ne

A nice editor
Version 1.0

by Sebastiano Vigna

1 Introduction

`ne` is a full screen text editor for UN*X (or, more precisely, for POSIX: see Chapter 7 [Motivations and Design], page 47) and for the Amiga. I came to the decision of writing such an editor after getting completely sick of `vi`, both from a feature and user interface point of view. I needed an editor that I could use through a `telnet` connection or a phone line, and that wouldn't fire off a full-blown LITHP[1] operating system just to do some editing.

The first versions of `ne` were created on an Amiga 3000T, using the port of the `curses` library by Simon John Raybould. After switching to the lower-level `terminfo` library, the development continued under UN*X. Finally, I ported `terminfo` to the Amiga, thus making again possible to develop on that platform. For `ne` 1.0, an effort has been made in order to provide a `terminfo` emulation using GNU's `termcap`.

The main inspiration for this work came from Martin Taillefer's `TurboText` for the Amiga, which is the best editor I ever saw, on any computer.

The design goal of `ne` was to write an editor easy to use at first sight, powerful, and completely configurable. Running on any terminal that `vi` could handle was also a basic issue, because there is no use in getting accustomed to a new tool if you cannot use it when you *really* need it. Finally, sparing resource usage was considered essential. `ne` currently runs only on UN*X terminals, but a future release will provide an X interface.

Of course, the Amiga user will find `ne` much less attractive. There are several excellent editors for the Amiga, and `ne` lacks many powerful features the users are now accustomed to. However, for very special usages, such as editing through a serial terminal connected to the `AUX:` device, `ne` is the only choice, since it runs in any CLI (even in remote ones). Of course, a correct installation of `aterminfo` (the Amiga `terminfo` clone) is a basic requirement. See Chapter 8 [Some Notes for the Amiga User], page 49.

A concise overview of the main features follows:

- three user interfaces: control keystrokes, command line, and menus; keystrokes and menus are completely configurable;
- the number of documents and clips, the dimensions of the display, and the file/line lengths are limited only by the integer size of the machine;
- simple scripting language where scripts can be generated *via* an idiotproof learn/play method;
- unlimited undo/redo capability (can be disabled with a command);
- automatic preferences system based on the extension of the file name being edited;
- a file requester for easy file retrieval;
- extended regular expression search and replace a la `emacs` and `vi`;
- editing of binary files.

[1] This otherwise unremarkable language is distinguished by the absence of an '`s`' in its character set; users must substitute '`th`'. LITHP is said to be useful in prothethhthing liththth.

2 Basics

Simple things should be simple, and complex things should be possible. (Alan Kay)

`ne`'s user interface is essentially a compromise between the limits of character driven terminals, and the power of GUIs. While I would *never* deny that *real* editing is done without ever touching a mouse, it is also true that it should be doable without ever touching a manual. This two conflicting goals can be easily accomodated in a single program if we can offer a series of interfaces which allow for a differentiated usage.

In other words, it is unlikely that a `ne` wizard will ever have to activate a menu, but in order to become an expert user you just have to use the menus enough to learn by heart the most important keystrokes. A good manual is always invaluable when one comes to configuration and esotheric features, but not all users will ever need to change `ne`'s menus or key bindings.

Another important thing is that powerful features should always be accessible, at least in part, to every user. Putting a macro capability that depends on learning LITHP is undoubtly a strange design choice. The average user should be able to record his actions, replay them, and save them in a humanly readable format for further usage and editing.

In the following sections we will make a quick tour of `ne`'s features.

2.1 Starting

In order to start `ne`, just type '`ne`' and press `Return`. If you want to edit some specific file(s), you can put their name(s) on the command line just after the command name, as for any UN*X command. Immediately (hopefully), the screen of your terminal will be cleared (or filled with the contents of the first file you specified).

At the bottom of the screen, you will see a line containing some numbers and letters. It is named the *status bar*, because it reports to you part of the internal state of the editor.

Writing text is pretty straightforward: if you terminal is properly configured, every key will (should) do what you expect. Alphabetic characters insert text, cursor keys move the cursor, and so on. You can use the `Delete` and `Backspace` key in order to perform corrections. If your keyboard has an `Insert` key, you can use it in order to toggle insert mode. In general, `ne` tries to squeeze everything from your keyboard—functions keys and special movement keys should work flawlessly if your terminal is properly configured. If not, complain with your system administrator.

At startup, the status bar has the following form:

```
L:     1  C:     1  i      pvu t
```

(the numbers could be different, and a filename could be shown as last item). You probably already guessed that the numbers after '`L:`' and '`C:`' are the line and column numbers, respectively. The small letters represent instead user flags that you can turn on and off. In particular, the '`i`' letter tells you that insert mode is on, while '`p`' indicates that the automatic preferences system is activated. For a thorough explanation of the meaning of the flags on the status line, see Section 3.2 [The Status Bar], page 9.

Once you are accustomed to cursor movement and line editing, it is time to press the `f1` (the first function key), or in case your keyboard does not have such a key, `Escape`. Immediately, the *menu bar* will appear, and the first menu will be drawn. You can now move around menus and menu items by pressing the cursor keys. Moreover, a lower case alphabetic key will move to the first item whose name starts with that letter, and an upper case alphabetic key will move to the first menu whose name starts with that letter.

Moving around the menus should give you an idea of the capabilities of `ne`. If you want to save your work, you should use the '`Save As...`' item from the '`Project`' menu. Menus are fully discussed in Section 3.6 [Menus], page 12. When you want to exit from the menu system, press

f1 (or Escape) again. If you instead prefer to choose a command and execute it, move over the respective menu item and press Return.

At the end of several menu items you will find strange symbols like ^A or f1. They represent *shortcuts* for the respective menu items. In other words, instead of activating, selecting and executing a menu item, which can take seconds, you can simply press a couple of keys. The symbol '^' in front of a character denotes the shortcut produced by the Control key plus that character (I suppose here you are perfectly aware of the usage of the Control key: it is just as if you had to type a capital letter with Shift). The descriptions of the form f*n* represent instead function keys. Note that under certain conditions (for instance, while using ne through a telnet connection) some of the shortcuts could not work, because they are trapped by the operating system for other purposes (see Chapter 6 [Hints and Tricks], page 45).

Finally, we have the third and last interface to ne's features: the *command line*. If you press *Control-K*, or Escape followed by ':' (a la vi), you will be requested some command to execute. Just press Return for the time being.

In what follows, when explaining how to use a command, we will usually describe the corresponding menu item. The related shortcut and command can be found, respectively, on the menu item itself, and in Section 3.6 [Menus], page 12.

2.2 Loading and Saving

The first thing to learn about an editor is how to exit. ne has a CloseDoc command which can be activated by pressing *Control-Q*, by choosing the 'Close' item of the 'Document' menu, or by activating the command line with *Control-K*, writing 'cd' and pressing Return. Its effect is to close the current document without saving any modification (you will be requested to confirm your choice in case the current document has been modified since the last save).

There is also a command Quit, which leaves the editor without saving any modification, and an Exit command which saves the modified documents before quitting.

This choice of shortcuts could surprise you—wouldn't Quit be a much better candidate for *Control-Q*? Well, experience shows that the most common operation is closing a document, rather than quitting the editor. If there is just one document, the two operations coincide (this is typical, for instance, when you use ne for writing electronic mail), and if there are many documents, it is far more common to close a single document than all the existing documents.

If you want to load a file, you have to use the Open command, which can be activated by pressing *Control-O*, by choosing the 'Open...' item of the 'Project' menu, or by typing it on the command line (as in the previous case). You will be prompted with the list of files and directories in the current working directory (you can tell the directory names because they end with a slash). You can move on any of them by using the cursor keys, or any other movement key. Pressing an alphabetic key will move the cursor on the first entry after the cursor starting with the given letter. When the cursor is positioned over the file you want to open, just press Return, and the file will be opened. If instead you move on a directory name, pressing Return will display the contents of that directory.

You can also escape with f1 or Escape, and type manually the file name on the command line (or escape again, and abort the loading operation).

When you want to save a file, just use the command Save (*Control-S*). It will use the current document name (and will ask for one if none is available). SaveAs, on the contrary, will always ask for a new name before saving the file.

If ne is interrupted by an external signal (for instance, if your terminal crashes), it will try to save your work on some emergency files. See Section 3.9 [Emergency Save], page 19.

2.3 Editing

An editor is presumably used for editing. If not, you could decide to not use `ne`, because it just does that—it edits. It does not play `Tetris`. It does not evaluate recursive functions. It does not solve your love problems. It just allows you to edit.

The design of `ne` makes editing extremely natural and straightforward. There is no special thing to do in order to start—I assume that if you start an editor, you want to edit, and *not* to give commands. Thus, just press the keys of your keyboard, and see what happens.

`ne` provides two ways of deleting characters, bound to the `Backspace` (or *`Control-H`*, if you have no such key) and to the `Delete` key respectively. In the former case you delete the character at the left of the cursor, while in the latter you delete the character just under the cursor. This is in contrast with many UN*X editors, which for unknown reasons decide to limit your ways of destroying things—something notoriously much funnier than creating. (see also Section 4.11.2 [DeleteChar], page 39, and see Section 4.11.3 [Backspace], page 39).

If you want to delete a line, you can use the `DeleteLine` command, or *`Control-Y`*. A very nice feature of `ne` is that each time a nonempty line is deleted, it is stored in a temporary buffer from which it can be undeleted via the `UndelLine` command, or *`Control-U`* (see also Section 4.11.5 [DeleteLine], page 40, and see Section 4.7.3 [UndelLine], page 30).

If you want to copy, cut, paste or erase a block of text, you have to set a mark. This is done via the `Mark` command, activated by choosing the '`Mark Block`' item of the '`Edit`' menu, or by pressing *`Control-B`* (=block). This command toggles the mark (puts it in the current cursors position, or remove it). Whenever the mark is set, the zone between the mark and the cursor can be cut, copied or erased. Note that by using *`Control-@`* you can set a *vertical* mark instead, that allows to cut exactly rectangles of text. Whenever a mark has been set, an '`M`' appears on the command line; a '`V`' appears instead if the mark is vertical. If you forget where the mark is currently, you can use the '`Goto Mark`' menu item of the '`Search`' menu in order to move to cursor over it.

When you cut or copy a block, you can save it with the '`Save Clip...`' menu item of the '`Edit`' menu. You can also load in memory a file with '`Open Clip...`', and paste it anywhere. All such operation act on the *current clip*, which is by default the clip 0. You can change the current clip number with the `ClipNumber` command. See Section 4.4.10 [ClipNumber], page 25.

One of the most noteworthy features of `ne` is its *unlimited undo/redo* capability. Each editing action is recorded, and can be played back and forth as much as you like. Undo and redo are bound to the function keys `f5` and `f6`.

Another interesting feature is the possibility of loading an unlimited number of documents. If you activate the `NewDoc` command (using the '`Document`' menu, *`Control-D`* or the command line), a new, empty document will be created. You can switch between the existing documents in memory with `f2` and `f3`, which are bound to the `PrevDoc` and `NextDoc` commands. If you have a lot of documents, the '`Select...`' menu item prompts you with the list of names of currently loaded documents, and allows you to choose directly what to edit.

2.4 Basic Preferences

`ne` has a number of *flags* which specify alternative behaviours, the most prototypical example being the *insert* flag, which specifies if the text you type in is inserted in the existing text, or overwrites it. You can toggle this flag with the '`Insert`' menu item of the '`Prefs`' menu, or with the `Insert` key of your keyboard (*toggle* means to change the value of a flag from true to false, or from false to true; also see Section 4.9.3 [Insert], page 33).

Another important flag is the *free form* flag, which specifies if the cursor can be moved anywhere, or only on existing text (a la `vi`). Programmers usually prefer non free form editing;

text writers prefer free form. See Section 4.9.5 [FreeForm], page 33, for some elaboration. The free form flag can be set with the '`Free Form`' menu item of the '`Prefs`' menu

At this point, I suggest you to explore by trial and error the other flags of the '`Prefs`' menu. I would prefer spending a couple of words about the *automatic preferences*, or, in short, *autoprefs*.

Having many flags ensures a high degree of flexibility, but can turn editing into a nightmare if for each different kind of file loaded one has to turn on and off dozens of options. The solution is having the program handling all the details, depending on some characteristic of the file.

The solution chosen in `ne` is to look at the *extension* of the name of a file, i.e., the last group of letters after a dot. For instance, the extension of `ne.texinfo` is '`texinfo`', while the extension of `source.c` is '`c`'.

Whenever you select the '`Save AutoPrefs`' menu item, `ne` saves in a directory named `.ne` (in your home directory) a file, with the same name as the extension of the name of current document (postfixed with '`#ap`'), containing all the commands which will rebuild the current settings. Whenever you will open a file with the same extension in its name, `ne` will reload automagically the same set of preferences (there is a flag which inhibits the process; see Section 4.9.1 [AutoPrefs], page 32).

Finally, when you select the '`Save Def Prefs`' menu item, a special preferences file named `.default#ap` is saved that is loaded whenever `ne` is run, before loading any file. Here is the place to put in the preferences you always want to be set.

