ne

A nice editor
Version 2.5

by Sebastiano Vigna and Todd M. Lewis

1 Introduction

`ne` is a full screen text editor for UN*X (or, more precisely, for POSIX: see Chapter 7 [Motivations and Design], page 61). I came to the decision to write 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.

A concise overview of the main features follows:

- three user interfaces: control keystrokes, command line, and menus; keystrokes and menus are completely configurable;
- syntax highlighting;
- full support for UTF-8 files, including multiple-column characters;
- 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 record/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;
- automatic completion of prefixes using words in your documents as dictionary;
- a file requester with completion features for easy file retrieval;
- extended regular expression search and replace à la `emacs` and `vi`;
- a very compact memory model—you can easily load and modify very large files;
- 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 prothethhing lihtth.

2 Basics

Simple things should be simple. 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 *real* editing is done without ever touching a mouse, it is also true that editing should be doable without ever touching a manual. These two conflicting goals can be accommodated easily in a single program if we can offer a series of interfaces that allow for differentiated use.

In other words, it is unlikely that an `ne` wizard will ever have to activate a menu, but 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 esoteric features, but few 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. The average user should be able to record his actions, replay them, and save them in a humanly readable format for further use and editing.

In the following sections we shall take a quick tour of `ne`'s features.

2.1 Terminology

In this section we explain and contrast some of the terms `ne` uses. Understanding these distinctions will go a long way towards making the rest of this manual make sense.

A *file* is a group of bytes stored on disk. This may seem rather obvious, but the important distinction here is that `ne` does not edit files; it edits *documents*.

A *document* is what `ne` calls one of the "text thingies" that you can edit. It is a sequence of lines of text in the computer's memory—not on disk. Documents can be created, edited, saved in files, loaded from files, discarded, *et cetera*. When a document is loaded from or saved to a file, it remains associated with that file by name until the document is either closed or saved to a different file. Interactions between documents and files are handled by the commands under the '`File`' menu. The '`Documents`' menu commands only deal with documents. See Section 3.7 [Menus], page 16.

Internally, `ne` holds its documents in *buffers*. A *buffer* is a chunk of memory in which `ne` holds something. For example, each document is held in its own buffer, as are any loaded or recorded macros, undo records, a copy of your last deleted line of text, a copy of all your previous responses to long input, and several other things.

2.2 Starting

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. The screen of your terminal will be cleared (or filled with text loaded from the first file you specified). See Section 3.1 [Arguments], page 11 for other command line options.

Writing text is pretty straightforward: if your 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 to perform corrections. If your keyboard has an INSERT key, you can use it to *toggle* (switch from on to off, or vice versa) insert mode. In general, `ne` tries to squeeze everything it can from your keyboard. Function keys and special movement keys should work flawlessly if your terminal is properly configured. If not, complain to your system administrator. If that doesn't help, see Section 5.1 [Key Bindings], page 55.

At the bottom of the screen, you will see a line containing some numbers and letters. This is called the *status bar* because it reports to you part of the internal state of the editor. At startup, the status bar has the following form:

```
L:       1 C:       1  12% ia----pvu-t------@A <unnamed>
```

(the numbers could be different, and a file name could be shown as last item instead of '<unnamed>'). You probably already guessed that the numbers after 'L:' and 'C:' are your cursor's line and column numbers, respectively, whereas the percentage indicates approximately your position in the file. The small letters represent user flags that you can turn on and off. In particular, 'i' tells you that insert mode is on, while 'p' tells that the automatic preferences system is activated. For a thorough explanation of the meaning of the flags on the status bar, see Section 3.2 [The Status Bar], page 12.

Once you are accustomed to cursor movement and line editing, it is time to press 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. (If you find yourself waiting for the menu to appear, you can press ESCAPE twice in a row.) You can now move around menus and menu items by pressing the cursor keys. Moreover, a lower case alphabetic key will move to the next item in the current menu whose name starts with that letter, and an upper case alphabetic key will move to the next 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 'File' menu. Menus are fully discussed in Section 3.7 [Menus], page 16. When you want to exit from the menu system, press F1 (or ESCAPE) again. If instead you prefer to choose a command and execute it, move to 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 (we assume here that 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. Finally, the symbol '[' in front of a character denotes the shortcut produced by CONTROL plus META (a.k.a. ALT) plus that character, *or* META plus that character, depending on your terminal emulator—you must check by yourself. Moreover, these last bindings could not work with some terminals, in which case you can replace them with a sequence: just press the ESCAPE key followed by the letter. A few menu items are bound to two control sequences (just in case one does not work, or it is impractical).

