a block delete. A block delete always purges the line delete buffer before adding the new block.

Put simply, if you delete lines one at a time, they are undeleted one at a time and if you delete a block of lines they are undeleted as a block. Mixing blocks or types is prevented by purging before adding, if necessary.

When working with a very large buffer it is possible for the editor to run out of memory. When this happens it will purge the delete buffer in an attempt to free up some space. You will be warned of this by a message on the command line which you must clear (like an error) by typing a carriage return. Deleting all lines in a file, then attempting to edit another large file will often generate this message. In this case you have all of the original file in memory in the line delete buffer and are trying to read another large file into the text buffer. They may not both fit!

Break Handling

The editor will terminate any operation gracefully at the earliest possible moment after the Break key is typed. As a result of the break, any operation may be incomplete on the range of lines specified for a command, however, no line will be left in a partially modified state. Should you break out of an **`Edit`** (**`e`**), **`Read`** (**`r`**), or **`Write`** (**`w`**) command you may only move a subset of the lines into or out of your buffer to the specified file.

After servicing the Break the editor will leave you in command state.

The Pattern Matcher

The editor has a very powerful pattern matching facility which will match the class of patterns known as regular expressions. Patterns are used for line searches and by the **`Global`** (**`g`**) and **`Substitute`** (**`s`**) commands. It is the editor's pattern matching facility that gives it flexibility in writing powerful macros. For example, the Ctrl left and right arrow keys are implemented by a pattern which searches for the next or previous word in your text. We will attempt to describe the patterns accepted by the editor in a very rigorous manner. It is assumed that option meta characters is *on* (**`m+`**) during the definition of your pattern. If it is *off* (**`m-`**) then the editor will only recognize the class of patterns represented by (1) and (2) below.

1 The simplest pattern is a single character. Such a pattern matches the given character in either upper or lower case, unless option dual is *on* (**`d+`**) to make the pattern matcher differentiate between cases.

2 Patterns arranged adjacently form a single pattern. For example: **/abc/** matches any string “abc”.

3 The character **^** (caret) specifies a pattern which matches the null string at the beginning of the line. Thus a line search of: **/^charm/** would match the string “charm” at the beginning of a line.

4 The character **$** specifies a pattern which matches the null string at the end of the line. Thus a line search of: **/charm$/** would match the string “charm” at the end of a line.

5 The character **.** (dot) specifies a pattern which matches any character. Thus a line search of: **/charm./** would match the string “charm” followed by any character on a line.

6 Any pattern followed by a ***** defines a pattern which matches a string of zero or more occurrences of that pattern. Thus: **/b*/** matches the strings “b”, “bb”, “bbb”, etc. In addition, since *****-patterns will match zero occurrences of a given pattern, “/b*/” will also match “”. The pattern matcher will match a ***** construction with the longest sequence matching the given pattern beginning in a given column; for example, if a line contains the string “bbbbbb”, **/b*/** will match all six **b**’s as a unit, not individually.

7 The construction **@(***number***)** is a pattern which matches the null string immediately before the *number*th column on the line. Thus a line search of: **@(10)charm** would match the string “charm” starting in character position ten on the line.

If *number* is zero or the character **.** (dot) then this pattern will match the null string before the current cursor position in the text area.

If *number* is the character **t** then this pattern will match the null string before the next tab stop. This can be used to turn runs of spaces into tabs. See the **Substitute** (**s**) command.

8 The construction **[***string***]** matches any one character in string and no other. Thus: **/[0123456789]/** is a pattern which will match a single digit. Characters in the square brackets are taken literally, without their special meanings; so **/[.]/** will match the character “**.**” and no other. A sequence of characters may be specified by separating the lower and upper character by a dash (**-**). For example: **/[a-z0-9]/** is a pattern which will match any letter or any digit. To match a “-” you may protect it with the backslash (\) character.

9 The construction **[^***string***]** matches any one character that is *not* in string. Thus: **/[^^0-9]/** is a pattern which will match any character that is not a digit.

10 A null pattern (**//**) is equivalent to the pattern most recently specified within the editor. This feature is convenient for searching through a file for a particular string. For example, if you are looking for the string “hello” you could specify: **/hello/** the first time and: **//** on subsequent searches. An even quicker

method of performing this task would use the F9 key to simply re-execute your line search.

11 Any character preceded by the backslash character (\) loses its special meaning. Thus: **/\\\^^\.\$/** would match the string "\^^.$".

Some Pattern Examples

Match the string "hello" anywhere on a line:

```
/hello/
```

Match the string "hello" at the start of a line:

```
/^hello/
```

Match the string "hello" at the end of a line:

```
/hello$/
```

Match a line containing *only* the string "hello":

```
/^hello$/
```

Match all trailing blanks (including zero) on a line:

```
/  *$/
```

Match all characters (including zero) on a line:

```
/^.*$/
```

Match a number like "3", "862", etc:

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

Match a C language identifier:

```
/[a-z_][a-z_0-9]*/
```

Match a digit in column ten:

```
/@(10)[0-9]/
```

Match an empty line:

```
/^^$/
```

Match a line containing only blanks:

```
/^^*$/
```

Match a line starting with a period followed by a name (such as a command in the QNX2 **DOC** markup language):

```
/^\.[a-z][a-z]*/
```

Command Reference:

The pages following detail the all available editor commands, in alphabetical order.

Append (a)

append after current line

Syntax:

[*line***]a[***text***]**

[*line***]ad**

Description:

The **a** form of this command turns on option newline (**n+**) and opens up a newline after the current line. If *text* is specified then that text will be placed at the start of the newly opened line. This form is often used within a **Global** (**g**) command to append a string after a set of matched lines. The blank between the **a** and *text* is required.

The **ad** form appends the last deleted line (or range of lines) from the delete buffer. Option newline is not affected.

The append command must be the last command on a line.

Current line:

Set to the address of the new line opened up.

Condition register:

Not affected.

Branch (b)

branch

Syntax:

b*number***[t|f]**

Description:

This command allows you to branch over *number*-1 command lines. When neither **t** nor **f** are appended, the command branches unconditionally. If **t** is appended, it branches if the condition register is TRUE. If **f** is appended it branches if the condition register is FALSE. The condition register is not affected by the branch. Branches are often used inside macros defined by the **Translate** (**t**) command and make little sense when executed directly.

If *number* is zero then the current command line is re-executed. If *number* is one, then the rest of the current command line is skipped. If *number* is greater than one, then *number*-1 command lines will be read and discarded. A command line is considered as a string of characters terminated by a linefeed separator (**\0A**).

The **Branch** command cannot be used inside a **Global** (**g**) or **Until** (**u**) command.

Current line:

Not affected.

Condition register:

Not affected.

Change (c)

change lines

Syntax:

[*line_range***]c [***text***]**

Description:

This command deletes the specified range of lines from the text buffer and places them in the line delete buffer. It then turns on option newline (**n+**) and opens up one line for input. If *text* is specified then that text will be placed at the start of the newly opened line.

The change command is functionally equivalent to a **Delete** (**d**) command followed by an **Insert** (**i**) command.

Current line:

Set to the address of the new line opened up.

Condition register:

Not affected.

Delete (d)

delete lines

Syntax:

[*line_range***]d**

Description:

This command deletes the specified range of lines from the text buffer and places them in the delete buffer.

If the *line_range* specifies more than one line to be deleted *or* the delete buffer contains a previous delete of a range of lines, then the delete buffer is purged before appending the new lines.

If a delete of a single line is requested and the delete buffer is empty or contains lines which have been deleted one by one, then this new line is inserted at the beginning of the delete buffer.

Current line:

Set to the address of the line after the last line deleted.

Condition register:

Not affected.

Edit (e)

edit a new file

Syntax:

e [*filename***]**

ee [*filename***]**

Description:

This command deletes the contents of the current buffer (without placing the lines in the delete buffer) and reads in the lines associated with the specified file. Since the contents of the delete buffer are not affected, you can use it to move lines from one file to another. To do this, perform a **Delete** (**d**), **Edit** (**ee**) a new file, and then undelete the lines into the new file with the **Append** command (**ad** variant).