Note also that a preferences file is just a macro (as described in the following section). Thus, it can be edited manually if necessary.

2.5 Basic Macros

Very often, the programmer or the text writer has to repeat some complex editing action over a series of similar blocks of text. Unless you are an `awk` wizard, this is where *macros* come in.

Macros allows you to record complex actions and play them many times. They can be saved on disk for further usage, edited, loaded, or bound to any key. This allows to reconfigure each key of your keyboard with a complex action, if you want so.

Recording a macro is very simple. The keystroke *`Control-T`* starts and stop a macro recording (you can see you are actually recording if an '`R`' appears on the status bar). Whatever you do is registered, and when you stop (again with *`Control-T`*) the recording process, you can play the macro with the '`Play Once`' item of the '`Macros`' menu, or with the `f9` key. If you want to repeat the action many times, the `Play` command allows you to specify a number of times to repeat the macro. You can always interrupt the execution with *`Control-\`*.

After recording a macro, you can save it with the '`Save Macro...`' menu item. Any macro file can be played (without modifying the current macro) with the '`Macro...`' menu item. Useful macros can be permanently bound to a keystroke, as explained in Section 5.1 [Key Bindings], page 43. Moreover, whenever a command line does not specify an "internal" command, it is assumed to specify the name of a macro to execute. Thus, you can execute macros just by typing their complete file name.

In order to make this mechanism even more transparent, if the first attempt to open a macro fails `ne` checks for a macro with the given name in the `.ne` subdirectory of your home directory. This allows you to program simple extensions to `ne`'s language. For instance, all automatic preferences macros can be executed just by typing their names—if you have an automatic preference for the '`doc`' extension, by typing the command `doc#ap` you will set `ne`'s flags exactly as if you loaded a file ending with '`.doc`'. In general, it is a good idea to save frequently used macros in `$HOME/.ne`, so that you can invoke them just by name (of course, you cannot recall in this way macros with the same name as a command).

Since loading a macro each time it is invoked would be a rather slow and expensive process, once a macro has been executed it is cached internally. The next invocations of the macro will used the cached version.

Warning: the macro names are *not* case sensitive or path sensitive. `ne` caches internally only the file name of a macro, not the path name, and uses a case insensitive comparison. That is, if you call '`~/foobar/macro`', a subsequent call for '`/usr/MACRO`' will use the cached version. You can clear the cache by using the `UnloadMacros` command. See Section 4.6.6 [UnloadMacros], page 30.

2.6 More Advanced Features

It often happens that you have to browse through a file, switching frequently between a small number of positions. In this case, you can use *bookmarks*. There are ten bookmarks per document: they can be set with the `SetBookmark` command, and reached with the `GotoBookmark` command. See Section 4.10.19 [SetBookmark], page 39, and see Section 4.10.20 [GotoBookmark], page 39. Note that in the default configuration no key binding is assigned to these commands: if you use them frequently, you may want to change the key bindings. See Section 5.1 [Key Bindings], page 43.

`ne` allows a simplified form of *binary editing*. If the binary flag is set, only NULLs are considered newlines when loading or saving. Thus, binary files can be safely loaded, modified and saved. Inserting a new line or joining two lines has obviously the effect of inserting or deleting a NULL. Please be careful to not mismatch the state of the binary flag when loading and saving the same file.

The `NoFileReq` command allows to deactivate the file requester. It is intended for "tough guys" who always remember the name of their files and can type them at the speed of light.

Should you need to execute UN*X commands while using `ne`, you have two possibilities. The `System` command can run any UN*X command; you will get back into `ne` as soon as the command execution terminates. See Section 4.12.8 [System], page 41. The `Through` command (which can be found in the '`Edit`' menu), however, is much more powerful: it cuts the current block, passes it as standard input to any UN*X command, and pastes the command output at the current cursor position. The neat effect with filter commands (UN*X commands which read from standard input and write to standard output, e.g., `sort`) is that the currently selected block is passed through the filter. See Section 4.4.11 [Through], page 25.

For an exaustive list of the remaining features of `ne`, see Chapter 3 [Reference], page 9.

3 Reference

In this chapter we will methodically overview each part of `ne`. It is a required reading for becoming an expert user, because some commands and features are not available through menus.

3.1 Arguments

The main argument you can give to `ne` is a list of files you want to edit. They will be loaded within separate documents.

The `--noconfig` option allows to skip the reading of the key bindings and menu configuration files (see Chapter 5 [Configuration], page 43). This is essential if you are experimenting a new configuration, and you make mistakes in it.

The `--macro` *`filename`* option allows to specify the name of a macro that will be started just after all documents have been loaded. A typical macro would move the cursor on a certain line.

3.2 The Status Bar

The last line of the screen, the *status bar*, is reserved by `ne` in order to display some information about its internal state. Note that on most terminals is physically impossible to write a character on the last column of this line, so that we are not really stealing precious space to editing.

The status bar looks more or less like that:

```
L:   31  C:   25  iabcwfpvurBMR* /foo/bar
```

Some of the letters may be missing—their presence is related to the value of a series of flags, as we will see later.

The numbers after '`L:`' and '`C:`' represent the line and column of the cursor position. The first line and the first column are numbered by 1. They change while the cursor is moving, and this fact can really slow down the cursor movement if you are using `ne` through a slow connection. In this case, it is a good idea to turn off the status bar using the '`Status Bar`' menu item of hte '`Prefs`' menu, or the `StatusBar` command. See Section 4.9.7 [StatusBar], page 34. Note that if you really need it, it is anyway a good idea to turn on the fast GUI mode using the '`Fast GUI`' menu item of the '`Prefs`' menu, or the `FastGUI` command (see Section 4.9.4 [FastGUI], page 33), because in this case the status bar is not draw in reverse, and some additional optimization can be done when refreshing it.

The letters after the line and column number represent the status of the flags associated to a series of command. In detail:

'`i`' appears if the insert flag is true. See Section 4.9.3 [Insert], page 33.

'`a`' appears if the auto indent flag is true. See Section 4.8.8 [AutoIndent], page 32.

'`b`' appears if the back search flag is true. See Section 4.5.8 [SearchBack], page 28.

'`c`' appears if the case sensitive search flag is true. See Section 4.5.9 [CaseSearch], page 28.

'`w`' appears if the word wrap flag is true. See Section 4.8.7 [WordWrap], page 32.

'`f`' appears if the free form flag is true. See Section 4.9.5 [FreeForm], page 33.

'`p`' appears if the automatic preferences flag is true. See Section 4.9.1 [AutoPrefs], page 32.

'`v`' appears if the verbose macros flag is true. See Section 4.9.12 [VerboseMacros], page 35.

'`u`' appears if the undo flag is true. See Section 4.7.4 [DoUndo], page 30.

'r' appears if the read only flag is true. See Section 4.9.8 [ReadOnly], page 34.

't' appears if the turbo parameter is nonzero. See Section 4.9.11 [Turbo], page 35.

'B' appears if the binary flag is true. See Section 4.9.2 [Binary], page 33.

'M' appears if you are currently marking a block. See Section 4.4.1 [Mark], page 24.

'V' can appear in place of 'M' if you are currently marking a vertical block. See Section 4.4.2 [MarkVert], page 24.

'R' appears if you are currently recording a macro. See Section 4.6.1 [Record], page 28.

'*' appears if the document has been modified since the last save.

The file name appearing after this group of letters is the file name of the current document. Very long file names may end off screen. Of course, ne is keeping track internally of the complete file name, which is used by the Save command, and as default input by the SaveAs command. See Section 4.2.4 [Save], page 22, and Section 4.2.5 [SaveAs], page 22.

Note that sometimes ne needs to communicate you some message. The message is then usually written over the status bar, where it stays until you do something (any such message ends with a full stop). Any action (moving the cursor, inserting a character *et cetera*) will restore the normal look of the status bar.

3.3 The Input Line

The last line of the screen is usually occupied by the status bar (see Section 3.2 [The Status Bar], page 9); however, whenever you have to interact with ne, for instance providing some input to a command which requires it, the last line becomes the *input line*. You can see this because a *prompt* is displayed at the start of the line, suggesting what kind of input is required. The prompt always ends with a colon, so it is impossible to miss it for an error message.

There are two essentially different ways in which input can be done: whenever ne just needs to know a simple decision which can be expressed by one character, you can type it and ne will immediately accept and use your input: it is called a *immediate* input. This is the case, for instance, of the prompt which asks you if you really want to quit a modified document. If, instead, a whole string is required, you can type several characters, perform some editing actions, and end your input with the Return key: it is called a *long input*. You can easily distinguish between this two modes because in immediate mode the cursor is not on the input line.

When doing a immediate input, there is usually a character appearing just after the prompt. It is the default value, which is used if you just press the Return key. Note that case is not significant in immediate inputs. Moreover, if a yes/no choice is requested, *anything* else than 'y' will be considered a negative answer.

When doing a long input, there are a number of editing features available. As anywhere else in ne, *knowledge reuse* is the basic goal.

Essentially, you can edit the input line *exactly as a line of text*. All key bindings related to line editing work on the command line exactly as in a document. This is true even of custom key bindings. Thus, no particular explanation is needed here—just edit as you are used to. Moreover, the contents of the input line can be replaced by the first line of the current clip using the keystroke which is bound to the Paste command, usually *Control-V*.

Note that if you type a line which is longer then the screen width, it is scrolled away. This allows to input very long lines even on small monitors. There is a limit of 1024 characters, but I do not really think you are going to feel it as a limitation.

There is a useful feature of long inputs: you are often offered with a default input value (for instance, if you change the TAB size, the old TAB width). If you type immediately an alphabetic

character, the default value is completely erased. If instead you use any non alphabetical character (for instance, you move the cursor or delete the first character), the default input can be further edited. A simple way of keeping the default value without really doing anything is to press the `Backspace` key (or any key which is bound to the `Backspace` command). No character can be deleted, but being the first key non alphabetical, the default input value will be retained.

You can always cancel a long input using `f1`, `Escape`, or in general any key which is bound to the `Escape` command. The effect will vary depending on what your were requested to input, but in general the execution of the command requiring the input is stopped.

3.4 The Command Line

The command line is a typical (topical) way of controlling an editor on character driven systems. It has some advantages in term of access speed, but it is a complete failure from a user interface point of view. `ne` has a command line which should be used whenever strange features have to be accessed, or whenever you are enough accustomed to know by heart the commands you want to use, and which are not bound to any key.

In order to access the command line, you have two possibilities: either activating menu mode and typing a colon (‘`:`’), or typing *`Control-K`* (or any key which is bound to the `Exec` command; see Section 4.12.3 [Exec], page 40). The first method will work regardless of any key binding configuration if you activate the menus with the `Escape` key, since it is not possible to reconfigure it.

Once you activate the command line, the status bar will turn into an input line (see Section 3.3 [The Input Line], page 10) with prompt ‘`Command:`’, waiting for you doing a long input. In other words, you can now type any command (possibly with arguments), and when you press `Return` the command will be executed.

If the command you specify does not appear in `ne`’s internal tables, it is considered to be a macro name. See Section 2.5 [Basic Macros], page 6, for details.

3.5 The Requester

In various situations, `ne` needs to ask you to choose one of several strings (where “several” can mean a lot, even hundreds). For this kind of event, the *requester* is issued. The requester displays the strings in as many columns as possible, and let you move with the cursor from one string to another. The strings can fill many screens, which are handled as consecutive pages. Always in the spirit of knowledge reuse, all the navigation keys work exactly as in normal editing. This is true even of custom key bindings. Thus, for instance, you can move to the top or bottom of the list with *`Control-^`* (in the standard keyboard configuration).

As with the input line (see Section 3.3 [The Input Line], page 10), you can confirm your input with `Return` or escape the requester with `f1` (or the `Escape` key, or whatever has been bound to the `Escape` command).

A special feature is bound to alphabetic characters: they move you on the next entry starting with the letter you typed. The search is case insensitive, and continues on the first string after having passed the last one.

An example of requester is the file requester that `ne` issues whenever a file operation is going to take place. In this case, pressing `Return` while on a directory name will enter the directory. Note also that, should the requester take too long to appear, you can interrupt the directory scanning with *`Control-\`*. However, the listing will likely be incomplete.

Note that there are two items which always appear in the file listing: `./` and `../`. The first one represents the current directory (and can be used to force a reread), the second one the parent directory (and can be used to move up by one level).

Another example of requester is the list of commands appearing when you use the `Help` command. (Note that even the help text appearing on the screen is handled by the requester—your "choice" of a line in the text is of course discarded, but the flexibility of the requester allows to gain in code size, since no separate code is necessary in order to display the on-line help.)

3.6 Menus

`ne`'s menus are extremely straightforward. The suggested way of learning their use is by trial and error, with a peek here and there at this manual when some doubts arise.

Menus are activated by `f1`, or in case your keyboard does not have such a key, `Escape`, or in general any key which is bound to the `Escape` command. Movement is accomplished by pressing the cursor keys and the page up/down keys (which move to the first or last menu item in a menu). You can also move around menus and menu items by pressing the alphabetic keys; a lower case letter will move to the first item whose name starts with the given letter; an upper case letter will move to the first menu whose name starts with the given letter.