Note that under certain conditions (for instance, while using ne through a telnet connection) some of the shortcuts might not work because they are trapped by the operating system for other purposes (see Chapter 6 [Hints and Tricks], page 59).

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 to enter a command to execute. Just press RETURN for the time being (or, if you are really interested in this topic, see Section 3.4 [The Command Line], page 14).

In the sections that follow, when explaining how to use a command we shall usually describe the corresponding menu item. The related shortcut and command can be found on the menu item itself, and in Section 3.7 [Menus], page 16.

2.3 Loading and Saving

The first thing to learn about an editor is how to exit. ne has a CloseDoc command that 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 modifications. (You will be requested to confirm your choice in case the current document has been modified since the last save.)

There is also a Quit command, which closes all the documents without saving any modifications, and a Save&Exit (*META-X*) 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 may use the Open command, which can be activated by pressing *CONTROL-O*, by choosing the 'Open...' item of the 'File' menu, or by typing it on the command line (as in the previous case). You will be prompted with a list of files and directories in the current working directory. (You can tell the directory names because they end with a slash; they will also appear in a bold face if your terminal allows it.) You can select any of the file names by using the cursor keys, or any other movement key. Pressing an alphabetic key will move the cursor to the first entry after the cursor that starts with the given letter. When the cursor is positioned over the file you want to open, press RETURN, and the file will be opened. If instead you move to a directory name, pressing RETURN will display the contents of that directory.

You can also escape with F1, ESCAPE or ESCAPE-ESCAPE and manually type the file name on the command line (or escape again, and abort the Open operation). If you escape with TAB instead, the file or directory under the cursor will be copied in the input line, where you can modify it manually. ne has also file name completion features activated by TAB (see Section 3.3 [The Input Line], page 13).

When you want to save a file, just use the command Save (*CONTROL-S*). It will use the current document name or will ask you for one if the current document has no name. SaveAs, on the other hand, 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 in some emergency files. These files will have names similar to your current files, but they will have a pound sign '#' prefixed to their names. See Section 3.10 [Emergency Save], page 24.

2.4 Editing

An editor is presumably used for editing text. If you decide not to edit text, you probably don't want to use ne, because that's all it does—it edits text. It does not play Tetris. It does not evaluate recursive functions. It does not solve your love problems. It just allows you to edit text.

The design of ne makes editing extremely natural and straightforward. There is nothing special you have to do to start editing once you've started ne. Just start typing, and the text you type shows up in your document.

ne provides two ways of deleting characters, the BACKSPACE (or *CONTROL-H*, if you have no such key) and the DELETE key. In the former case you delete the character to the left of the cursor, while in the latter case 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 Section 4.11.4 [DeleteChar], page 51 and Section 4.11.7 [Backspace], page 52.)

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 Section 4.11.9 [DeleteLine], page 52 and Section 4.7.3 [UndelLine], page 35.)

If you want to copy, cut, paste, shift 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* (think "block"). This command sets the mark at the current cursor position. 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, which allows you to mark rectangles of text. Whenever a mark has been set, either an 'M' appears on the command line or a 'V' appears 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 to move the cursor to 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 a file into a clip with 'Open Clip...', and paste it anywhere. All such operations 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 4.4.11 [ClipNumber], page 30.

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 of ne is it's ability to load an unlimited number of documents. If you activate the NewDoc command (using the 'Document' menu 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 (F4) prompts you with the list of names of currently loaded documents and allows you to choose directly what to edit.

2.5 Basic Preferences

ne has a number of *flags* that specify alternative behaviors, the most prototypical example being the *insert* flag, which specifies whether the text you type is inserted into 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; see Section 4.9.4 [Insert], page 39.)

Another important flag is the *free form* flag, which specifies whether the cursor can be moved beyond the right end of each line of text or only to existing text (a la vi). Programmers usually prefer non free form editing; text writers seem to prefer free form. See Section 4.9.6 [FreeForm], page 40 for some elaboration. The free form flag can be set with the 'Free Form' menu item of the 'Prefs' menu.