The **e** and **ee** commands read from the currently defined filename, or if a filename is specified they read from the indicated filename and makes that the current filename. The **e** form of the command will refuse to destroy a non-empty buffer which has been modified, while the **ee** form will act regardless of whether or not the current buffer has been saved.

The current filename can be queried or changed by the **File** (**f**) command.

The **Edit** command must be the last command on a line.

The **Edit** command cannot be used inside a **Global** (**g**) or **Until** (**u**) command.

Current line:

Set to the first line of the buffer.

Condition register:

Not affected.

File (f)

file query/set

Syntax:

f [*filename***]**

Description:

This command allows you to query or set your current filename which is used by default in the **Edit** (**e**), **Read** (**r**), **Write** (**w**) and **Execute** (**x**) commands.

When a filename is not specified, this command displays your current filename on the command line and waits for you to clear it by typing the carriage return key.

If a filename is specified, the command sets the current filename to be the filename specified.

The **File** command must be the last command on a line.

Current line:

Not affected.

Condition register:

Not affected.

Syntax:

[*line_range***]g/***pattern***/***more_editor_commands*

[*line_range***]g^/***pattern***/***more_editor_commands*

Description:

The **Global** command checks each line in the indicated line range for the presence of the indicated *pattern*. The first form (**g**) marks lines which contain the pattern while the second form (**g^**) marks lines which *do not* contain the *pattern*. The **Global** command then executes the following commands for each marked line, with the current line set at the marked line before executing. If no line range is specified then *all* lines are searched for the *pattern*. For example:

```
g/^Comment/d
```

would remove all lines containing the string “Comment” at the beginning of the line. To print them before deleting use:

```
g/^Comment/pd
```

To Change all occurrences of the string “minimum” to “maximum” and print the changed lines, issue a:

```
g/minimum/s//maximum/p
```

To delete all lines which don’t end in the string “.c” issue a:

```
g^/\.c$/d
```

To append the line “———-” after each line in the buffer, issue a:

```
g/^/a  -------
```

To reverse the order of lines in your buffer, issue a:

```
g/^/.m0
```

Finally, to print the lines around each line containing the string “function”, issue a:

```
g/function/.-5,.+5p
```

Current line:

When the **Global** command is finished, the current line has the value it had after the last command executed during the **Global** operation.

Condition register:

When the **Global** command is finished, the condition register has the value it had after the last command executed during the **Global** operation.

Insert (i)

insert before current line

Syntax:

[*line***]i [***text***]**

[*line***]id**

Description:

The first form of this command turns on option newline (**n+**) and opens up a new line before the current (or specified) line. If *text* is specified then that text will be placed at the start of the newly opened line. This form is often used within a **Global** (**g**) command to append a string before a set of matched lines. The blank between the **i** and *text* is required.

The second form (**id**) inserts the last deleted line (or range of lines) from the delete buffer before the current (or specified) line. Option newline is not affected.

The **Insert** command must be the last command on a line.

Current line:

Set to the address of the new line opened up.

Condition register:

Not affected.

Join (j)

join two lines

Syntax:

[*line***]j**

Description:

This command joins the indicated line to the line following it. As an interesting example:

```
uf1j
```

will attempt to join all lines in the file.

Current line:

Set to *line*.

Condition register:

Set if a join occurs. Will always be FALSE on last line.

Kopy (k)

kopy lines

Syntax:

[*line_range***]k***target_line*

Description:

This command "kopies" the indicated range of lines to the line following the specified *target_line*. The *target_line* may be zero to insert before line one but the *target_line* may *not* fall within the source *line_range*. The source lines specified by *line_range* are left untouched in the buffer.

The "k" for copy was chosen since the letter "c" was already in use by the QNX 2 line editor and it was felt that its function should be maintained in the fullscreen editor for consistency. It's not that we kan't spell.

Current line:

Set to the first of the kopied lines.

Condition register:

Not affected.

Learn (l)

learn

Syntax:

l *character*

Description:

All input until a Ctrl – Break will be saved as a macro. From this point, each occurrence of *character* will be replaced by the learned input stream. To learn a key sequence for the F1 key you would enter:

```
l \81
```

Current line:

Not affected.

Condition register:

Not affected.

Move (m)

move lines

Syntax:

[*line_range***]m***target_line*

Description:

This command moves the indicated range of lines from their current position in the buffer to just after the specified *target_line*. The *target_line* may be zero to insert before line one but may not fall within the source *line_range*.

Current line:

Set to the first of the moved lines.

Condition register:

Not affected.

Option (o)

option query/set

Syntax:

o{*option_character***}{***option_modifier***}**

Description:

The **Option** command allows you to change or query your editing options. The option selected is specified by the **{***option_character***}** and the operation is specified by the **{***option_modifier***}**.

The **{***option_modifier***}** may be one of **+**, **-**, **~** (tilde) or **?**. A **+** (plus) will turn an option *on*, a **-** (minus) will turn an option *off* and a **~** (tilde) will cause the option to toggle. A **?** does not affect the option, but it sets the condition register TRUE if the option is *on* and FALSE if the option is *off*. The **+**, **-** and **~** forms of the command do not affect the condition register.

The **{***option_character***}** is a single letter and specifies which option will be acted on. It may be one of **a**, **b**, **c**, **d**, **e**, **f**, **i**, **j**, **l**, **m**, **n**, **s**, **t** or **w**). Each option is briefly described in a subsection below.

Options

oa - Option Anchor

Whenever a pattern search is made, the editor will leave your cursor at the start (**a+**) or end (**a-**) of the pattern matched. Should you specify a search for the string “mouse” with **a+** your cursor will be anchored to the “m” in mouse. With (**a-**), the cursor would be positioned at the character after the matched string “mouse” (the character after the “e”).

ob - Option Blank

When viewing a piece of text on the screen, lines appear to be padded with blanks to the end of the screen. Since the editor reads files with variable sized lines containing 0 (a line containing only a carriage return) to 512 characters you may wonder whether the blanks at the end of the line are read or not. With (**b+**), the editor will allow you to differentiate between real blanks by displaying nulls (non-existent characters) as small centered dots.

oc - Option Command

This option indicates whether your active cursor is in the command area (**c+**) or text area (**c-**). It is often queried and set within macros. The F1, F2 and large + key are examples of macros which do this.

od - Option Dual

This option, like option anchor (**oa**) concerns pattern matching. If *off* (**d-**) then a search for “mouse” would match the string “MOUSE”, “MoUse”, “MOUsE” and so on in your text. With (**d+**) the pattern matcher differentiates between upper and lower case.

oe - Option Environment

This option is handled quite differently from the other options. It does not have an on or off state. Each time you specify an **oe+** command, your current options will be saved. They may be restored by an **oe-** command. The editor only maintains a single level of environment stacking. Multiple **oe+** commands simply overwrite previous saves and multiple **oe-** commands simple restore the last environment saved by an **oe+**. This command is useful within macros where it may be necessary to temporarily change an option to perform a required function. Only the current state (on/off) off each option is saved.

of - Option Fill This option when on (**f+**) will cause automatic filling of input lines at your defined right margin. The default right margin is column 60. Should you attempt to enter a character in this column, then any preceding characters of a word will be moved to the next line. Using this option you may enter text without having to type a carriage return to end each line. This is extremely useful when entering documentation and letters.

oi - Option Insert With (**i+**), all characters typed will be inserted before the cursor.

oj - Option Justify

With (**j+**), all filled lines will also be justified at your right margin.

ol - Option Limit This option will flash (**l+**) when a limited tag has been set.

om - Option Meta Characters

With (**m+**), meta characters are enabled during pattern matching. The meta characters "**@$^&*[.**" are explained in detail on the section describing the pattern matcher.

on - Option Newline

Whenever a carriage return is entered in the text area while this option is on (**n+**), a new line will open up after your current line to allow you to type in new text. This can be thought of as line insert mode and is similar to character insert mode (**oi**).

os - Option Autosave

With (**s+**), your buffer will be automatically saved in the file "**autosave**" after every 20 lines of input. While writing to the disk you may still continue to type up to 256 characters per line, however, your keys will not be echoed until the write is complete.

ot - Option Tabs This option is similar to option blank (**b**). Tab characters in your text are expanded into enough spaces to reach the next tab

stop. What may appear as 4 spaces may only be one real character. Turning this option on (**t+**) will display the actual tab character as a right triangle. With (**b+**) also on, the spaces which pad the tab to the next tab stop will be displayed as centered dots since they do not exist within the buffer. Tabs are characters which are heavily used within C programs for indentation.

ow - Option Wrap — With (**w+**) pattern searches will wrap around from top to bottom or bottom to top in your buffer.