Each menu item of `ne`'s standard menu correspond exactly to a single command. Thus, in explaining what each menu item allows you to do, you will be simply referred to the section which explains the command relative to the menu item.

If you plan to change `ne`'s menu (see Section 5.2 [Changing Menus], page 43), you should take a look at the file `default.menus` which comes with `ne`'s distribution. It contains a complete menu configuration which clones the standard one.

3.6.1 Project

The Project menu contains standard items which allow to load and save files. Quitting or exiting (with save) `ne` is also possible.

'`Open...`' See Section 4.2.2 [Open], page 22.

'`Open New...`'
See Section 4.2.3 [OpenNew], page 22.

'`Save`' See Section 4.2.4 [Save], page 22.

'`Save As...`'
See Section 4.2.5 [SaveAs], page 22.

'`Clear`' See Section 4.2.1 [Clear], page 22.

'`Quit`' See Section 4.3.1 [Quit], page 23.

'`Exit`' See Section 4.3.2 [Exit], page 23.

'`About`' See Section 4.12.1 [About], page 40.

3.6.2 Documents

The Documents menu contains commands which create new documents, destroy them, and browse through them.

'`New`' See Section 4.3.3 [NewDoc], page 23.

'`Close`' See Section 4.3.4 [CloseDoc], page 23.

'`Next`' See Section 4.3.5 [NextDoc], page 23.

'`Prev`' See Section 4.3.6 [PrevDoc], page 23.

'`Select...`'
See Section 4.3.7 [SelectDoc], page 23.

3.6.3 Edit

The Edit menu contains the commands related to cutting and pasting text.

'`Mark Block`'
: See Section 4.4.1 [Mark], page 24.

'`Cut`'
: See Section 4.4.4 [Cut], page 24.

'`Copy`'
: See Section 4.4.3 [Copy], page 24.

'`Paste`'
: See Section 4.4.5 [Paste], page 24.

'`Erase`'
: See Section 4.4.7 [Erase], page 25.

'`Through`'
: See Section 4.4.11 [Through], page 25.

'`Delete Line`'
: See Section 4.11.5 [DeleteLine], page 40.

'`Mark Vert`'
: See Section 4.4.2 [MarkVert], page 24.

'`Paste Vert`'
: See Section 4.4.6 [PasteVert], page 25.

'`Open Clip...`'
: See Section 4.4.8 [OpenClip], page 25.

'`Save Clip...`'
: See Section 4.4.9 [SaveClip], page 25.

3.6.4 Search

'`Find...`'
: See Section 4.5.1 [Find], page 26.

'`Find RegExp...`'
: See Section 4.5.2 [FindRegExp], page 26.

'`Replace...`'
: See Section 4.5.3 [Replace], page 26.

'`Replace Once...`'
: See Section 4.5.4 [ReplaceOnce], page 27.

'`Replace All...`'
: See Section 4.5.5 [ReplaceAll], page 27.

'`Repeat Last`'
: See Section 4.5.6 [RepeatLast], page 27.

'`Goto Line...`'
: See Section 4.10.5 [GotoLine], page 37.

'`Goto Col...`'
: See Section 4.10.6 [GotoColumn], page 37.

'`Goto Mark...`'
: See Section 4.10.7 [GotoMark], page 37.

'`Match Bracket`'
: See Section 4.5.7 [MatchBracket], page 27.

3.6.5 Macros

'Record' See Section 4.6.1 [Record], page 28.

'Stop' See Section 4.6.1 [Record], page 28.

'Replace...'
See Section 4.5.3 [Replace], page 26.

'Play Once'
'Play Many...'
See Section 4.6.2 [Play], page 28.

'Play Macro...'
See Section 4.6.3 [Macro], page 29.

'Open Macro...'
See Section 4.6.4 [OpenMacro], page 29.

'Save Macro...'
See Section 4.6.5 [SaveMacro], page 29.

3.6.6 Extras

This menu contains a couple of special items.

'Exec...' See Section 4.12.3 [Exec], page 40.

'Help...' See Section 4.12.5 [Help], page 41.

'Refresh' See Section 4.12.7 [Refresh], page 41.

'Undo' See Section 4.7.1 [Undo], page 30.

'Redo' See Section 4.7.2 [Redo], page 30.

'Undel Line'
See Section 4.7.3 [UndelLine], page 30.

'Center' See Section 4.8.1 [Center], page 31.

'Paragraph'
See Section 4.8.2 [Paragraph], page 31.

'ToUpper' See Section 4.8.3 [ToUpper], page 31.

'ToLower' See Section 4.8.4 [ToLower], page 31.

'Capitalize'
See Section 4.8.5 [Capitalize], page 31.

3.6.7 Navigation

'Move Left'
See Section 4.10.1 [MoveLeft], page 36.

'Move Right'
See Section 4.10.2 [MoveRight], page 36.

'Line Up' See Section 4.10.3 [LineUp], page 37.

'Line Down'
See Section 4.10.4 [LineDown], page 37.

'Prev Page'
See Section 4.10.8 [PrevPage], page 37.

‘`NextPage`’
See Section 4.10.9 [NextPage], page 37.

‘`Top/Bottom`’
See Section 4.10.17 [ToggleSEOF], page 38.

‘`Beg Of Line`’
See Section 4.10.13 [MoveSOL], page 38.

‘`End Of Line`’
See Section 4.10.12 [MoveEOL], page 38.

‘`Prev Word`’
See Section 4.10.10 [PrevWord], page 38.

‘`Next Word`’
See Section 4.10.11 [NextWord], page 38.

3.6.8 Prefs

‘`Tab Size...`’
See Section 4.9.10 [TabSize], page 35.

‘`Insert/Over`’
See Section 4.9.3 [Insert], page 33.

‘`Free Form`’
See Section 4.9.5 [FreeForm], page 33.

‘`Status Bar`’
See Section 4.9.7 [StatusBar], page 34.

‘`Fast GUI`’ See Section 4.9.4 [FastGUI], page 33.

‘`Word Wrap`’
See Section 4.8.7 [WordWrap], page 32.

‘`Right Margin`’
See Section 4.8.6 [RightMargin], page 31.

‘`Auto Indent`’
See Section 4.8.8 [AutoIndent], page 32.

‘`Load Prefs...`’
See Section 4.9.13 [LoadPrefs], page 35.

‘`Save Prefs...`’
See Section 4.9.14 [SavePrefs], page 36.

‘`Load AutoPrefs`’
See Section 4.9.15 [LoadAutoPrefs], page 36.

‘`Save AutoPrefs`’
See Section 4.9.16 [SaveAutoPrefs], page 36.

‘`Save Def Prefs`’
See Section 4.9.17 [SaveDefPrefs], page 36.

3.7 Regular Expressions

Regular expressions are a powerful way of specifying complex search and replace operations.

3.7.1 Syntax

The following section is taken (with minor modifications) from the GNU regular expression library documentation, and is Copyright © Free Software Foundation.

A regular expression describes a set of strings. The simplest case is one that describes a particular string; for example, the string '`foo`' when regarded as a regular expression matches '`foo`' and nothing else. Nontrivial regular expressions use certain special constructs so that they can match more than one string. For example, the regular expression '`foo|bar`' matches either the string '`foo`' or the string '`bar`'; the regular expression '`c[ad]*r`' matches any of the strings '`cr`', '`car`', '`cdr`', '`caar`', '`cadddar`' and all other such strings with any number of '`a`''s and '`d`''s.

Regular expressions have a syntax in which a few characters are special constructs and the rest are *ordinary*. An ordinary character is a simple regular expression which matches that character and nothing else. The special characters are '`$`', '`^`', '`.`', '`*`', '`+`', '`?`', '`[`', '`]`' , '`(`', '`)`' and '`\`'. Any other character appearing in a regular expression is ordinary, unless a '`\`' precedes it.

For example, '`f`' is not a special character, so it is ordinary, and therefore '`f`' is a regular expression that matches the string '`f`' and no other string. (It does *not* match the string '`ff`'.) Likewise, '`o`' is a regular expression that matches only '`o`'.

Any two regular expressions *a* and *b* can be concatenated. The result is a regular expression which matches a string if *a* matches some amount of the beginning of that string and *b* matches the rest of the string.

As a simple example, we can concatenate the regular expressions '`f`' and '`o`' to get the regular expression '`fo`', which matches only the string '`fo`'. Still trivial.

Note: special characters are treated as ordinary ones if they are in contexts where their special meanings make no sense. For example, '`*foo`' treats '`*`' as ordinary since there is no preceding expression on which the '`*`' can act. It is poor practice to depend on this behavior; better to quote the special character anyway, regardless of where is appears.

The following are the characters and character sequences which have special meaning within regular expressions. Any character not mentioned here is not special; it stands for exactly itself for the purposes of searching and matching.

'`.`' is a special character that matches anything except a newline. Using concatenation, we can make regular expressions like '`a.b`' which matches any three-character string which begins with '`a`' and ends with '`b`'.

'`*`' is not a construct by itself; it is a suffix, which means the preceding regular expression is to be repeated as many times as possible. In '`fo*`', the '`*`' applies to the '`o`', so '`fo*`' matches '`f`' followed by any number of '`o`''s.

The case of zero '`o`''s is allowed: '`fo*`' does match '`f`'.

'`*`' always applies to the *smallest* possible preceding expression. Thus, '`fo*`' has a repeating '`o`', not a repeating '`fo`'.

'`+`' '`+`' is like '`*`' except that at least one match for the preceding pattern is required for '`+`'. Thus, '`c[ad]+r`' does not match '`cr`' but does match anything else that '`c[ad]*r`' would match.

'`?`' '`?`' is like '`*`' except that it allows either zero or one match for the preceding pattern. Thus, '`c[ad]?r`' matches '`cr`' or '`car`' or '`cdr`', and nothing else.

‘`[ ... ]`’ ‘`[`’ begins a *character set*, which is terminated by a ‘`]`’. In the simplest case, the characters between the two form the set. Thus, ‘`[ad]`’ matches either ‘`a`’ or ‘`d`’, and ‘`[ad]*`’ matches any string of ‘`a`’s and ‘`d`’s (including the empty string), from which it follows that ‘`c[ad]*r`’ matches ‘`car`’, *et cetera*.

Character ranges can also be included in a character set, by writing two characters with a ‘`-`’ between them. Thus, ‘`[a-z]`’ matches any lower-case letter. Ranges may be intermixed freely with individual characters, as in ‘`[a-z$%.]`’, which matches any lower case letter or ‘`$`’, ‘`%`’ or period.

Note that the usual special characters are not special any more inside a character set. A completely different set of special characters exists inside character sets: ‘`]`’, ‘`-`’ and ‘`^`’.

To include a ‘`]`’ in a character set, you must make it the first character. For example, ‘`[]a]`’ matches ‘`]`’ or ‘`a`’. To include a ‘`-`’, you must use it in a context where it cannot possibly indicate a range: that is, as the first character, or immediately after a range.

‘`[^ ... ]`’ ‘`[^`’ begins a *complement character set*, which matches any character except the ones specified. Thus, ‘`[^a-z0-9A-Z]`’ matches all characters *except* letters and digits.

‘`^`’ is not special in a character set unless it is the first character. The character following the ‘`^`’ is treated as if it were first (it may be a ‘`-`’ or a ‘`]`’).

‘`^`’ is a special character that matches the empty string – but only if at the beginning of a line in the text being matched. Otherwise it fails to match anything. Thus, ‘`^foo`’ matches a ‘`foo`’ which occurs at the beginning of a line.

‘`$`’ is similar to ‘`^`’ but matches only at the end of a line. Thus, ‘`xx*$`’ matches a string of one or more ‘`x`’s at the end of a line.

‘`\`’ has two functions: it quotes the above special characters (including ‘`\`’), and it introduces additional special constructs.

Because ‘`\`’ quotes special characters, ‘`\$`’ is a regular expression which matches only ‘`$`’, and ‘`\[`’ is a regular expression which matches only ‘`[`’, and so on.

For the most part, ‘`\`’ followed by any character matches only that character. However, there are several exceptions: characters which, when preceded by ‘`\`’, are special constructs. Such characters are always ordinary when encountered on their own.

‘`|`’ specifies an alternative. Two regular expressions *a* and *b* with ‘`|`’ in between form an expression that matches anything that either *a* or *b* will match.

Thus, ‘`foo|bar`’ matches either ‘`foo`’ or ‘`bar`’ but no other string.

‘`|`’ applies to the largest possible surrounding expressions. Only a surrounding ‘`( ... )`’ grouping can limit the grouping power of ‘`|`’.

‘`( ... )`’ is a grouping construct that serves three purposes:

1. To enclose a set of ‘`|`’ alternatives for other operations. Thus, ‘`(foo|bar)x`’ matches either ‘`foox`’ or ‘`barx`’.
2. To enclose a complicated expression for the postfix ‘`*`’ to operate on. Thus, ‘`ba(na)*`’ matches ‘`bananana`’, etc., with any (zero or more) number of ‘`na`’s.
3. To mark a matched substring for future reference.

This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature which happens to be assigned as a second meaning to the same ‘`( ... )`’ construct because there is no conflict in practice between the two meanings. Here is an explanation of this feature:

'`\digit`' After the end of a '`( ... )`' construct, the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use '\' followed by *digit* to mean "match the same text matched the *digit*'th time by the '`( ... )`' construct." The '`( ... )`' constructs are numbered in order of commencement in the regexp.

The strings matching the first nine '`( ... )`' constructs appearing in a regular expression are assigned numbers 1 through 9 in order of their beginnings. '`\1`' through '`\9`' may be used to refer to the text matched by the corresponding '`( ... )`' construct.

For example, '`(.+)\1`' matches any non empty string that is composed of two identical halves. The '`(.+)`' matches the first half, which may be anything non empty, but the '`\1`' that follows must match the same exact text.

'`\b`' matches the empty string, but only if it is at the beginning or end of a word. Thus, '`\bfoo\b`' matches any occurrence of '`foo`' as a separate word. '`\bball(s|)\b`' matches '`ball`' or '`balls`' as a separate word.

'`\B`' matches the empty string, provided it is *not* at the beginning or end of a word.

'`\<`' matches the empty string, but only if it is at the beginning of a word.

'`\>`' matches the empty string, but only if it is at the end of a word.

'`\w`' matches any word-constituent character.

'`\W`' matches any character that is not a word-constituent.

3.7.2 Replacing regular expressions

Also the replacement string has some special feature when doing a regular expression search and replace. Exactly as during the search, '\' followed by *digit* stands for "the text matched the *digit*'th time by the '`( ... )`' construct in the search expression". Moreover, '`\0`' represent the whole string matched by the regular expression. Thus, for instance, the replace string '`\0\0`' has the effect of doubling any string matched.

Another example: if you search for '`(a+)(b+)`', replacing with '`\2x\1`', you will match any string composed by a series of '`a`''s followed by a series of '`b`''s, and you will replace it with the string obtained by moving the '`a`' in front of the '`b`''s, adding moreover '`x`' inbetween. For instance, '`aaaab`' will be matched and replaced by '`bxaaaa`'.

Note that the backslash character can escape itself. Thus, in order to put a backslash in the replacement string, you have to use '`\\`'.

3.8 Automatic Preferences

Automatic preferences let you set up a custom configuration that will be automatically loaded whenever you open a file with a given extension. For instance, you could like a TAB size of three while editing C sources, but eight could be more palatable while writing electronic mail.

The use of this feature is definitely straightforward: you simply use the '`Save AutoPrefs`' menu item (or the `SaveAutoPrefs` command; see Section 4.9.16 [SaveAutoPrefs], page 36) when the current document has the given extension, and the current configuration suits your tastes. The internal status of a series of options will be recorded as a macro, containing commands which reproduce the current configuration. The macro is then saved in the `$HOME/.ne` directory (which is created, if necessary) with the name given by the extension, postfixed with '`#ap`'. Thus, the C sources automatic preferences file will be named '`c#ap`', the one of TEX files '`tex#ap`', and so on.

The macros are generated with short or long command names depending on the status of the verbose macros flag. See Section 4.9.12 [VerboseMacros], page 35.

Automatic preferences file are loaded and executed whenever a file with a known extension is opened. Note that you can edit manually such files, and even insert commands, but any command which does something else than setting a flag will be rejected, and an error message will be issued.

3.9 Emergency Save

Sometimes it can happen that `ne` is interrupted by an abnormal event (for instance, the crash of your terminal). In this cases, it will try, if it is given the possibility, to save all unsaved documents in its current directory. Named documents will have their name prefixed with a '`#`'. Unnamed documents will be named using hexadecimal numbers obtained by some addresses in memory which will make them unique.

4 Commands

Everything `ne` can do is specified through a command. Commands can be manually typed on the command line, bound to a key, to a menu item, or grouped into macros for easier manipulation. If you want to fully exploit the power of `ne`, you will be faced sooner or later with using directly commands.

4.1 Generals

Every command of `ne` has a long and a short name. Except in a very few cases, the short name is given by two or three letters which are the initials of the words which form the long name. Thus, for instance, `SearchBack` has short name `SB`. However, most used commands such as `Exit` have one-letter short names (`X`), and `StatusBar`'s short name is `ST` in order to avoid clashes with `SearchBack`'s.

A command has always at most an argument. This is a chosen limitation, which allows `ne`'s parsing of commands and macros to be very fast (a hash table with no conflicts decodes the command name). Moreover, it almost cancel all problems related to delimitators, escape characters, *et cetera*. The unique argument can be a number, a string, or a flag modifier. You can easily distinguish these three cases even without this manual by looking at what the `Help` command says about the given command. Note that when a command argument is enclosed in square brackets, it is optional.

Strings are general purpose arguments. Numbers are used to modify internal parameters, such as the size of a TAB. Flag modifiers are an optional number which is interpreted as follows:

- 0 means clearing the flag;
- 1 (or any positive number) means setting the flag;
- no number means toggling the flag.

Thus, `StatusBar 1` will activate that status bar, while `I` will toggle insert/overstrike. This design choice is due to the fact that, most of the time, during interactive editing one is faced with *changing* a flag; for instance, one is in insert mode and wants to overstrike, or viceversa. Absolute settings (i.e., with a number) are useful essentially for macros. It is reasonable to keep the fastest approach for the most frequent interactive event. When a number or a string is required, and the argument is optional, most of the times the user will be prompted to typing the argument on the command line.

When a number represent the times `ne` should repeat an action, it is always understood that the command will terminate anyway when the conditions for applying it are not longer true. For instance, the `Paragraph` commands accepts the number of paragraphs to format. But if not enough paragraphs exists in the text, only the available ones will be formatted.

This allows to easily perform operation on the whole text by specifying preposterously huge numbers as arguments. `ToUpper 2000000000` will (hopefully) upper the case of all the words in the document. Note that this is much faster than recording a macro with a command and playing it many times, because the command has to be parsed just one time.

In any case, if a macro or a repeated operation takes too long, you can stop it using the interrupt key (*`Control`*`-\`).

In order to handle situations such as an argument string starting with a space, `ne` implements the following simple mechanism: an argument string can be included in quotes, provided that the closing quote is the last character of the command line.

The main advantage of this approach is that no escape convention is necessary when putting quotes inside a quoted string, since `ne` can use contextual information in order to tell the real delimitator. The only case needing a special treatment is the case of an argument string starting

and ending with a quote: unless it is again enclosed in quotes, `ne` will believe the quotes are delimitators, and act accordingly.

4.2 File Commands

These commands allow to open and save files. They all act in the context of the current document (i.e., the document displayed when the command is issued).

4.2.1 Clear

Syntax: `Clear`
Abbreviation: `CL`

destroys the contents of the current document and of its undo buffer. Moreover, the document becomes unnamed. If the current document is marked as modified, you have to confirm the action.

4.2.2 Open

Syntax: `Open [`*`filename`*`]`
Abbreviation: `O`

loads the file specified by the *filename* string into the current document.

If the optional *filename* argument is not specified, the file requester is opened, and you are prompted to select a file (you can inhibit the file requester opening by using the `NoFileReq` command; see Section 4.9.6 [NoFileReq], page 34).

If you escape from the file requester, you can input the file name on the command line, the default being the current document name, if available.

If at the time the command is issued the current document is marked as modified, you have to confirm the action.

4.2.3 OpenNew

Syntax: `OpenNew [`*`filename`*`]`
Abbreviation: `ON`

is the same as `Open`, but loads the file specified by the *filename* string into a new document. See Section 4.2.2 [Open], page 22.

4.2.4 Save

Syntax: `Save`
Abbreviation: `S`

saves the current document using its default file name.

If the current document is unnamed, the file requester is opened, and you are prompted to select a file (you can inhibit the file requester opening by using the `NoFileReq` command; see Section 4.9.6 [NoFileReq], page 34).

If you escape from the file requester, you can input the file name on the command line.

4.2.5 SaveAs

Syntax: `SaveAs [`*`filename`*`]`
Abbreviation: `SA`

saves the current document using the specified string as file name.

If the optional *filename* argument is not specified, the file requester is opened, and you are prompted to select a file (you can inhibit the file requester opening by using the `NoFileReq` command; see Section 4.9.6 [NoFileReq], page 34).

If you escape from the file requester, you can enter the file name on the input line, the default being the current document name, if available.

4.3 Document Commands

These command allow to manipulate the circular list of documents of `ne`.

4.3.1 Quit

Syntax: `Quit`
Abbreviation: `Q`

closes all documents and exits. If some document is modified, you have to confirm the action.

4.3.2 Exit

Syntax: `Exit`
Abbreviation: `X`

saves all modified documents, closes them and exits. If some document cannot be saved, the action is suspended and an error message is issued.

4.3.3 NewDoc

Syntax: `NewDoc`
Abbreviation: `N`

creates a new, empty, unnamed document which becomes the current document. The position of the document in the document list is just after the current document. The preferences of the new document are obtained by cloning the preferences of the current document.

4.3.4 CloseDoc

Syntax: `CloseDoc`
Abbreviation: `CD`

closes the current document. The document is removed from `ne`'s list and, in case it is the only existing document, `ne` exits. If the document was modified from the last save, you have to confirm your choice.

4.3.5 NextDoc

Syntax: `NextDoc`
Abbreviation: `ND`

sets as current document the next document in the document list.

4.3.6 PrevDoc

Syntax: `PrevDoc`
Abbreviation: `PD`

sets as current document the previous document in the document list.

4.3.7 SelectDoc

Syntax: `SelectDoc`
Abbreviation: `SD`

opens a requester containing the names of all the documents in memory. You can select the document you want to edit.

If you escape from the requester, you can enter the document name on the input line, the default being the current document name, if available.

This command is really useful only if you have a large (say, more than 10) number of documents loaded. Otherwise, `NextDoc` and `PrevDoc` should be enough. See Section 4.3.5 [NextDoc], page 23, and Section 4.3.6 [PrevDoc], page 23.

4.4 Clip Commands

These commands control the clipping system. `ne` can have any number of clips, which are distinguished by an integer index. Most commands act on the current clip, which can be selected with `ClipNumber`. Note that clips can be copied and pasted in two ways—normally or vertically.

Note that by using the `Through` command you can automatically pass a (possibly vertical) block of text through any filter (such as `sort` under UN*X).

4.4.1 Mark

Syntax: `Mark [0|1]`
Abbreviation: `M`

sets the mark at the current position or cancels the previous mark. The mark can then be used in order to perform clip operations. The clip commands act on the characters laying between the mark and the cursor.

If you call this command with no arguments, it will toggle the mark. If you specify 0 or 1, the mark will be canceled or set to the current position, respectively. A capital '`M`' appears on the status bar, if the mark is active.

See Section 4.6.1 [Record], page 28, for the reason why the mark is implemented as a flag.

4.4.2 MarkVert

Syntax: `MarkVert [0|1]`
Abbreviation: `MV`

is the same as `Mark`, but the mark is interpreted as vertical by the clip handling commands. This means that the region manipulated by the cut/paste commands is the rectangle having as vertices the cursor and the mark. Moreover, a capital '`V`', rather than a capital '`M`', will appear on the status bar. Vertical cut/paste operations are most useful for handling structured program indentation.

4.4.3 Copy

Syntax: `Copy [`*`n`*`]`
Abbreviation: `C`

copies the contents of the characters laying between the cursor and the mark into the clip specified by the optional numeric argument, the default clip being the current clip, which can be set with the `ClipNumber` command (see Section 4.4.10 [ClipNumber], page 25). If the current mark was vertical, the rectangle of characters defined by the cursor and the mark is copied instead.

4.4.4 Cut

Syntax: `Cut [`*`n`*`]`
Abbreviation: `CU`

acts just like `Copy`, but also deletes the block being copied.

4.4.5 Paste

Syntax: `Paste [`*`n`*`]`
Abbreviation: `P`

pastes the contents of specified clip, the default being current clip, which can be set with the `ClipNumber` command (see Section 4.4.10 [ClipNumber], page 25), at the cursor position.

4.4.6 PasteVert

Syntax: `PasteVert [n]`
Abbreviation: `PV`

pastes vertically the contents of the specified clip, the default being the current clip. Each line of the clip is inserted on consecutive lines at the horizontal cursor position.

4.4.7 Erase

Syntax: `Erase`
Abbreviation: `E`

acts like `Cut`, but the block is just deleted, and not copied into any clip.

4.4.8 OpenClip

Syntax: `OpenClip [filename]`
Abbreviation: `OC`

loads the given file name as the current clip (just as if you copied it; see Section 4.4.3 [Copy], page 24).

If the optional *filename* argument is not specified, the file requester is opened, and you are prompted to select a file (you can inhibit the file requester opening by using the `NoFileReq` command; see Section 4.9.6 [NoFileReq], page 34).

If you escape from the file requester, you can enter the file name on the input line.

4.4.9 SaveClip

Syntax: `SaveClip [filename]`
Abbreviation: `SC`

saves the current clip on the given file name.