Current line:

Not affected.

Condition register:

Set TRUE or FALSE if *option_modifier* is a "**?**".

Print (p)

print lines

Syntax:

[*line_range***]p**

[*line_range***]P**

Description:

This command clears your screen and prints the specified lines on your screen. If the number of lines exceeds the size of your screen, the command will pause until you type a character to continue. If the first form is used (small p), then any non-printing characters (less than hex 20 and greater or equal to hex 80) will be expanded into a *hh* sequence. This can be useful in finding out the hex values of some of the Function and Cursor keys (proceed them with the "-" on the keypad to put them in your text). This command is nearly always used in conjunction with the **Global** command. It allows you to display lines which are not adjacent in your buffer. For example:

```
g/index/p
```

will print all lines on your screen that contain the string "index". You may not edit the displayed lines using the cursor keys. The editor has temporarily dropped out of full screen mode.

Current line:

Set to the last line printed.

Condition register:

Not affected.

Quit (q)

quit

Syntax:

q

qq

Description:

This command terminates the current editing session. For your protection the editor will not let you quit via a single "**q**" if your buffer has been modified since the last time you saved it. You can force a quit without saving by using the second form, "**qq**". The **Quit** command must be the last command on a line and must not be within a **Global** command to be recognized.

Current line:

Not affected.

Condition register:

Not affected.

Read (r)

read a file

Syntax:

[*line***]r [***filename***]**

Description:

This command reads in the contents of a file and appends it immediately following the line *line*. If a *filename* is not given, the current file is used. If *line* is omitted, it will append the file after the current line.

The **Read** command must be the last command on a line.

The **Read** command can not be used inside a **Global** (**g**) or **Until** (**u**) command.

Current line:

Set to the address of the first line read from the file.

Condition register:

Not affected.

Substitute (s)

substitute text

Syntax:

[*line_range***]s/***pattern***/***replacement_text***/**

[*line_range***]s***number***/***pattern***/***replacement_text***/**

Description:

This command replaces occurrences of *pattern* with the string *replacement_text* in the line range specified. If no line address is specified, substitutions are made on the current line. The pattern and replacement text are delimited by a single character, usually "**/**". It is general practice to use the "**/**" character for this purpose, however, any other character not appearing in *pattern* or *replacement_text* may be used instead. When another character occurs in the place of "**/**" it will automatically be used as the delimiter character.

The first form replaces *all* occurrences of the given pattern on a line while the second form only replaces the *number*th occurrence on the line.

The "&" character has a special meaning (with option **m+**) when used in the replacement text. It will be replaced by the text of the pattern matched. For example:

```
s/however/&,/
```

is equivalent to:

```
s/however/however,/
```

This can be extremely useful when matching complex patterns where you may not know the exact text matched. The "&" is *not* a special character in *pattern*, and all the pattern matching meta characters are not special in *replacement_text*.

The backslash "\" character may be used in the definition of *pattern* and *replacement_text* to escape the meta characters, "&" and the delimiter character (usually "**/**") so they lose their special meaning. For example:

```
s/myfile\*/\/yourfile\&/
```

will replace the string "myfile*" with "/yourfile&". You can also match and replace your text with exact hexadecimal character values using a *hh* sequence. One common use is splitting a line by inserting a record separator character. The command:

```
s/and further more/&\0a/
```

will split the current line after the string "and further more". You can not match a linefeed character in you text buffer as they are not actually stored, but stripped and added when you read and write your buffer to a file.

The **@(t)** construct can be used to turn runs of spaces into tabs

Mark tab stops with a Ctrl – a:

```
s/@(t)/\01/
```

Replace runs of spaces ending on a tab stop with a Tab character:

```
s/  *\01/\09/
```

Current line:

Set to the address of the last line of the specified range.

Condition register:

Set TRUE if a successful substitution takes place, otherwise, it is set FALSE.

Translate (t)

translate a key on input

Syntax:

t *character replacement_text*

t *character*

t ? *character*

T *character replacement_text*

T *character*

T ? *character*

Description:

This command translates an input character into a string of characters. It is this command which makes the macro definition of the function and cursor keys possible.

The first form translates the indicated character *character* into the string of characters *replacement_text* whenever that character is typed. The second form removes any translation in effect.

The third form will display the current translation of the indicated character on the command line allowing you to modify it if desired.

The definition of *character* may be a *****hh sequence. If you wish to enter the character to be translated directly (you may not know its hexadecimal value) you should precede the character with the "-" key on the numeric keypad. This suppresses any translation currently in effect for the key. It is recommended that you *not* translate the "-" or "\" keys. You should keep in mind that one level of backslash escapes is stripped off during the translate command. This unfortunately means that if you wanted to match a backslash in a substitute command, you would have to type four of them to get one.

Entering this:

```
t  \84  \ffs/\\\\/?/\0a
```

would save the following for the definition of F4.

```
\ffs/\\/?/\0a
```

The **Substitute** command would turn the **** into a single **** when the macro was executed.

If you use the little "**t**" command then recursive definitions are allowed. If you use a capital **T**, then a macro within a macro will not be expanded.

The special character **\ff** (see section on special characters) should be prefixed to any translation that you wish executed as a command. It causes all characters up to the next record separator **\0a** to be collected (no echo) in a hidden buffer and then

executed as a command. Any current text on the command line is *not* affected. This allows you to define command translations which are independent of where your active cursor is. For example:

```
t  \01  \ff/procedure/\0a
```

would cause a Ctrl – a to be translated into a scan for the pattern **/procedure/**. Note that the **\0a** was necessary to force execution. Without it you would have to type the carriage return key yourself at the keyboard. If we were to omit the **\ff**, then the text would be entered at the active cursor which would probably be ok if you were on the command line, but unpleasant if you were in the text area.

As a further example, to translate the F1 key into the string "QNX Software Systems" you would enter the command:

```
t  \81  QNX Software Systems
```

Note that we did *not* prefix the replacement text with a **\ff**, nor did we end it with a **\0a**. Anytime you now type an F1, the string "QNX Software Systems" will be entered at the active cursor, exactly as if you had typed the characters on the keyboard yourself.

Current line:

Not affected.

Condition register:

Not affected.

Until (u)

until

Syntax:

[*line_range***]u***count* *more_editor_commands*

[*line_range***]u***condition* *more_editor_commands*

[*line_range***]u***count***[***condition***]** *more_editor_commands*

[*line_range***]u** *more_editor_commands*

Description:

This command is used to repeat a list of commands a number of times until a given condition is satisfied. The list of commands may contain another **Until** if desired, however, it may *not* contain a **Branch** (**b**) command. The space between the *count* or *condition* and *more_editor_commands* is required.

The first form will repeat the list of commands *count* times.

The second form will set the condition register to the opposite of *condition* and repeat the entire list of commands until the condition register matches *condition at the end* of one of the repetitions of the command list. The *condition* should be specified as a single character which may be **t** for TRUE or **f** for FALSE.

The third form will set the condition register as in the second form but will repeat the list until either *count* is reached, or the condition register matches *condition*, whichever occurs first.

The final form will repeat forever, or until you type Break (or an error occurs).

In all cases the **Until** command will terminate immediately if any command in the list generates an error. In this case the condition register will be in the state it had before the error. The error will *not* be printed, but absorbed by the **Until** command. This allows you to prevent a command from issuing an error message by preceding it with a "**u1** ". For example:

```
u1  s/IBM/I.B.M/
```

will not generate an error if the substitute fails.

If both *count* and *condition* are omitted, then the **Until** will repeat until an error occurs or you type Ctrl – Break.

Current line:

When the **Until** command is finished, the current line will have the value it had after the last command executed during the **Until** operation.