If the optional *filename* argument is not specified, the file requester is opened, and you are prompted to select a file (you can inhibit the file requester opening by using the `NoFileReq` command; see Section 4.9.6 [NoFileReq], page 34).

If you escape from the file requester, you can enter the file name on the input line.

4.4.10 ClipNumber

Syntax: `ClipNumber [n]`
Abbreviation: `CN`

sets the current clip number. This number is used by `OpenClip` and `SaveClip`, and by `Copy`, `Cut` and `Paste` if they are called without any argument. Its default value is zero. *n* is limited only by the integer size of the machine `ne` is running on.

If the optional argument *n* is not specified, you can enter it on the input line, the default being the current clip number.

4.4.11 Through

Syntax: `Through [command]`
Abbreviation: `T`

asks the shell to execute *command*, piping the current block in the standard input, and replacing it with the output of the command. This command is most useful with filters, such as `sort`. Its practical effect is to pass the block through the specified filter.

Note that by selecting an empty block (or equivalently by having the mark unset) you can use `Through` in order to insert the output of any UN*X command in your file.

If the optional argument *command* is not specified, you can enter it on the input line.

4.5 Search Commands

These commands control the search system. `ne` offers two complementary searching techniques: an simple, fast exact matching search (optionally, modulo case), and a very flexible and powerful, but slower, regular expression search based on the GNU `regex` library (again, optionally modulo case).

4.5.1 Find

Syntax: `Find [`*`pattern`*`]`
Abbreviation: `F`

searches for the given pattern. The cursor is positioned on the first occurrence of the pattern, or an error message is given. The direction and the case sensitivity of the search are established by the value of the back search and case sensitive search flags. See Section 4.5.8 [SearchBack], page 28, and Section 4.5.9 [CaseSearch], page 28.

If the optional argument *pattern* is not specified, you can enter it on the input line, the default being the last pattern used.

4.5.2 FindRegExp

Syntax: `FindRegExp [`*`pattern`*`]`
Abbreviation: `FX`

searches the buffer for the given extended regular expression (see Section 3.7 [Regular Expressions], page 16) . The cursor is positioned on the first string matching the expression. The direction and the kind of search are established by the value of the back search and case sensitive search flags. See Section 4.5.8 [SearchBack], page 28, and Section 4.5.9 [CaseSearch], page 28.

If the optional argument *pattern* is not specified, you can enter it on the input line, the default being the last pattern used.

4.5.3 Replace

Syntax: `Replace [`*`string`*`]`
Abbreviation: `R`

moves on the first match of the most recent find string or regular expression, and then asks you which action to perform. You can choose among:

- replacing the string found with the given string and moving to the next match (‘`Yes`’);
- moving to the next match (‘`No`’);
- replacing the string found with the given string, and stop the search (‘`Last`’);
- stopping immediately the search (‘`Quit`’);
- replacing *all* occurrences of the find string with the given string (‘`All`’);
- reverting the search direction (‘`Backward`’ or ‘`Forward`’); this choice will also modify the value of the back search flag. See Section 4.5.8 [SearchBack], page 28.

This command is mainly useful for interactive editing. `ReplaceOnce`, `ReplaceAll` and `RepeatLast` are more suited to macros.

If no find string was ever specified, you can enter it on the input line. If the optional argument *string* is not specified, you can enter it on the input line, the default being the last string used. When the last search was a regular expression search (see Section 4.5.2 [FindRegExp], page 26), there are some special features you can use in the replace string (see Section 3.7 [Regular Expressions], page 16) .

Note that normally a search starts just one character after the cursor. However, when `Replace` is invoked, the search starts at the character just *under* the cursor, so that you can safely `Find` a pattern and `Replace` it without having to move back.

Warning: when recording a macro (see Section 4.6.1 [Record], page 28), there is no trace in the macro of your interaction with `ne` during the replacement process. When the macro is played, you will have again to choose which actions to perform. If you want to apply automatic replacement of strings for a certain number of times, you should look at Section 4.5.4 [ReplaceOnce], page 27, Section 4.5.5 [ReplaceAll], page 27, and Section 4.5.6 [RepeatLast], page 27.

4.5.4 ReplaceOnce

Syntax: `ReplaceOnce [`*`string`*`]`
Abbreviation: `R1`

acts just like `Replace`, but without any interaction with you (unless there is no find string). The first string matched by the last search pattern, if existing, is replaced by the given replacement string.

If the optional argument *string* is not specified, you can enter it on the input line, the default being the last string used.

4.5.5 ReplaceAll

Syntax: `ReplaceAll [`*`string`*`]`
Abbreviation: `RA`

is similar to `ReplaceOnce`, but replaces *all* occurrences of the last search pattern with the given replacement string.

If the optional argument *string* is not specified, you can enter it on the input line, the default being the last string used.

Note that `Undo` will restore *all* the occurrences of the search pattern this command replaced. See Section 4.7.1 [Undo], page 30.

4.5.6 RepeatLast

Syntax: `RepeatLast [`*`times`*`]`
Abbreviation: `RL`

repeats for the given number of times the last find or replace operation (with replace we mean here a single replace, even if the last `Replace` operation ended with a global substitution).

This command is mainly useful for researching a given number of times, or replacing something a given number of times. The standard technique for accomplishing this is:

1. `Find` (or `FindRegExp`) the string you are interested in;
2. if you want to repeat a replace operation, `ReplaceOnce` with the replacement string you are interested in;
3. now issue a `RepeatLast` *`n`*`-1` command, where *n* is the number of occurrences you wanted to skip over, or replace.

The important thing about this sequence of actions is that it will work this way even in a macro. The `Replace` command cannot be used in a macro unless you really want to interact with `ne` during the macro execution. This is the only reason why the commands `ReplaceAll` and `ReplaceOnce` are provided.

4.5.7 MatchBracket

Syntax: `MatchBracket`
Abbreviation: `MB`

moves the cursor over the bracket associated to the bracket the cursor is on. If the cursor is not on a bracket, or there is no bracket associated to the current one, an error message is issued. '`{}`', '`()`', '`[]`' and '`<>`' are recognized.

4.5.8 SearchBack

Syntax: `SearchBack [0|1]`
Abbreviation: `SB`

sets the back search flag. When this flag is true, every search or replacement command is performed backwards.

If you call this command with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively. A lower case '`b`' will appear on the status bar if the flag is true.

Note that this flag can be also set by the interaction happening during a replace. See Section 4.5.3 [Replace], page 26.

4.5.9 CaseSearch

Syntax: `CaseSearch [0|1]`
Abbreviation: `CS`

sets the case sensitivity flag. When this flag is true, the search commands distinguish between the upper case and lower case versions of a letter. By default the flag is false, since this seems to be what most user want.

If you call this command with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively. A lower case '`c`' will appear on the status bar if the flag is true.

4.6 Macros Commands

Macros are lists of commands. Any series of operations which has to be performed frequently is a good candidate for being a macro.

Macros can be written manually: they are just ASCII files, each command occupying a line (lines starting with any non-alphabetical character are considered comments). But the real power of macros is that they be recorded during the normal usage of `ne`. When the recording terminates, the operations which have been recorded can be saved for later use. Note that each document has a current macro (the last macro which has been opened or recorded).

4.6.1 Record

Syntax: `Record [0|1]`
Abbreviation: `Rec`

sets the recording state flag. When this flag becomes true, a macro recording is initiated. When it becomes false, the macro recording is stopped, and the macro can be played (see Section 4.6.2 [Play], page 28).

If you call this command with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively. An upper case '`R`' will appear on the status bar if the flag is true.

The reason for providing a flag instead of an explicit start/stop recording command pair is that this way it is possible to bind to the same key the start and stop recording command, while still being able to specifying "absolute" menu items (by using `Record 0` and `Record 1`). For instance, the default key binding for *`Control-T`* is simply `Record`, which means that this shortcut can be used both for initiating and for terminating a macro recording.

4.6.2 Play

Syntax: `Play [`*`times`*`]`
Abbreviation: `PL`

plays the current macro for the given number of times. If the optional argument *times* is not specified, you can enter it on the input line.

A (possibly iterated) macro execution terminates as soon as its stream of instructions is exhausted, or one of its commands returns an error. This means that, for instance, you can perform some complex operation on all the lines contaning a certain pattern by recording a macro that searches for the pattern and performs the operation, and then playing it a preposterously huge number of times. Note that the execution of a macro can be interrupted by *Control*-\.

4.6.3 Macro

Syntax: `Macro [`*filename*`]`
Abbreviation: `MA`

executes the given file name as a macro.

If the optional *filename* argument is not specified, the file requester is opened, and you are prompted to select a file (you can inhibit the file requester opening by using the `NoFileReq` command; see Section 4.9.6 [NoFileReq], page 34).

If you escape from the file requester, you can input the file name on the command line.

Note that macros whose name does not conflict with a command can be called without using `Macro`. Whenever `ne` is required to perform a command it cannot find in its internal tables, it will look for a macro named as the command. If also this search fails, `ne` looks in `$HOME/.ne` for a file named as the command.

Warning: in order to (greatly) improve efficiency, the first time a macro is executed it is cached into a hash table and is kept *forever* in memory (unless the `UnloadMacros` command is issued; see Section 4.6.6 [UnloadMacros], page 30). The next time a macro with the same file name is invoked, the cached list is searched for it before accessing the file using a case insensitive string comparison. That is, if you call '`~/foobar/macro`', a subsequent call for '`/usr/MACRO`' or just '`MaCrO`' will use the cached version. Note that the cache table is global to `ne`.

4.6.4 OpenMacro

Syntax: `OpenMacro [`*filename*`]`
Abbreviation: `OM`

loads the given file name as the current macro (just as if you `Record`ed it; see Section 4.6.1 [Record], page 28).

If the optional *filename* argument is not specified, the file requester is opened, and you are prompted to select a file (you can inhibit the file requester opening by using the `NoFileReq` command; see Section 4.9.6 [NoFileReq], page 34).

If you escape from the file requester, you can input the file name on the command line.

4.6.5 SaveMacro

Syntax: `SaveMacro [`*filename*`]`
Abbreviation: `SM`

saves the current macro on the given file name.

If the optional *filename* argument is not specified, the file requester is opened, and you are prompted to select a file (you can inhibit the file requester opening by using the `NoFileReq` command; see Section 4.9.6 [NoFileReq], page 34).

If you escape from the file requester, you can input the file name on the command line.

This command is of course most useful for saving macros you just recorded. The macros can then be loaded as normal text files for further editing, if necessary.

4.6.6 UnloadMacros

Syntax: `UnloadMacros`
Abbreviation: `UM`

frees the macro cache list. After this command, the `Macro` command will be forced to search for the file containing the macro it has to play.

This command is really useful only if you are experimenting with a macro bound to some keystroke, and you are interactively modifying it and playing it. `UnloadMacros` forces `ne` to look for the newer version available.

4.7 Undo Commands

The following commands control the undo system.

4.7.1 Undo

Syntax: `Undo [n]`
Abbreviation: `U`

undoes the last *n* actions. If *n* is not specified, it is assumed to be one. Once you undo a number of actions, you can `Redo` them (or part of them; see Section 4.7.2 [Redo], page 30).

4.7.2 Redo

Syntax: `Redo [n]`
Abbreviation: `RE`

redoes the last *n* actions. If *n* is not specified, it is assumed to be one. You can only `Redo` actions which have been `Undo`ne. See Section 4.7.1 [Undo], page 30.

4.7.3 UndelLine

Syntax: `UndelLine [n]`
Abbreviation: `UL`

inserts at the cursor position for *n* times the last non-empty line which was deleted with the `DeleteLine` command. If *n* is not specified, it is assumed to be one.

This command is most useful in that it allows a very fast way of moving one line around. Just delete it, and undelete it somewhere else. It is also an easy way to replicate a line without getting involved with clips.

Note that this command works independently of the status of the undo flag. See Section 4.7.4 [DoUndo], page 30.

4.7.4 DoUndo

Syntax: `DoUndo [0|1]`
Abbreviation: `DU`

sets the flag which enables or disables the undo system. When you turn the undo system off, all the recorded actions are discarded, and the undo buffers are reset.

If you call this command with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively. A lower case '`u`' will appear on the status bar if the flag is true.

The usefulness of this option relies in the fact the undo system is a major memory eater. If you plan to do massive editing (say, cutting and pasting megabytes of text) it is a good idea to disable the undo system, both for improving (doubling) performance and for using less (half) memory. Except for this, on a virtual memory system I see no reason to not keep the undo flag always true, and this is indeed the default.

4.8 Formatting Commands

The following commands allow simple formatting operations on the text. Note that for `ne` a paragraph is delimited by an empty line.

4.8.1 Center

Syntax: `Center [n]`
Abbreviation: `CE`

centers *n* lines from the cursor position onwards. If *n* is not specified, it is assumed to be one. The lines are centered with spaces, relatively to the value of the right margin as set by the `RightMargin` command. See Section 4.8.6 [RightMargin], page 31.

4.8.2 Paragraph

Syntax: `Paragraph [n]`
Abbreviation: `PA`

reformats *n* paragraphs from the cursor position onwards. If *n* is not specified, it is assumed to be one. The paragraph are formatted relatively to the value of the right margin as set by the `RightMargin` command. See Section 4.8.6 [RightMargin], page 31.