Condition register:

When the **Until** command is finished, the condition register will have the value it had after the last command executed during the **Until** operation.

View (v)

view screen options

Syntax:

va*number number number*

vc[*number***]**

vf

vl[+|-]*quantity*

vr[+|-]*quantity*

vs*quantity*

vt[2|4|8]

vz[*number***]**

Description:

This command allows you to change some of the parameters associated with your screen. The parameter is specified by the character following the **View** (**v**) command. The condition register is not affected by this command.

Each parameter is briefly described in the subsection below.

View screen options

va - Attribute **va***number number number*

This command allows you to change the attributes (color) of the three regions of your display. The first number must lie between 1 and 3 and indicates the region.

- **1** - Status line
- **2** - Command line
- **3** - Text lines

The second and third number select the foreground and background color.

0 - Black **1** - Blue **2** - Green **3** - Cyan

4 - Red **5** - Magenta **6** - Yellow **7** - White

The foreground color will be intensified if values greater than 7 are used:

8 - Black **9** - Blue **10** - Green **11** - Cyan

12 - Red **13** - Magenta **14** - Yellow **15** - White

This command is ignored when used with the monochrome display. Users with a color display may wish to add this command to the end of the file **/usr/lib/ed.macros**.

vc - Center Line **vc**[*number*]

When used without an argument this causes your screen to be redrawn with the line your cursor is on positioned at your defined center line. You may redefine your center line by specifying a number between 1 and 23. The default macro file sets your center line at 3.

vf - Full Display of Text and Attributes

vf

The editor keeps track of the attributes stored on each line. To increase the speed of screen updates on the console, **qed** will not update the attributes when it believes they have not changed. Programs which modify the screen attributes without the editors knowledge may result in permanent attributes which **qed** will not remove. The view full command will force **qed** to always update the attributes as well as the text on your screen.

vl - Left Margin **vl**[**+|-**]*quantity*

This command defines your left margin. The *quantity* may be a simple number in which case your margin will be set to that value, or it may be a number preceded by a "**+**" or "**-**" in which case that value will be added or subtracted from the current value. This lets you define absolute or relative changes to your margin. For example:

vl+5

will increase your left margin by five and:

vl5

will define your left margin to be five. As a special case, a "**.**" (dot) will set your margin to the column that your text cursor is currently on.

Your left margin defines the point you will return to in your text whenever you type a carriage return. The characters before your margin will be filled by blanks unless you have changed your default fill character (see the **Zap** (**z**) command).

Your left margin is constrained to lie between 1 and your right margin.

vr - Right Margin **vr**[**+|-**]*quantity*

This command defines your right margin in the same manner as **vl** defined your left margin. Your right margin defines the point at which filling (automatic generation of a carriage return) will occur on text entry if you have option fill on (**f+**).

Your right margin is constrained to lie between your left margin and column 1000.

vs - Scroll Screen **vs***quantity*

This command causes your screen to be scrolled. If you specify a negative quantity then the scroll will be backward and if you specify a positive quantity then the scroll will be forward. This command always causes an immediate refresh of your screen. This allows you to look at snap shots of your screen during the execution of an until (or any list of commands). The command:

```
u20  s/^/+/
```

would only display the final version of the line you modified when the editor finished execution. The command:

```
u20  s/^/+/vs0
```

would give you a snap shot after each iteration of the substitute command.

vt - Set tab settings

vt[**2|4|8**]

You may set your tab stops at every 2, 4 or 8 characters on the screen. The default is 4.

vz - Zoom the size of your screen

vz[*number*]

When given without arguments this command will switch to the next hardware supported screen size. If the number is given it will switch to

vz0 Size of screen upon entry.

vz1 Screen size 1 (usually 25 by 80).

vz2 Screen size 2 (usually 43 by 80).

vz*n* Hardware dependent.

Current line:

Not affected.

Condition register:

Not affected.

Write (w)

write buffer to a file

Syntax:

[*line_range***]w [***filename***]**

[*line_range***]wa [***filename***]**

Description:

This command writes out the specified lines to a file. If a *filename* is not given the current file is used. If the *line_range* is omitted then *all* lines are written.

If the **wa** form is used, the editor will append the specified lines to the end of the file.

If a *filename* is specified and the current filename is not defined, then the specified *filename* will become the current filename. Note that *filename* can of course be a device like **/dev/par**.

Current line:

Set to the address of the first line written.

Condition register:

Not affected.

Execute (x)

execute a file of editor commands

Syntax:

x [*filename***]**

Description:

This command will open up a file and execute its contents as a series of editor commands. If a *filename* is not given the current file is used.

This command is commonly used to load macros via a file of translates. The editor does an **Execute** on the file **/usr/lib/ed.macros** immediately after execution begins. An execute file may *not* contain another execute.

The **Execute** command cannot be used inside a **Global** (**g**) or **Until** (**u**) command.

Current line:

The current line is set by the commands in the execute file.

Condition register:

The condition register is set by the commands in the execute file.

Yut? (prompt) (y)

Yut? (prompt)

Syntax:

[*line_range***]y"***prompt_text***"***command_chars***"***arguments*

Description:

This command does not know what it is. It will print *prompt_text* on the command line then wait for you to type the letter of a command. If the character you type matches the *n*th character of *command_chars* then control will transfer to the *n*th line. Each line is separated with a linefeed (**\0A**).

For example,

```
y"k for kopy or m for move"km"\0a\ff#k.b2\0a\ff#m.\0a
```

would prompt:

```
k for kopy or m for move
```

on the command line and pause for you to type a character. If you typed an 'm' then the command:

```
#m.
```

would be executed, which happens to be a tagged move to the current line.

If the typed character is not found, then the last command will be executed. The **n** command is an invalid command and can be used to generate errors on invalid keys. In the following example, a space or any invalid key will generate an error:

```
y"k for kopy or m for move"km  "\0a\ff#k.b3\0a\ff#m.b2\0a\ffn\0a
```

Current line:

Set according to the command executed.

Condition register:

Set according to the command executed.

Syntax:

[*line_range***]zcc***character*
[*line_range***]zcd**
[*line_range***]zcD***location*
[*line_range***]zce***location*
[*line_range***]zcf***character*
[*line_range***]zch***location*
[*line_range***]zcl**
[*line_range***]zcp**
[*line_range***]zcr**
[*line_range***]zcR***number*
[*line_range***]zcs**
[*line_range***]zh**
[*line_range***]zk***line*
[*line_range***]zlc**
[*line_range***]zld**
[*line_range***]zle**
[*line_range***]zlf**
[*line_range***]zlj**
[*line_range***]zlo**
[*line_range***]zlp**
[*line_range***]zlq**
[*line_range***]zlr**
[*line_range***]zlR["***filename***"]**
[*line_range***]zls**
[*line_range***]zlt**
[*line_range***]zlu**
[*line_range***]zlw["***filename***"]**
[*line_range***]zm***text*
[*line_range***]zp**
[*line_range***]zq***c*
[*line_range***]zv***digit*

Description:

This command consists of a number of useful subcommands which relate directly to the capabilities of a full screen editor. The subgroup starting with "**zc**" (**Zap Cursor**) deal with the cursor and are further subdivided by a third character to indicate a particular function. The *line_range* for the "**zc**" group may be a single line address of zero in which case the command will be applied to the command line instead of the text area. Each **Zap** function is described in a subsection below.

zcc - Zap Cursor Change

[*line_range*]**zcc***character*

This function changes the character under the cursor to the character specified. The *character* may *not* be a *hh* escape sequence but may be a carriage return. If option insert is *on* (**i+**) then *character* is inserted *before* the cursor.

Condition Register: Not affected.

zcd - Zap Cursor Delete

[*line_range*]**zcd**

This function deletes the character under the cursor causing the rest of the line to Shift to the left.

Condition Register: Trying to delete a non-existent character (you are to the right of the end of the line) will set the condition register FALSE. A successful delete sets it TRUE.

zcD - Zap Cursor Delete Multiple

[*line_range*]**zcD***location*

This function will delete all characters from the current cursor position to the position specified by *location*. The *location* is specified in the same manner as the **zch** command.

Condition Register: Trying to delete a non-existent character (you are to the right of the end of the line) will set the condition register FALSE. A successful delete sets it TRUE.

zce - Zap Cursor Erase

[*line_range*]**zce***location*

This function will erase all characters from the current cursor position to the position specified by *location*. This allows erase to end of field (right margin) to be implemented via a

zcer

The *location* is specified in the same manner as the **zch** command. If the field does not end the line, the field is replaced with blanks instead of being deleted. This preserves column integrity.

Condition Register: Set TRUE if any characters are erased.

zcf - Zap Cursor Fill

[*line_range*]**zcf***character*

This function defines the fill character to be used whenever you type a character to the right of your text (indicated by a small centered dot in option blank (**b+**)). The default is to extend the line with blanks. However it may be changed to any character, a period being a typical example. The *character* may *not* be a *hh* escape sequence.

The fill character is often changed to a Tab when used in conjunction with the Ctrl – b (begin indent) and Ctrl – e (end indent) keys when writing programs.

Condition Register: Not affected.

zch - Zap Cursor Horizontal

[*line_range*]**zch***location*

This function moves the cursor horizontally on the line. The movement may be absolute, relative or based upon a pattern. The field *location* determines the type of movement and can be one of:

Location	Movement
number	Absolute movement to indicated character *number*.
+*number*	Relative movement forward *number* characters.
-*number*	Relative movement backward *number* characters.
$	Move to end of line.
l	Move to left margin.
r	Move to right margin.
/*pattern***/**	Starting at the current cursor attempt to match the indicated pattern. Note that the line is assumed to start at the cursor and all characters to the left of the cursor are ignored. The pattern **/^./** would match the character at the cursor *not* the beginning of the line. Option anchor will determine the new location of the cursor on a match.
?*pattern***?**	Starting at the current cursor and scanning *backwards*, attempt to match the indicated pattern. Note that the line is assumed to start at the cursor and extend backwards to the beginning of the line. All characters to the right of the cursor are ignored. Due to the backward scan, the line will appear reversed to the pattern. To match the string "IBM" you would specify the pattern **?MBI?**. This is only true of backward searches using the **zch** command.

In all cases above, the cursor is limited to lie between index 1 and 512.

Condition Register: An attempt to move the cursor outside the text on the line (or a pattern match fail) will set the condition register FALSE, otherwise, it is set TRUE.

zcl - Zap Cursor Lock

[*line_range*]**zcl**

When the editor is forced to move the cursor to a line which is not within the currently displayed screen, it will by default redisplay with the cursor line positioned at your defined center

line (see **vc***number*). The **Zap Cursor Lock** command *locks* the cursor to the *top* or *bottom* of the screen on a redisplay. Conceptually, if your cursor must move off the top of your screen then, it will remain locked at that position and your text will scroll down as required. Likewise, if your cursor must move off the bottom of your screen, then it will remain locked at that position and your text will scroll up as required.

As an example the up-arrow (↑) and down-arrow (↓) are implemented as:

```
.-1|zcland.+1|zcl
```

to prevent a re-centering of your display when your cursor attempts to leave the screen.

Condition Register: Not affected.

zcp - Zap Cursor Purge

[*line_range*]**zcp**

This command purges all characters in the character delete buffer.

Condition Register: Not affected.

zcr - Zap Cursor Restore

[*line_range*]**zcr**

This command causes the last character put in the delete buffer to be restored at the current cursor position. If option insert is on (**i+**) the character will be inserted before the cursor.

Condition Register: Set TRUE if the character delete buffer contains at least one character to restore.

zcR - Zap Cursor Restore Multiple

[*line_range*]**zcR***number*

This command will restore the last *number* characters put in the delete buffer at the current cursor position. If option insert is on (**o+**) the characters will be inserted before the cursor.

Condition Register: Not affected.

zcs - Zap Cursor Save

[*line_range*]**zcs**

If the cursor is positioned on a character, then it is saved in the character delete buffer. If the cursor is not on a character (off to the right of the line) then nothing is saved.

The character delete buffer may hold a maximum of 256 characters. Saving more than this number causes any new

character to be inserted at the front and the oldest character is lost. Only the last 256 characters are kept.

Condition Register: Set TRUE if a character is saved.

zh - Zap Home [*line_range*]**zh**

This command will move your cursor to the top left hand corner of a tagged block of text. For example, say you wish to write a macro which saves a tagged block of text to a file, executes the **sort** command and replaces the tagged tagged text when sorted. It is necessary to ensure that the cursor is positioned at the top left corner of the text when you read the file back in.

```
t \01 \ffzh\0a\ffzp#zlw"tmp"!!sort tmp +r\0a\ffzpzlR"tmp"\0a
```

Condition Register: Not affected.

zk - Zap Kopy [*line_range*]**zk***line*

This command provides a convenient method of kopying a text line to the command line and vise versa. A line address of zero designates the command line. For example:

Kopy command line to current line:

```
0zk.
```

Kopy current line to command line:

```
.zk0
```

Condition Register: Not affected.

zlc - Zap Line Center

[*line_range*]**zlc**

This command will center the text between the left and right margins of the indicated lines.

Condition Register: Not affected.

zld - Zap Line Delete

[*line_range*]**zld**

This command will delete the text between the left and right margins of the indicated lines.

Condition Register: Not affected.

zle - Zap Line Erase

[*line_range*]**zle**

This command will erase the text between the left and right margins of the indicated lines.

Condition Register: Not affected.

zlf - Zap Line Fill [*line_range*]**zlf**

This command will fill the text between the left and right margins of the indicated lines.

Condition Register: Not affected.

zlj - Zap Line Join

[*line_range*]**zlj**

This command will set the join flag on the indicated lines.

Condition Register: Not affected.

zlo - Zap Line Overstrike

[*line_range*]**zlo**

This command will set the overstrike flag on the indicated lines.

Condition Register: Not affected.

zlp - Zap Line Paragraph

[*line_range*]**zlp**

This command will set the paragraph flag on the indicated lines.

Condition Register: Not affected.

zlq - Zap Line Query

[*line_range*]**zlq**

This command will set the condition code TRUE if there are any marks set.

Condition Register: TRUE if any marks set.

zlr - Zap Line Restore

[*line_range*]**zlr**

This command will restore the contents of the delete buffer where the current text cursor is. The delete buffer is not affected.

Condition Register: Not affected.

zlR - Zap Line Restore File

[*line_range*]**zlR**[**"***filename***"**]

This command will read the contents of a file into the delete buffer then restore the contents of the delete buffer where the current text cursor is. If no file name is specified, then the file **/tmp/***groupID***.***userID* will be used where group and member are numbers.

Condition Register: Not affected.

zls - Zap Line Save

[*line_range*]**zls**

This command will save the text between the left and right limit at the end of the delete buffer. Limits are set by tags.

Condition Register: Not affected.

zlt Zap Line Tag [*line_range*]**zlt**

This command tags the indicated line if it was untagged and untags it if it was tagged. If there are already two tags in effect then this tag will replace one of the tags. NOTE: If two **zlt** commands occur within 1/2 second then option limit will be enabled and the line will not be untagged.

Condition Register: Not affected.

zlu - Zap Line Untag

[*line_range*]**zlu**

This command removes all tags in the file if any are set or sets the last tags if there are not any tags set. It is a toggle.

Condition Register: Not affected.

zlw - Zap Line Write

[*line_range*]**zlw**[**"***filename***"**]

This command will save the text between the left and right limit at the end of the delete buffer. Limits are set by tags. The data is then saved into a file. If no file name is specified, then the file '**/tmp/***groupID***.***userID*' will be used where group and member are numbers.

Condition Register: Not affected.

zm - Zap Message [*line_range*]**zm***text*

This command will print the message which follows on the command line. This is useful as a prompt.

Condition Register: Not affected.

zp - Zap Purge [*line_range*]**zp**

This command purges all lines in the line delete buffer.

Condition Register: Not affected.

zq - Zap Query [*line_range*]**zq***c*

This command sets the condition code based upon the character which follows the **zq**. If *c* may be **#** or **!**.