Do not ever ask me to implement justified paragraphing. I *hate* non-proportional justified text.

If you think paragraphing should insert "smart" spaces after full stops and colons, and do other such "smart" things, you should consider using a text formatter. TEX is usually the best choice.

4.8.3 ToUpper

Syntax: `ToUpper [n]`
Abbreviation: `TU`

uppers the case of the letters from the cursor position up to the end of a word, and moves to the fist letter of next word for *n* times.

The description of the command may seem a little bit cryptic. What is really happening is that there are situations where you want to upper case only the last part of a word. In this case, you just have to position the cursor in the first character you want to upper case, and use this command with no argument.

If you apply `ToUpper` on the first character of a word, it will just upper case *n* words.

4.8.4 ToLower

Syntax: `ToLower [n]`
Abbreviation: `TL`

acts exactly like `ToUpper`, but lowers the case. See Section 4.8.3 [ToUpper], page 31.

4.8.5 Capitalize

Syntax: `Capitalize [n]`
Abbreviation: `CA`

acts exactly like `ToUpper`, but capitalizes, i.e., makes the first letter upper case and the other ones lower case. See Section 4.8.3 [ToUpper], page 31.

4.8.6 RightMargin

Syntax: `RightMargin [n]`
Abbreviation: `RM`

sets the right margin for all formatting operations, and for `WordWrap` (see Section 4.8.7 [WordWrap], page 32).

If the optional argument *n* is not specified, you can enter it on the input line, the default being the current value of the right margin.

A value of 0 for *n* will force `ne` to use (what it thinks it is) the current screen width as right margin.

4.8.7 WordWrap

Syntax: `WordWrap [0|1]`
Abbreviation: `WW`

sets the word wrap flag. When this flag is true, `ne` will automatically break lines of text longer than the current right margin (see Section 4.8.6 [RightMargin], page 31) while you type them.

If you call this command with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively. A lower case '`w`' will appear on the status bar if the flag is true.

4.8.8 AutoIndent

Syntax: `AutoIndent [0|1]`
Abbreviation: `AI`

sets the auto indent flag. When this flag is true, `ne` will automatically insert tabs and spaces on a new line (created by an `InsertLine` command, or by automatic word wrapping) in such a way to replicate the initial spaces of the previous line. Most useful for indenting programs.

If you call this command with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively. A lower case '`a`' will appear on the status bar if the flag is true.

`AutoIndent` features a nice interaction with `Undo`. Whenever a new line is created, the insertion of spaces is recorded as a separate action in the undo buffer (with respect to the line creation). If you are not satisfied with the indentation, just give the `Undo` command and the indentation will disappear (but the new line will remain in place, since its creation has been recorded as a separate action). See Section 4.7.1 [Undo], page 30.

4.9 Preferences Commands

These commands allow you to set your preferences, i.e., the value of a series of flags which modify the behaviour of `ne` (some of the flag commands, like the command for the indent flag, appear in other sections). The status of the flags can be saved and restored later (the file saved is just a macro which suitably sets the flags). The back search and the read only flags are not saved, because they do not represent a preference, but rather a temporary state. The escape time is global to `ne`, and it is not saved. The turbo parameter is better set at run time by `ne`. However, you can add manually to a preferences file any preferences command (such as `EscapeTime` or `Turbo`); usually, this will be done to the default preferences file `$HOME/.ne/.default#ap`.

Note that there is an automatic preferences system, which loads automagically a preferences file related to the extension of the name of a file. The automatic preferences files are kept in a directory named `.ne` (in your home directory), and they are named as an extension postfixed with '`#ap`'. Each time you open a file whose name has an extension for which there is an automatic preferences file, the latter is executed. If you want to inhibit this process, you can clear the automatic preferences flag. See Section 4.9.1 [AutoPrefs], page 32.

4.9.1 AutoPrefs

Syntax: `AutoPrefs [0|1]`
Abbreviation: `AP`

sets the automatic preferences flag. If this flag is true, each time an `Open` command is executed and a file is loaded, `ne` will look for an automatic preferences file in the `$HOME/.ne` directory. The preferences file name is given by the extension of the file loaded, postfixed with '`#ap`'. Thus, for instance, C sources have an associated `c#ap` file. See Section 3.8 [Automatic Preferences], page 18.

If you call this command with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively. A lower case '`p`' will appear on the status bar if the flag is true.

4.9.2 Binary

Syntax: `Binary [0|1]`
Abbreviation: `B`

sets the binary flag. When this flag is true, loading and saving a document is performed in a different way. On loading, only nulls are considered newlines; on saving, nulls are saved instead of newlines. This allows you to edit a binary file, fix some text in it, and save it without modifying anything else. Normally, linefeeds, carriage returns and nulls are considered newlines, so that what you load will have all nulls and carriage returns substituted by newlines when saved.

Note that since usually binary files contain a great number of nulls, and every null will be considered a line terminator, the memory necessary for loading a binary file can be several times bigger than the length of the file. Thus, binary editing within `ne` should be considered not a normal, but rather an exceptional activity.

If you call this command with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively. An upper case '`B`' will appear on the status bar if the flag is true.

4.9.3 Insert

Syntax: `Insert [0|1]`
Abbreviation: `I`

sets the insert flag. If this flag is true, the text you type is inserted, otherwise it overwrites the existing characters. This is also true of the `InsertChar` command.

If you call this command with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively. A lower case '`i`' will appear on the status bar if the flag is true.

4.9.4 FastGUI

Syntax: `FastGUI [0|1]`
Abbreviation: `FG`

sets the fast graphical user interface flag. When this flag is true, `ne` tries to print as few as possible while displaying menus and the status bar. In particular, menu items are highlighted by the cursor only, and the status bar is not highlighted (which allows to print it with less characters). This option is very useful if you are using `ne` through a slow connection.

If you call this command with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively.

4.9.5 FreeForm

Syntax: `FreeForm [0|1]`
Abbreviation: `FF`

sets the free form flag. When this flag is true, you can move with the cursor anywhere on the screen, even where no text is present (however, you cannot move inside the space expansion of a TAB character).

If you call this command with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively. A lower case '`f`' will appear on the status bar if the flag is true.

The issue free-form-versus-non-free-form is a major religious war which is engaging hackers, lusers and programmers from day one. The due of the implementor is to allow both choices, and to set as default the correct one (in his humble opinion). In this case, non-free-form.

4.9.6 NoFileReq

Syntax: `NoFileReq [0|1]`
Abbreviation: `NFR`

sets the file requester flag. When this flag is true, the file requester is never opened, under any circumstances.

If you call this command with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively.

4.9.7 StatusBar

Syntax: `StatusBar [0|1]`
Abbreviation: `ST`

sets the status bar flag. When this flag is true, the status bar is displayed at the bottom of the screen. There are only two reasons to turn off the status bar I am aware of:

- if you are using `ne` through a slow connection, updating the line/column indicator can really slow down editing;
- scrolling caused by cursor movement on terminals which do not allow to set a scrolling region can produce annoying flashes at the bottom of the screen.

If you call this command with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively.

4.9.8 ReadOnly

Syntax: `ReadOnly [0|1]`
Abbreviation: `RO`

sets the read only flag. When this flag is true, no editing can be performed on the document (any such attempt produces an error message). This flag is automatically set whenever you open a file which you cannot write to. See Section 4.2.2 [Open], page 22.

If you call this command with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively. A lower case '`r`' will appear on the status bar if the flag is true.

4.9.9 EscapeTime

Syntax: `EscapeTime [`*n*`]`
Abbreviation: `ET`

sets the escape time. The `Escape` key is recognized as such after *n* tenths of second (see Chapter 7 [Motivations and Design], page 47). Along slow connections, it can happen that the default value of 10 is too low: in this case, escape sequences (e.g., those of the arrow keys) could be erroneously broken into an escape and some spurious characters. Rising the escape time usually solves this problem. Allowed values range from 0 to 255.

Note that the escape time is global to `ne`, and it is not saved. However, you can add manually to a preferences file an `EscapeTime` command.

4.9.10 TabSize

Syntax: `TabSize [size]`
Abbreviation: `TS`

sets the number of spaces `ne` will use when expanding a TAB character.

If the optional argument *size* is not specified, you can enter it on the input line, the default being the current TAB size. Allowed values are strictly between 0 and half the width of the screen.

4.9.11 Turbo

Syntax: `Turbo [steps]`
Abbreviation: `TUR`

sets the turbo parameter. When it is nonzero, iterated actions and global replaces will update at most *steps* line of the screen; then, update will be delayed to the end of the action.

This feature is most useful when massive operations (such as replacing thousands of occurrences of a pattern) have to be performed. After having updated *steps* lines, `ne` can proceed at maximum speed, because no visual update has to be performed.

The value of the turbo parameter has to be adapted to the kind of terminal you are using. Very high values (or zero, which completely disables the delayed update) can be good on high-speed terminals, since the time required for the visual updates is very small, and it is always safer to look at what the editor is really doing. On slow terminals, however, small values ensure that operations such as paragraph formatting will not take too long.

You have to be careful about setting the turbo parameter too low. `ne` keeps track internally of the part of the screen which needs refresh in a very rough way. This means that a value of less than, say, 8 will force it to do a lot of unnecessary refresh.

The default value of this parameter is given by twice the number of lines of the screen, which for several reasons does seem to offer a good value.

4.9.12 VerboseMacros

Syntax: `VerboseMacros [0|1]`
Abbreviation: `VM`

sets the verbose macros flag. When this flag is true, all macros generated by recording or by automatic preferences saving will contain full names, instead of short names. This is highly desirable if you are going to manually editing the macro, but it can slow down considerably the parsing of the commands.

If you call this command with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively. A lower case '`v`' will appear on the status bar if the flag is true.

The only reason to use this flag is when recording a macro that will be played a great number of times. Automatic preferences files are too short to be an issue with respect to execution timing.

4.9.13 LoadPrefs

Syntax: `LoadPrefs [filename]`
Abbreviation: `LP`

loads the given preference file, and sets the current preferences accordingly.

If the optional *filename* argument is not specified, the file requester is opened, and you are prompted to select a file (you can inhibit the file requester opening by using the `NoFileReq` command; see Section 4.9.6 [NoFileReq], page 34). If you escape from the file requester, you can input the file name on the command line.

Note that a preferences file is just a macro containing option modifiers only. You can manually edit a preferences file for special purposes, such as filtering out specific settings. See Chapter 6 [Hints and Tricks], page 45.

4.9.14 SavePrefs

Syntax: `SavePrefs [filename]`
Abbreviation: `SP`

saves the current preferences on the given file.

If the optional *filename* argument is not specified, the file requester is opened, and you are prompted to select a file (you can inhibit the file requester opening by using the `NoFileReq` command; see Section 4.9.6 [NoFileReq], page 34). If you escape from the file requester, you can input the file name on the command line.

4.9.15 LoadAutoPrefs

Syntax: `LoadAutoPrefs`
Abbreviation: `LAP`

loads the preferences file in `$HOME/.ne` relative to the current document file name. If the current file name has no extension, the default preferences are loaded. See Section 4.9.1 [AutoPrefs], page 32.

4.9.16 SaveAutoPrefs

Syntax: `SaveAutoPrefs`
Abbreviation: `SAP`

saves the current preferences on the file in `$HOME/.ne` relative to the current document file name. If the current file name has no extension, an error message is issued. See Section 4.9.1 [AutoPrefs], page 32.

4.9.17 SaveDefPrefs

Syntax: `SaveDefPrefs`
Abbreviation: `SDP`

saves the current preferences on the `$HOME/.ne/.default#ap` file. This file is always loaded by `ne` at startup.

4.10 Navigation Commands

These commands allows you to move through a document. Besides the standard commands which allow you to move by lines, pages, *et cetera*, `ne` has bookmarks which let you mark a position in a file, in order to move to the same position later.

4.10.1 MoveLeft

Syntax: `MoveLeft [n]`
Abbreviation: `ML`

moves the cursor on the left by one character *n* times. If the optional *n* argument is not specified, it is assumed to be one.

4.10.2 MoveRight

Syntax: `MoveRight [n]`
Abbreviation: `MR`

moves the cursor on the right by one character *n* times. If the optional *n* argument is not specified, it is assumed to be one.

4.10.3 LineUp

Syntax: `LineUp [n]`
Abbreviation: `LU`

moves the cursor up by one line *n* times. If the optional *n* argument is not specified, it is assumed to be one.

4.10.4 LineDown

Syntax: `LineDown [n]`
Abbreviation: `LD`

moves the cursor down by one line *n* times. If the optional *n* argument is not specified, it is assumed to be one.

4.10.5 GotoLine

Syntax: `GotoLine [line]`
Abbreviation: `GL`

moves the cursor on the *line*th line of the file. If *line* is greater than the number of lines of the file, the cursor is moved on the last line.

If the optional argument *line* is not specified, you can enter it on the input line, the default input being the current line number.