Condition Register:

TRUE if any tags have been set

! TRUE if zero exit status of last command

zv - Zap Version [*line_range*]**zv***digit*

This command sets the condition code based upon the version number of the editor. The version is a number between 0 and 9. If the editors version is greater than or equal to the digit specified the condition code will be set TRUE.

e.g. Set condition code TRUE if editor version 1 or later:

```
zv1
```

Condition Register: TRUE if editor is greater than or equal to the indicated version.

Current line:

In all cases, it is set to the last line of the specified range; however, not all **Zap** commands use the line range information (**zp** for example).

Condition register:

Depends on the specific **Zap** command; see description.

Chapter 4

Defining Your Own Macros

This section is intended for the advanced user who wishes to add functions to the supplied macro file **/usr/lib/ed.macros** or define a set of new macros for a particular editing task at hand. Before trying your hand at new macros you should have read both the Tutorial Guide and Reference Manual in detail. Macros are basically very simple to write, but only if you have a good grasp of the many editor commands described in the reference manual.

What is a Macro

A macro is just the simple replacement of an input key with a string of characters. The replacement string contains the keys you would have to type to perform the required function manually. As an example, let's look at the definition of the Ins key. Each time it is typed, it toggles option character insert (**oi**). You could perform this task manually by going to the command line and typing the command

```
oi~
```

which will toggle the option. You would then want to go back to the text area if that's where you came from. If you were already on the command line with a partially typed command then the situation is slightly more complex. You can't type in a new command on the command line without deleting what is currently there. To overcome this difficulty, the editor provides a very special character **{***command_char***}** which causes all input following it, up to the next newline to be collected (without echo) in a hidden command buffer which is then executed. By prefixing all commands with this character you can execute commands regardless of whether you are on the command line or in the text area. The value of the command character is hexadecimal FF and may be entered directly from your key board by holding down the Alt key and typing the Backspace key normally used for character delete. You can now toggle insert mode at *any* time by typing the string

```
{command_char}oi~<newline>
```

where **{***command char***}** is an Alt – Backspace and **<**newline**>** is either a carriage return or a linefeed. To turn this into a macro for the Ins key you would translate the value generated by the Ins key into the above string.

```
t \ab \ffoi~\0a
   |    |    |
   |    |    +--  Linefeed character.
   |    +-------  Command character.
   +------------  Hexadecimal value generated by
                  the Ins key.
```

If you leave off the command char then the string

```
oi~
```

followed by a newline would be entered at the active cursor. Likewise, if you leave off the newline, then the command will be collected, but you will have to supply the newline yourself by typing a carriage return.

Most macros are this simple. However, care should always be taken to ensure reasonable behavior when a requested operation may fail. An example of this is the down arrow key on the keypad. It is designed to move you down one line. This could be performed by executing the command

```
.+1
```

which works fine until you try to move past the last line in your buffer where you will be greeted by an unpleasant error message which will quickly annoy any user. The simple solution is to limit the line address to remain within the buffer using the **|** operator. The resulting macro definition would be:

```
\ff.+1|\0a
```

which behaves in a friendlier fashion. As one final point, to prevent your screen from being re-centered when you attempt to leave the screen, you might add a **Zap Cursor Lock** command resulting in

```
\ff.+1|zcl\0a
```

Lets look at one more simple example illustrating the importance of planning your macro's behavior in all possible situations. You want the PgDn key to display the next page of your buffer. Your text area is 23 lines so you might simply attempt to move forward 23 lines via a

```
.+23
```

This has two problems

1. You may not have another 23 lines in the buffer.
2. If your current line was not on line 1, then you would miss part of the next page. You should be jumping forward relative to the address of the top line of the current screen. The PgDn is currently defined as

```
t \aa \ff@+23|\0a
   |       | |
   |       | +----  Limit address to lie within buffer.
   |       +------  Address of top line on the screen.
   +--------------  Hexadecimal value generated by the
                    PgDn key.
```

Multi-line Macros

Most of your macros will consist of one or more editor commands entered as a string which makes up a single command line. There are cases, however, where you will find that you need to enter a multi-line command. You may do this by simply embedding **<**newline**>** characters in your translate string. The macro

```
t \84 \ffu1 *s/register//\0a \ffu1 *s/short/int/\0a
   |   |                 |   |
   |   |                 |   +-  Start of line two.
   |   |                 +-----  End of line one.
   |   +-----------------------  Start of line one.
   +---------------------------  Hexadecimal value
                                 generated by F4 key.
```

defines a macro which will remove all occurrences of the string “register” and change all occurrences of the string “short” to the string “int”. The **Until** one (**u1**) will prevent an error from being printed if the substitute fails to find a match. By placing them on two lines we are guaranteed to perform the second substitute even if the first one fails. Remember, an error (even inside an **Until**) terminates execution of the current line.

Macros Containing Branches

The most common use of multi-line macros involves the **Branch** (**b**) command. This command was included in the editor to allow you to perform conditional execution of editor commands within macros. A simple example is the large + key on your keypad which performs one of two tasks. If you are in the text area it simply places you on the command line. If, however, you are already on the command line, it simulates your entering a carriage return to execute any command, then places you back on the command line. Hitting carriage return would normally execute any command then put you back in the text area. This key is defined as.

```
t \a7 \ffoc?b2f\0a\0a\ffoc+\0a
   |     |         |    |
   |     |         |    +--  Return to command mode.
   |     |         +-------  Literal newline to execute
   |     |                   command line.
   |     +-----------------  Query and set condition
   |                         register if on command line.
   +-----------------------  Hexadecimal value generated
                             by the large + key.
```

This macro is interesting for two reasons:

1. It makes use of the **Branch** (**b**) command to conditionally execute editor commands. If you are in the text area it will skip two **<**newline**>** characters and only execute the “**\ffoc+\0a**” which will place you in command mode.

2. It mixes literal text (which is entered at the current cursor) with command text (which is preceded by a **\ff** and is collected in an execute buffer). If you were already on the command line it will not branch, and therefore enter a **<**newline**>** on the command line which will cause it to be executed. You will then fall into the code which sets option command and goes to the command line.

Defining complex macros which contain several branches and several lines quickly becomes confusing when displayed as a single line with embedded **<**newline**>** character escapes. The macro for the F2 key illustrates this.

```
t \82 \ffon?b3t\0a\ffi\0a\ffb4\0a\ffon-zch1\0a\ffb2t\0a\ffd\0a
```

This macro can be better understood when displayed as the multi-line sequence

t \82 \ffon?b3t

	Check if option newline is on or off.
\ffi	If it is off, then do an insert command
\ffb4	and skip the rest of the macro.
\ffon-zch1	Else go to the beginning of the line and skip
\ffb2t	to the end of the macro if line isn't empty.
\ffd	Delete the empty line.

It should be remembered that the **Zap Cursor Horizontal** command (**zch**) will set the condition register false if you move to a position which does not contain a character. If there is no character in the first column then the line must be empty.

There is a macro which will allow you to type in a macro in your text buffer in the multi-line form above and convert it into a single string on the command line which you may type carriage return to enter. This macro is defined in your **/usr/lib/ed.macros** file, but has been commented out by a double quote. You may remove this quote and then execute the file

x /usr/lib/ed.macros

to define this macro as your Alt – m key. If you type the Alt – m key in the text area, it will compress your buffer into a single line and move it to the command line for execution. If you are on the command line, then it will take a macro which you have displayed via the

t ? \hh

command and place it in your text buffer in a multi-line format.

For interest this macro is defined as:

```
t \84 \ffoc?b7f*da
\ffon-om+0zk1
\ffu1 s/\\\\\\\\\/\\80/
\ffu1 s/\\\\0a/\\0a/
\ffu1 *s/\\80/\\\\\\\\\/
\ff0zce255 1zch1b5
\ffom+                          -Start of code in the case
\ffu1 *s/$/\\\\0a/               where you are in the text area.
\ffu40 1j
\ff1zk0zch1oc+
```

This macro uses **Until** (**u**) commands to prevent errors should a substitute fail.

In the case where you are on the command line, it deletes your buffer and places an empty line in it via the **Append** command. It then copies the command line to the empty line and splits the line at each **\0a**. Note that it is not tricked by a **\\0a** sequence.

The reverse process appends each line with a **\0a** sequence, joins all lines, then copies it to the command line. It leaves the joined line in your text buffer allowing you to save it with a write or write append command.

Suggestions

At this point you should examine the macros defined in the file **/usr/lib/ed.macros** to gain further insight into the writing of new macros. You may examine them by reading the file, or by displaying each key's translation one at a time via the "**t ? \hh**" command and then using the F4 key to show it as a nice multi-line sequence. If you do not know the hexadecimal value of a key, then you may enter it's literal value by preceding it with the macro disable key (- key on the keypad). For example, you could type either

```
t ? \81
```

or

```
t ? <MINUS key><F1 key>
```

The possibilities for macros are endless, but you are warned that creating them is very much a black art...

Chapter 5

Appendix A - Error Messages

The Full Screen Editor prints any errors it detects on the command line and pauses for you to type a carriage return to clear the error.

`out of memory` The Editor was unable to allocate space to replace an existing line or add a new line.

`unknown command` The editor encountered an unknown major command or subcommand character.

`invalid pattern specification`

A pattern was specified which was not acceptable because:

- Terminating delimiter of pattern missing.
- Pattern exceeds 128 characters.
- Pattern starts with a "**`*`**".
- **`//`** pattern specified when no pattern was defined.

`buffer has been modified, use qq to quit without saving`

An attempt to leave the editor via the quit command when your text buffer has been modified since your last write to a file.

`invalid line number or line range`

A command has detected an invalid line number or range of line numbers. This can be caused by:

- *line1*,*line2* where *line1* is greater than *line2*.
- *line* is greater than the last line in the buffer.
- *line* is zero on a command other than **`a`**,**`k`**, **`m`**,**`r`** or **`z`**.
- Target of a move or kopy falls within the source range.

`x command encountered within an execute file`

An attempt has been made to nest execute commands.

`current file not defined`

An **`e`**,**`r`**,**`w`** or **`x`** command was entered without a filename and no current filename was defined. You can define it with the **`f`** command.

`unable to access file`

The file system was unable to open the requested file. This can be caused by:

- Invalid filename specified.
- No permissions to open file. (read, write or append)
- Filename specified can not be found. Check spelling.

syntax error A recognized command was specified incorrectly. Some commands which do not take a line range will flag a syntax error if given one (e.g.: **e**, **f**).

filename too long

A pathname of more than 64 characters was specified on **a** **e**,**f**,**r**,**w** or **x** command.

The QNX maximum pathname size is 255 characters, however the editor will accept only pathnames up to 64 characters in length. You may have to **cd** to a directory first in order to edit some files.

unknown option An unknown option character was specified in the option command.

line greater than 512 chars

On a read from a file, a line greater than 512 characters was encountered. The line is broken at 512 characters. Reading will continue when you type carriage return. The **j** and **s** commands can also cause this error.

pattern not found

The pattern specified in a line search or a substitute command was not found anywhere in the buffer.

disk error An error has occurred while trying to read/write a disk file. Make sure that the disk is not write protected (if writing). Otherwise, this error may signal a defective diskette or a bad block.

Attempt to write to a file which is not the current file. Use ww to force.

You are trying to write to a named file which is different from the one you edited. Perhaps you wanted to type "**e** *file*" and missed the "**e**" and hit the "**w**" instead. If you are sure, recall the line using the F10 key and insert a second "**w**" for a command of "**ww** *file*".

Chapter 6

Quick Reference

Control Keys

keypad +	Go to the command area. If you are already in command area then execute any typed command and remain in command area (i.e. same effect as hitting carriage return, except that instead of leaving the command area and returning to the text area, you stay in the command area ready to enter another command.)
keypad -	Take next key typed literally. This disables macro expansion and allows you to enter the function and cursor keys as data.

Cursor Movement Keys

Home	Position the cursor at the first character of the first line of the file.
Ctrl – Home	Position the cursor at of the first line of the currently displayed page.
End	Position the cursor at the last character of the last line of the file.
Ctrl – End	Position the cursor at the last line of the currently displayed page.
PgUp	Display previous page. (Back 23 lines)
Ctrl – PgUp	Scroll the screen back 1 line.
PgDn	Display next page. (Forward 23 lines)
Ctrl – PgDn	Scroll the screen forward 1 line.
↑	Move cursor up one line.
Ctrl – ↑	Move cursor up 4 lines.
↓	Move cursor down 1 line.
Ctrl – ↓	Move cursor down 4 lines.
←	Move cursor left 1 column.
Ctrl – ←	Move cursor to start of prev word.
→	Move cursor right 1 column.
Ctrl – →	Move cursor to start of next word.
Keypad 5	Redisplay with cursor line as defined center line.
Ctrl – Tab	Move to end of line.
Shift – Tab	Move to start of line.
Ctrl – b	Begin an automatic indent. (+4 columns).
Ctrl – e	End an automatic indent. (-4 columns).

Character Editing Keys

Backspace	Delete previous character.
Ins	Toggle character insert mode.
Ctrl – Ins	Insert last deleted character from delete buffer before cursor.
Del	Delete character at cursor. Character is saved in delete buffer.
Ctrl – x	Erase all characters on the line.
Ctrl – y	Erase all characters from cursor to end of line.

Function Keys

F1	Toggle newline mode. If entering newline mode then append after current line. If leaving and current line is null then delete it.
Ctrl – F1	Append line delete buffer after current line.
F2	Toggle newline mode. If entering newline mode then insert before current line. If leaving and current line is null then delete it.
Ctrl – F2	Insert line delete buffer after current line.
F3	Delete current line and save in line delete buffer.
F4	Fill tagged lines or until the next blank line if no tags are set. If option justify is on, then justify the filled text.
Ctrl – F4	Center tagged lines or current line if no tags are set. If option justify is on, then justify the filled text.
F5	Split current line at cursor. If character at cursor is a space then delete it.
Ctrl – F5	Split current line at cursor. Character at cursor always saved.
F6	Join line at cursor with next line. If cursor line does not end in a space, append one before the join.
Ctrl – F6	Join line at cursor with next line.
F7	Tag/untag line for tag operation. If F7 is entered twice in rapid succession then a limited block tag is set.
Ctrl – F7	Remove all tags or reset last tags. Its a toggle.
F8	Request tag operation.
Ctrl – F8	Same as F8, however the move or kopy is performed before the current line. Not valid for limited block tags.
F9	Re-execute last typed command.
F10	Re-display last typed command.

Option Control

Alt – a	Toggle option Anchor.
Alt – b	Toggle option Blank.
Alt – d	Toggle option Dual.
Alt – f	Toggle option Fill.
Alt – j	Toggle option Justify.
Alt – l	Toggle option Limit.
Alt – m	Toggle option Meta.
Alt – s	Toggle option Save.
Alt – t	Toggle option Tabs.
Alt – w	Toggle option Wrap.

Margin Control

Shift – F1	Set left margin at cursor position.
Shift – F2	Set right margin at cursor position.
Shift – F3	Move left margin left.
Shift – F4	Move left margin right.
Shift – F5	Move right margin left.
Shift – F6	Move right margin right.
Shift – F7	Move cursor to left margin.
Shift – F8	Move cursor to right margin.

Line Flags

Alt – c	Toggle continuation flag.
Alt – o	Toggle overstrike flag.
Alt – p	Toggle paragraph flag.

Special Characters

hex 00	Null character. Used to terminate lines within the editor. This character can not be saved in your text.
hex 09	Tab character. Expanded on output to the screen as enough spaces to move to the next tab stop. Tab stops are set every 4 columns.
hex 0a	Linefeed character. The carriage return is mapped into this character on input.
hex a3	Macro disable character. Disables any macro expansion for the next key typed.
hex fd	Accept a character from the keyboard.
hex fe	Recall character. Recall last typed command to the command line.
hex ff	Command character. Collect all following characters up until the next newline in a hidden buffer, then execute the buffer as a command.
\	Escape character. Will remove special significance of any meta character it precedes in a pattern. In patterns and the translate command a \hh sequence (hh is two hex digits) will reduce to a single character with hex value hh.
@$^^&.*[-	Meta characters. Only special within patterns when option meta is enabled (**m+**).