4.10.6 GotoColumn

Syntax: `GotoColumn [column]`
Abbreviation: `GC`

moves the cursor on the *column*th column of the file.

If the optional argument *line* is not specified, you can enter it on the input line, the default input being the current column number.

4.10.7 GotoMark

Syntax: `GotoMark`
Abbreviation: `GM`

moves the cursor to the current mark, if existing. See Section 4.4.1 [Mark], page 24.

This command is mainly useful if you forgot where you started marking. If you want to record a position in a file and jump on it later, you may want to use a bookmark. See Section 4.10.19 [SetBookmark], page 39.

4.10.8 PrevPage

Syntax: `PrevPage [n]`
Abbreviation: `PP`

moves the cursor *n* pages backward, if the cursor is on the first line of the screen; otherwise moves the cursor on the first line of the screen, and moves by *n*-1 pages. If the optional *n* argument is not specified, it is assumed to be one.

4.10.9 NextPage

Syntax: `NextPage [n]`
Abbreviation: `NP`

moves the cursor *n* pages forward, if the cursor is on the last line of the screen; otherwise moves the cursor on the last line of the screen, and moves by *n*-1 pages. If the optional *n* argument is not specified, it is assumed to be one.

4.10.10 PrevWord

Syntax: `PrevWord [n]`
Abbreviation: `PW`

moves the cursor on the previous word *n* times. If the optional *n* argument is not specified, it is assumed to be one (in which case, if the cursor is in the middle of a word the effect is just to move it on the start of that word).

4.10.11 NextWord

Syntax: `NextWord [n]`
Abbreviation: `NW`

moves the cursor on the next word *n* times. If the optional *n* argument is not specified, it is assumed to be one.

4.10.12 MoveEOL

Syntax: `MoveEOL`
Abbreviation: `EOL`

moves the cursor to the end of the current line.

4.10.13 MoveSOL

Syntax: `MoveSOL`
Abbreviation: `SOL`

moves the cursor to the start of the current line.

4.10.14 MoveEOF

Syntax: `MoveEOF`
Abbreviation: `EOF`

moves the cursor to the end of the document (`EOF` = end of file).

4.10.15 MoveSOF

Syntax: `MoveSOF`
Abbreviation: `SOF`

moves the cursor to the start of the document (`SOF` = start of file).

4.10.16 MoveEOW

Syntax: `MoveEOW`
Abbreviation: `EOW`

moves the cursor one character after the end of the word it is on.

This commands is extremely useful in macros, because it allows to copy precisely the word the cursor is on. See Chapter 6 [Hints and Tricks], page 45.

4.10.17 ToggleSEOF

Syntax: `ToggleSEOF`
Abbreviation: `TSEOF`

moves the cursor to the start of document, if it is not already there; otherwise, moves it to the end of the document.

This kind of toggling command (also see Section 4.10.18 [ToggleSEOL], page 39) is very useful in order to gain some keystroke on systems with very few keys. See also Section 4.10.15 [MoveSOF], page 38, and Section 4.10.14 [MoveEOF], page 38.

4.10.18 ToggleSEOL

Syntax: `ToggleSEOL`
Abbreviation: `TSEOL`

moves the cursor to the start of the current line, if it is not already there; otherwise, moves it to the end of the current line.

This kind of toggling command (also see Section 4.10.17 [ToggleSEOF], page 38) is very useful in order to gain some keystroke on systems with very few keys. See also Section 4.10.13 [MoveSOL], page 38, and Section 4.10.12 [MoveEOL], page 38.

4.10.19 SetBookmark

Syntax: `SetBookmark [`*n*`]`
Abbreviation: `SBM`

sets the *n*th bookmark to the current cursor position. If the optional *n* argument is not specified, it is assumed to be zero. There are ten bookmarks, numbered from 0 to 9.

4.10.20 GotoBookmark

Syntax: `GotoBookmark [`*n*`]`
Abbreviation: `GBM`

moves the cursor to the *n*th bookmark. If the optional *n* argument is not specified, it is assumed to be zero. There are ten bookmarks, numbered from 0 to 9.

4.11 Editing Commands

This commands allows to modify directly a document.

4.11.1 InsertChar

Syntax: `InsertChar` *code*
Abbreviation: `IC`

inserts a character whose ASCII code is *code* at the current cursor position. *code* has always to be different from 0. All the currently active options (insert, word wrapping, auto indent, *et cetera*) are applied.

Note that inserting a line feed (10) is completely different from inserting a line with `InsertLine`. `InsertChar 10` puts the control char *`Control-J`* in the text at the current cursor position. See Section 4.11.4 [InsertLine], page 40.

4.11.2 DeleteChar

Syntax: `DeleteChar [`*n*`]`
Abbreviation: `DC`

deletes *n* characters from the text. If the optional *n* argument is not specified, it is assumed to be one. Deleting a character when the cursor is just after the last char on a line will join a line with the following one; in other word, the carriage return between the two lines will be deleted. Note that if the cursor is after the end of the current line, no action will be performed.

4.11.3 Backspace

Syntax: `Backspace [`*n*`]`
Abbreviation: `BS`

acts like `DeleteChar`, but moves to the right the cursor before deleting each character.

4.11.4 InsertLine

Syntax: `InsertLine [n]`
Abbreviation: `IL`

inserts *n* lines at the current cursor position, breaking the current line. If the optional *n* argument is not specified, it is assumed to be one.

4.11.5 DeleteLine

Syntax: `DeleteLine [n]`
Abbreviation: `DL`

deletes *n* lines at the current currsor position, putting the last one in the temporary buffer, from which it can be undeleted. See Section 4.7.3 [UndelLine], page 30. If the optional *n* argument is not specified, it is assumed to be one. Note that this action is in no way inverse with respect to `InsertLine`.

4.11.6 DeleteEOL

Syntax: `DeleteEOL`
Abbreviation: `DE`

deletes all characters from the current cursor position to the end of the line.

This command could be easily implemented with a macro, but it is such a common, basic editing feature that it seemed worth a separate implementation.

4.12 Support Commands

These commands perform miscellaneous useful actions. In particular, they provide access to the shell and a way to assign the functionality of `Escape` to another key.

4.12.1 About

Syntax: `About`
Abbreviation: `About`

displays a simple information line about `ne` on the status bar.

4.12.2 Beep

Syntax: `Beep`
Abbreviation: `BE`

beeps. If your terminal cannot beep, it flashes. If it cannot flash, nothing happens (but you have a very bad terminal).

4.12.3 Exec

Syntax: `Exec`
Abbreviation: `EX`

prompts the user on the input line, asking for a command, and executes it. It is never registered while recording a macro (but the command you type is).

This command is mainly useful for key bindings, menu configurations, and in manually programmed macros.

Note that if the command you specify does not appear in `ne`'s internal tables, it is considered to be a macro name. See Section 4.6.3 [Macro], page 29.

4.12.4 Flash

Syntax: `Flash`
Abbreviation: `FL`

acts as `Beep`, but interchanging the words "beep" and "flash". Same comments apply. See Section 4.12.2 [Beep], page 40.

4.12.5 Help

Syntax: `Help [`*name*`]`
Abbreviation: `H`

displays some help about the command *name* (both the short and the long version are accepted). If no argument is given, a list of all existing commands in long form is displayed, allowing you to choose one. You can browse the help text with the standard navigation keys. If you press `Return`, the command list will be displayed again. If you press `f1` or `Escape`, you will return to normal editing.

This command is never registered while recording a macro, so that you can safely access the help system while recording. See Section 4.6.1 [Record], page 28.

4.12.6 NOP

Syntax: `NOP`
Abbreviation: `NOP`

does nothing. Mainly useful for inhibiting standard key bindings.

4.12.7 Refresh

Syntax: `Refresh`
Abbreviation: `REF`

refreshes the display. This command is very important, and should preferably be bound to the *`Control-L`* sequence, for historical reasons. It can always happen that a noisy phone line or a quirk in the terminal corrupts the display. This command restores it from scratch.

4.12.8 System

Syntax: `System [`*command*`]`
Abbreviation: `SYS`

asks the shell to execute *command*. The terminal is temporarily reset to the state it was in before `ne`'s activation, and *command* is started. When the execution is finished, `ne` returns in control.

If the optional argument *command* is not specified, you can enter it on the input line.

4.12.9 Escape

Syntax: `Escape`
Abbreviation: `ESC`

toggles on and off the menus, or escapes from the input line. This command is mainly useful for reprogramming the menu activator, and it is never registered while recording a macro. See Section 4.6.1 [Record], page 28.

5 Configuration

In this chapter we will see how the menus and the key bindings of `ne` can be completely configured. Note that the configuration is parsed at startup time, and cannot be changed during the execution of the program. This is a chosen limitation.

It should also be remarked that the standard configuration of `ne` does not contain key bindings relative to the `Meta` key. This choice was forced by the fact that the behaviour of this key is unpredictable on most systems. If your `Meta` key does what it should (i.e., it rises the high bit of any character), you can configure about thirty new shortcuts—the *`Control-Meta-letter`* combinations—which will produce ASCII characters between 128 and 159, and will be parsed as shortcuts by `ne`.

5.1 Key Bindings

`ne` allows you to associate to any keystroke any command. In order to accomplish this task, you have to create a file named `$HOME/.ne/.keys`.

The format of the file is very simple: each line starting with the '`KEY`' sequence of capital characters is considered the description of a key binding. All other lines are considered commments. The format of a key binding description is

```
KEY hexcode command
```

The *hexcode* value is the ASCII code of the keystroke. For special keys such as `Insert` or function keys, you should take a look at `ne`'s source file `keycodes.h`, which contains the codes for all `terminfo`'s key capabilities. You can write just the hex digits, nothing else is necessary (but a prefixing '`0x`' is tolerated). For instance,

```
KEY 1 MOVESOL
```

binds to *`Control-A`* the action of moving to the start of a line, while

```
KEY 101 LINEUP
```

binds to the "cursor-up" key the action of moving the cursor one line up.

The file `default.keys` which comes with `ne`'s distribution contains a complete, commented definition of `ne`'s standard bindings. You can modify this file with a trial-and-error approach.

command can be any `ne` command, including `Escape` (which allows the reconfigure the menu activator) and `Macro`, which allows to bind complex sequences of actions to a single keystroke. The binding of a macro is very fast because on the first call the macro is cached in memory. See Section 4.6.3 [Macro], page 29.

Note that you cannot *ever* redefine `Return` and `Escape`. This is a basic issue—however brain damaged is the current configuration, you will always be able to exploit fully the menus and the command line.

The key binding file is parsed at startup. If something does not work, `ne` exits displaying an error message. If you want to skip this phase (for instance, in order to correct the broken file), just give `ne` the `--noconfig` argument. See Section 3.1 [Arguments], page 9.

5.2 Changing Menus

When `ne` is started, it looks at the file `$HOME/.ne/.menus`; if it exists, it is considered a menu configuration file.

Each line of a menu configuration file not starting with the '`MENU`' or '`ITEM`' keywords is considered a comment. You should describe the menus as in the following example:

```
MENU "Project"
ITEM "Open...     ^O" Open
```

```
ITEM "Close         " Close
ITEM "DoIt          " Macro DoIt
```

In other words: a line of this form

```
MENU "title"
```

will start the definition of a new menu, having the given title. Each line of the form

```
ITEM "text" command
```

will then define a menu item, and associate the given command to it.

Any number of menus can be accomodated, but you should consider that many terminals are 80 column wide. There is also a minor restriction on the items—their width has to be constant throughout each menu (but different menus can have different widths). Note that the text of an item, as the name of a menu, is between quotes. Whatever follows the last quote is considered the command associated to the menu.

Warning: the description of key bindings in menus ('`^O`' in the previous example) is very important for the beginner; there is no relation inside `ne` about what you say in the menu and how you configure the key bindings (see Section 5.1 [Key Bindings], page 43). Please do not say things in the menus which are not true in the key binding file.

The menu configuration file is parsed at startup. If something does not work, `ne` exits displaying an error message. If you want to skip this phase (for instance, in order to correct the broken file), just give `ne` the `--noconfig` argument. See Section 3.1 [Arguments], page 9.

6 Hints and Tricks

Use `f1`*, not* `Escape`.

Due to the limitations of the techniques used when communicating with a terminal, it is not possible to "decide" that the user pressed the `Escape` key for about a second after the actual key press (see Section 4.9.9 [EscapeTime], page 34). This means that you will experiment annoying delays when using menus. If you have no `f1` key, redefine a keystroke assigning the command `Escape`, and you will be able to use that keystroke instead of `Escape`.

Check for the presence of a `Meta` *key.*

If your system has a standard `Meta` or `Alt` key, there is a good chance that you have another thirty shortcuts or so. See Chapter 5 [Configuration], page 43.

`ne` *does tilda expansion.*

When you have to specify a file name, you can always start with `~/` in order to specify your home directory, or `~user/` in order to specify the home directory of another user.

The `Escape` *delay when activating menus can be avoided.*

If you press after `Escape` any key which does not produce the second character of an escape sequence, `ne` will immediately recognize the `Escape` key code as such. Since non-alphabetical keys have no effect while browsing through the menus, if you're forced to use `Escape` as menu activator you can press, for instance, '`,`' just after it in order to speed up the menu activation (note that '`:`' would not work, because it would activate the command line).