Editor Commands

All editor commands are entered in the command area and executed by typing either the carriage return or the keypad + key. The latter will keep you in the command area while the former will in general return you to the text area. In the following descriptions variable input of a particular class will appear in a special typeface e.g. *variable* Editor commands are of the form:

[*range*]*C*[*argument*]

where:

range	is a range of lines the command is to operate on.
C	is a single character indicating a command.
argument	is command specific information.

Line Range *range*

<none**>**	(none; range not specified) Refers to the current line only.
line	Refers to this line only.

line1,*line2*	Refers to lines between line1 and line2 inclusive.
line1;*line2*	Same as "," except that current line is updated by semicolon to *line1*.
*	Refers to all lines in the buffer.
#	Refers to any tagged lines.

Line Address *line*

<none>	(none; line not specified) Refers to the current line only.
number	Refers to the absolute line number specified.
.	Refers to the current line.
$	Refers to the last line.
&	Refers to the top line on your screen.
@	Refers to the defined center line.
%	This refers to the current line if you are in the text area and line zero if you are on the command line.
/*pattern*/	This serves as the line address of the next line which contains an instance of the specified pattern. The search for *pattern* will begin at the current cursor position and will continue to the end of the buffer. If no match has been found by that time, the search will wrap around to the beginning of the buffer and continue looking for *pattern* from line one. If no match is found in the entire buffer then an error will be issued.
?*pattern*?	This serves as the line address of the previous line which which contains an instance of the specified pattern. The search for *pattern* will begin at the current cursor position and will go backwards through the buffer wrapping around from the first to last line if necessary. If no match is found an error will be issued as above. In addition, line addresses may be combined with other line addresses using the + and - keys to form expressions. You may limit a line address to lie within the buffer (between 1 and $) by following it with an or-bar (|). For example `.+5|` Refers to the 5th line after the current line or the last line if there are less than five lines after the current line.

Meta Characters Used in Patterns

.	(dot) Matches any character.
^	(caret) Matches the null string at the beginning of the line.
$	Matches the null string at the end of the line.
@(*number***)**	Matches the null string before the *number*th character of a line.
@(.)	Matches the null string before the current cursor position.
@(t)	Matches the null string before the next tab stop.
[*characters***]**	Matches any one of the characters enclosed by the square brackets. (character class)
[^*characters***]**	Matches any one character which is not enclosed by the square brackets. (character class)
*****	Matches the preceding pattern zero or more times. (closure)
&	Only special in the replacement text of a **Substitute** command. It will be expanded into the text matched.
-	Only special within the square brackets of a character class above. The syntax *character1***-***character2* is an abbreviation to mean *all* characters between *character1* and *character2* inclusive. For example **[a-z0-9]**.

File I/O Commands

e *file*	Delete current buffer and read in the specified file if the buffer has not been modified.
ee *file*	Delete current buffer and read in the specified file.
r *file*	Read the specified file after the current line.
*line***r** *file*	Read the specified file after the specified line.
w *file*	Write your buffer to the specified file.
*range***w** *file*	Write the line range specified to the specified file.
wa *file*	Append your buffer to the specified file.
*range***wa** *file*	Append the line range specified to the specified file.
x *file*	Execute the specified file as a file of editor commands. This is typically used for loading custom macro files.

Alphabetical List of All Editor Commands

[*line*]**a**	append
[*line*]**a** *text*	append *text*
[*line*]**ad**	append delete buffer
b*number*	branch always *number* newlines
b*number***t**	branch if TRUE *number* newlines
b*number***f**	branch if FALSE *number* newlines
[*range*]**c**	change
[*range*]**c** *text*	change for *text*
[*range*]**d**	delete
e	edit current file
e *file*	edit *file* if buffer clean
ee *file*	edit *file*
f	query current file
f *file*	set current file
[*range*]**g/***pattern***/***editor commands*	global
[*range*]**g^/***pattern***/***editor commands*	global
h *number*	hold for *number* time units
[*line*]**i**	insert
[*line*]**i** *text*	insert *text*
[*line*]**id**	insert delete buffer
[*line*]**j**	join *line* and *line*+1
[*range*]**k***target_line*	kopy *range* after *target_line*
l *character*	learn character until Break
[*range*]**m***target_line*	move *range* after *target_line*
o*C***{+|-|~|?}**	Set (**+**), unset (**-**), toggle (**~**) or query (**?**) the option signified by character *C*. Valid option characters are:

a	option anchor
b	option blank
c	option command
d	option dual
e	option environment
f	option fill
i	option insert
j	option justify
l	option limit
m	option meta
n	option newline
s	option save
t	option tab
w	option wrap

[*range*]**p** print expanding non-printing ascii characters into \hh sequences.

[*range*]**P** print without expanding non-ascii characters.

q quit the editor if buffer has not been modified

qq quit the editor regardless

[*line*]**r** read the current file after *line*

[*line*]**r** *file* read *file* after *line*

[*range*]**s/***pattern***/***replacement_text***/**

Substitute text; characters other than **/** may also be used as the delimiter

[*range*]**s***number***/***pattern***/***replacement_text***/**

Substitute text, *number*th occurrence only.

Where:

/	is a single character deliminator (Usually a slash, but other characters may be used
number	is the *number*th occurrence

t *character replacement_text*

define translate for *character*

t *character* remove translate for *character*

t ? *character*	query translate for *character*
[*range*]**u***count editor_commands*	until *count*
[*range*]**u***cond editor_commands*	until *cond*
[*range*]**u***CountCond editor_commands*	until *Count* or *Cond*, where *cond* is one of "**t**" (TRUE) or "**f**" (FALSE).
va*number number number*	change color of display areas
vc	center display
vc *number*	set center line
vl*quantity*	set left margin
vr*quantity*	set right margin
quantity	scroll display
vw	don't wait for horizontal retrace when using color card
[*range*]**w**	write to current file
[*range*]**w** *file*	write to *file*
[*range*]**wa**	write append to current file
[*range*]**wa** *file*	write append to *file*
x	execute current file
x *file*	execute *file*
[*range*]**y"***prompt_text***"***command_chars***"***commands*	yut
[*range*]**zcc***character*	zap cursor change
[*range*]**zcd**	zap cursor delete
[*range*]**zce**	zap cursor erase
[*line*]**zcf***character*	zap cursor fill character
[*line*]**zch***location*	zap cursor horizontal

[*line*]**zcl**	zap cursor lock
[*line*]**zcp**	zap cursor purge character delete buffer
[*range*]**zcr**	zap cursor restore from character delete buffer
[*range*]**zcR***number*	zap cursor restore *number* characters
[*range*]**zcs**	zap cursor save in character delete buffer
[*line*]**zk***line*	zap kopy to/from command line
[*range*]**zlc**	zap line center between margins
[*range*]**zld**	zap line delete between limits
[*range*]**zle**	zap line erase between limits
[*range*]**zlf**	zap line fill between margins
[*range*]**zlj**	zap line join
[*range*]**zlo**	zap line overstrike
[*range*]**zlp**	zap line paragraph
zlq	zap line query if tags are set
[*line*]**zlr**	zap line restore delete buffer at cursor
[*range*]**zls**	zap line save text between limits in delete buffer
[*line*]**zlt**	zap line tag
zlu	zap line unset/set all tags
zm	zap message
[*line*]**zp**	zap purge line delete buffer
[*line*]**zq**	zap query

Examples

Remove leading blanks on all lines:

```
*s/^ *//
```

Remove trailing blanks on all lines:

```
*s/ *$//
```

Reverse order of all lines:

```
*m0
```

Duplicate all lines:

```
*k$
```

Duplicate current line:

```
.k.
```

Print first 4 lines after each “function”:

```
g/function/.,.+4p
```

Delete all empty lines:

```
g/^$/d
```

Delete all lines which contain only spaces or are empty:

```
g/^ *$/
```

Delete all lines containing the word comment:

```
g/comment/d
```

Place a “,” after each “however”:

```
g/however/s//&,/
```

Place a line containing the word “end” after each “begin”:

```
g/begin/a end
```

Turn runs of spaces into tabs:

```
s/@(t)/\01/s/ *\01/\09/
```