Use turbo mode for lengthy operations.

Turbo mode (see Section 4.9.11 [Turbo], page 35) allows to perform very complex operation without updating the screen up to the end. This can be a major plus if you are editing very long files, or if your terminal is slow. If the default value (twice the number of lines of the screen) does not give you the best results, experiment other values.

Regular expressions are powerful, and slow.

Regular expressions must be studied very carefully. If you spend a lot of time doing editing, it is definitely reasonable to study even their most esoteric features. Very complex editing actions can be performed by a single find/replace using the `\n` convention. But remember always that regular expressions are much slower than a normal search.

Use the right movement commands in a macro.

Many boring, repetitive editing actions can be performed in a breeze by recording them the first time. Remember, however, that while recording a complex macro you should always use a cursor movement that will apply in a different context. For instance, if you are copying a word, you cannot move with cursor keys, because that word at another application of the macro could be of a different length. Rather, use the next/previous word keys and the `MoveEOW` command, which guarantee a correct behaviour in all situations.

Some preferences can be preserved even with automatic preferences.

When you save an autoprefs file, the file simply contains a macro which, when executed, produces the current configuration. However, you could want, for instance, to never change the insert/overwrite state. In this case, just edit with `ne` the autoprefs files and delete the line containing the command setting the insert flag. When

the autoprefs will be loaded, the insert flag will be left untouched. This trick is particularly useful with the `StatusBar` and `FastGUI` commands.

If some keystrokes do not work, check for system-specific features.

Sometimes it can happen that a keystroke does not work—for instance, *`Control-O`* does not open a file. This usually is due to the kernel tracking that key for its purposes. For instance, along a `telnet` connection with xon/xoff flow control, *`Control-S`* and *`Control-Q`* would block and release the output instead of saving and quitting.

In these cases, if you do not need the system feature you should check how to disable it: for instance, some BSD-like systems feature a delayed suspend signal which is not in the POSIX standard, and thus cannot be disabled by `ne`. On HP-UX, the command '`stty dsusp ^-`' would disable the signal, and would let the control sequence previously assigned to it to run up to `ne`.

Another example is the `NCSA Telnet` software for the Macintrash. Unless you modify your setting in such a way to disable *`Control-S`* and *`Control-Q`* as flow controllers, you will not be able to use them as keystrokes (even if `ne` is doing all it can in order to explain to the software that it does *not* need xon/xoff flow control. . .).

7 Motivations and Design

In this chapter I will try to outline the rationale behind `ne`'s design choices. Moreover, some present, voluntary limitations of the current implementation will be described. The intended audience of such a description is the programmer wanting to hack up `ne`'s sources, or the informed user wanting to deepen its knowledge of the limitations.

`ne` has no concept of *mode*. All shortcuts are defined by a single key, possibly with a modifier (such as `Control` or `Meta`). Modality is in my opinion a Bad Thing unless it has a very clear visual feedback. As an example, menus are a form of modality. After entering the menus, the alphabetic keys and the navigation keys have a different meaning. But the modality is clearly reflected by a change in the graphical user interface. The same can be said about the input line, because it is always preceeded by a (possibly highlighted) prompt ending with a colon.

`ne` has no sophisticated visual updating system similar, for instance, to the one of `curses`. All updating is done while manipulating the text, and only if the turbo flag is set some iterated operations can delay the update (in this case, `ne` keeps track in a very rough way of the part of the screen which changed). Moreover, the output is not preempted by additional input coming in, so that along a slow connection the output could not keep up with the input. However, along fast connections, the responsiveness of the editor is greatly enhanced by the direct update. Moreover, a great deal of memory and computational power is gained, because it is not necessary to keep constantly updated two copies of the screen, and to compare them whenever doing an update. As it is typical in `ne`, when such design tradeoffs arise preference is given to the solution which is effective on a good part of the existing hardware, and will be very effective on most future hardware.

`ne` uses a particular scheme for handling the text. There is a doubly linked list of line descriptors which contain pointers to each line of text. The lines themselves are kept in a list of pools, which is expanded and reduced dynamically. The interesting thing is that for each pool `ne` keeps track just of the first and of the last character used. A character is free iff it containes a null, so there is no need for a list of free chunks. The point is that the free characters laying between that first and the last used characters (the *lost* characters) can only be allocated *locally*: whenever a line has to grow in length, `ne` first checks if there are enough free characters around it. Otherwise, it remaps the line elsewhere. Since editing is essentially a local activity, the number of such lost characters remains very low. And the manipulation of a line is extremely fast and independent of the size of the file, which can be very huge. A mathematical analysis of the space/time tradeoff is rather difficult, but empirical evidence suggests that the idea works.

`ne` takes the POSIX standard as the basis for UN*X compatibility. The fact that this standard has been designed by a worldwide recognized and impartial organization such as IEEE makes it in my opinion the most interesting effort in his league. No attempt is made of supporting ten thousands different versions and releases by using conditional compilation. Very few assumptions are made about the behaviour of the system calls. This has obvious advantages in terms of code testing, maintenance, and reliability. For the same reasons, the availability of an ANSI C compiler is assumed.

If the system has a `terminfo` database and the relative functions (which are usually contained in the library `libcurses.a`), `ne` will use them. The need for a terminal capability database is clear, and the choice of `terminfo` (with respect to `termcap`) is compulsory if you want to support a series of features (such as more than ten function keys) which `termcap` lacks. If `terminfo` is not available, `ne` can use a `termcap` database. Some details about this can be found in Chapter 9 [Portability Problems], page 51.

`ne` does not allow to redefine the `Escape` and `Return` keys, and the interrupt character *`Control`*`-\`. This decision has been taken mainly for two reasons. First of all, it is necessary to keep a user from transforming `ne`'s bindings to such a point that another unaware user

cannot work with it. These two keys and the alphabetic keys allow to activate any command without any further knowledge of the key bindings, so it seems to me this is a good choice. As a second point, the `Escape` key usage should generally be avoided. The reason is that most escape sequences that are produced by special keys start exactly with the escape character. When `Escape` is pressed, `ne` has to wait for one second (this timing can be changed with the `EscapeTime` command), just to be sure that it did not receive the first character of an escape sequence. This makes the response of the key very slow, unless it is immediately followed by another key such as '`:`'. See Chapter 6 [Hints and Tricks], page 45.

Note that it was stated several times that the custom key bindings work also when doing a long input, navigating through the menus or browsing the requester. This is only partially true. In order to keep down the code size and complexity, in these cases `ne` recognizes only direct bindings to commands, and discards the arguments. Thus, for instance, if a key is bound to the command line '`LineUp 2`', it will act like '`LineUp`', while a binding to '`Macro MoveItUp`' would produce no result. Of course full binding capability is available while writing text. This limitation will probably be lifted in a future version: presently it does not seem to limit seriously the configurability of `ne`.

`ne` has some restriction in its terminal handling. It does not support highlighting on terminals which use a magic cookie. Supporting correctly such terminals is a royal pain, and I did not have any means of testing the code anyway. Moreover, they are rather obsolete. Another lack of support is for the capability strings which specify a file to print or a program to launch in order to initialize the terminal.

The macro capabilities of `ne` are rather limited. For instance, you cannot give an argument to a macro: they are simply scripts which can be played back automatically. This makes them very useful for everyday usage in a learn/play context, but rather unflexible for extending the capabilities of the program. However, it is not reasonable to incorporate in an editor an interpreter for a custom language. Rather, a systemwide macro language should control the editor *via* interprocess communication. This is the way of the REXX language: unfortunately, a diffused, uniform, standard implementation of REXX under UN*X is not likely to appear. However, the next version of `ne` will certainly feature a REXX port on the Amiga.

`ne` has been written with sparing resource usage as a basic goal. Every possible effort has been made in order to reduce the use of CPU time and memory, and the number of system calls. For instance, command parsing is done through hash techniques, and the escape sequence analysis uses the order structure of strings for minimizing the number of comparisons. The optimal cursor motion functions were directly copied from `emacs`. No busy polling is allowed. Doubly headed, doubly linked lists allow for very fast list operations without any special case whatsoever. The search algorithm is a version of the Boyer-Moore algorithm which provides high performance with a minimal setup time. An effort has been taken to move to the text segment all data which do not change during the program execution.

A word should be spent about lists. Clearly, handling the text as a single block with an insertion gap (a la `emacs`) allows you to gain some memory, since you do not have to allocate the list nodes, which require usually 16 bytes per line. However, the management of the text as a linked list requires much less CPU time, and the tradeoff seems to be particularly favorable on virtual memory systems, where moving the insertion gap can require a lot of accesses to different pages.

Just to give a pratical example, on the HP-UX systems where `ne` was developed `vi` requires more memory than `ne`, unless the size of the file to edit is rather big, in which case `ne` requires a data segment about 20% bigger. (Of course, this does not take into account some sophisticated features of `ne`, such as unlimited undo/redo, which can cause a major memory consumption.)

8 Some Notes for the Amiga User

This section describes the differences between the Amiga and UN*X versions of `ne`, and some of the misfeatures inherited by its UN*Xish design.

In order to keep maintenance of the code simple, conditional code was avoided when possible. Thus, some feature had to be dropped. First of all, there is no interrupt character. This happens because the Amiga handles signals in a very different way than UN*X, and it would have been very complex to reproduce the original behaviour.

In the file requester, it is not possible to obtain a list of the available devices. Indeed, it is not even possible to pass from a device to another inside the requester. You have to escape, then input manually the device name as a filename (which will produce a spurious error) and open again: this time, the device scanned by the requester will be the new one. Another alternative, of course, is simply to input a complete pathname.

`ne` will not behave particularly well under low memory conditions. It won't crash, but it could behave improperly.

The `$HOME` directory has no meaning on the Amiga: rather, the `PROGDIR:` directory is used. For instance, the `$HOME/.ne` directory is really `PROGDIR:.ne`.

9 Portability Problems

This chapter is devoted to the description of the (hopefully very few) problems that could arise when porting `ne` to another version of UN*X. Compatibility within the Amiga family of computers is complete, so there are no problems in recompiling `ne` for a different processor or architecture in this case.

The fact that only POSIX calls have been used (see Chapter 7 [Motivations and Design], page 47) should guarantee that on POSIX-compliant systems a recompilation should suffice. Unfortunately, `terminfo` has not been standardized by IEEE, so that different calls could be available. The necessary calls are `setupterm()`, `fixterm()`, `resetterm()`, `tparm()` and `tputs()`. The other `terminfo` functions are never used.

If `terminfo` is not available, the source files `info2cap.c` and `info2cap.h` map `terminfo` calls on `termcap` calls. The complete GNU `termcap` sources are distributed with `ne`, so no library at all is needed in order to use them. You just have to compile using as makefile `Makefile.termcap`. Should you need comprehensive information on GNU `termcap`, you can find the distribution files on any `ftp` site which distributes the GNU archives. I should note that the GNU `termcap` manual is definitely the best manual ever written about terminal databases.

There are, however, some details which are not specified by POSIX, or are specified with insufficient precision. The places of the source where such details come to the light are evidenced by the '`PORTABILITY PROBLEM`' string, which is followed by a complete explanation of problem.

For instance, there is no standard way of printing extended ASCII characters (i.e., characters whose code is smaller than 32 or greater than 126). On many system, these characters have to be filtered and replaced with something printable: the default behaviour is to add 64 to all characters under 32 (so that control characters will translate to the respective letter) and to visualize all characters between 126 and 160 as a question mark (this works particularly well with ISO Latin 1). If your system has a more powerful display, you may want to change the `DECONTROL()` macro defined in `term.c` which takes a character variable as an argument, and transforms it into a printable character.

Note that it is perfectly possible that some system features not standardized by POSIX interfere with `ne`'s usage of the I/O stream. Such problems should be attached locally, by using the system facilities, rather than by `#ifdef`'ing horribly the source code. An example is given in Chapter 6 [Hints and Tricks], page 45.

10 Acknowledgments

A lot of people contributed to this project. Part of the code comes from the `emacs` sources. Many people, in particular at the silab (the Milan University Computer Science Department Laboratory), helped in beta testing.

In particular, I would like to cite Roberto Attias, Ivan Buttinoni, Alfredo Chizzoni, Marco Colombo, Heinfried Korn, Willy Langeveld, Fabrizio Lodi, Antonio Piccolboni, Gianpiero Puccioni, Marco Pugliese, Marco Rodolfi, Larry Rosenman, Sergio Ruocco, David Alan Sanna, Carlo Santagostino, Markus Senoner, Paolo Silvera, Reinhard Spisser, Elena Toninelli and Marvin Weinstein.

Comments, complaints, desiderata are welcome.

```
Sebastiano Vigna
Via California 22
I-20144 Milano MI

BIX: svigna@bix.com
INTERNET: vigna@ghost.dsi.unimi.it
UUCP:seba@sebamiga.adsp.sub.org
```

Command Index

R

S

T

U

V

W

Concept Index

T

U

V

W

Table of Contents