At this point, we suggest you explore by trial and error the other flags of the 'Prefs' menu, or try the Flags command (see Section 4.9.1 [Flags], page 38), which explains all the flags and the commands that operate on them. We prefer spending a few words discussing *automatic preferences* or *autoprefs*.

Having many flags ensures a high degree of flexibility, but it can turn editing into a nightmare if you have to turn on and off dozens of flags for each different kind of file you edit. ne's solution is to automatically set a document's flags when a file is loaded based on your stated preferences for each *file type*. A file's type is determined by the *extension* of its file name, that is, the last group of letters after the last dot. For instance, the extension of 'ne.texinfo' is 'texinfo', the extension of 'source.c' is 'c', and the extension of 'my.txt' is 'txt'.

Whenever you select the 'Save AutoPrefs' menu item, ne saves the flags of your current document to be used when you load other files with the same extension as your current document. These *autoprefs* are saved in a file in your '~/.ne' directory. This file has the same name as the extension of the current document with '#ap' appended to it. It contains all the commands necessary to recreate your current document's flag settings. Whenever you open a file with this file name extension, ne will automagically recreate your preferred flag settings for that file type. (There is a flag that inhibits the process; see Section 4.9.2 [AutoPrefs], page 39.)

Finally, when you select the 'Save Def Prefs' menu item, a special preferences file named '.default#ap' is saved. These preferences are loaded whenever ne is run before loading any file. This is how you set up the preferences you always want to be set.

A small set of preferences are global to ne rather than specific to particular document types. Thus they are saved in the '.default#ap' file by the SaveDefPrefs command or the 'Save Def Prefs' menu. They are not saved by the SaveAutoPrefs command. These preferences are: FastGUI, RequestOrder, StatusBar and VerboseMacros; see Section 4.9.5 [FastGUI], page 39, See Section 4.9.8 [RequestOrder], page 40, See Section 4.9.9 [StatusBar], page 40, and See Section 4.9.18 [VerboseMacros], page 42.

Similarly, the current syntax definition is specific to the current document type, so it is saved only in autoprefs files by the `SaveAutoPrefs` command or '`Save AutoPrefs`' menu; it is not saved in the '`.default#ap`' file.

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

2.6 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. This is where *macros* come in.

A *macro* is a stored sequence of commands. Any sequence of commands you find yourself repeating is an excellent candidate for being made into a macro. You could create a macro by editing a document that only contains valid `ne` commands and saving it, but by far the easiest way to create a macro is to have `ne` record your actions. `ne` allows you to record macros and then play them (execute the commands they contain) many times. You can save them on disk for future use, edit them, or bind them to any key. You could even reconfigure each key of your keyboard to play a complex macro if you wanted to.

`ne` can have any number of named macros loaded at the same time. It can also have one unnamed macro in its *current macro* buffer. The named macros are typically loaded from disk files, while the current macro buffer is where your recorded macro is held before you save it or record over it.

Recording a macro is very simple. The keystroke *CONTROL-T* starts and stops recording a macro. When you start recording a macro, `ne` clears the *current macro* buffer and starts recording all your actions (with a few exceptions). You can see that you are recording a macro if an '`R`' appears on the status bar. When you stop the recording process (again using *CONTROL-T*), 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 macro's execution with *CONTROL-*.

A recorded macro has no name. It's just an anonymous sequence of commands in the *current macro* buffer, and it will go away when you exit `ne` or record another macro. If you want to save your recorded macro for future use, you can give it a name and save it with the '`Save Macro...`' menu item or the `SaveMacro` command. The macro is saved as a file in your current directory by default or whatever directory you specify when prompted for the macro's name. If you save it in your '`~/.ne`' directory then it will be easy to access it later from any other directory. The '`Open Macro...`' menu item and the `OpenMacro` command load a macro from a file into the current macro buffer just as if you just `Record`ed it.

Any macro can be loaded from a file and played with the '`Play Macro...`' menu item or the `Macro` command. (This won't modify any recorded anonymous macro that may be in the *current macro* buffer; `OpenMacro` does that.) Useful macros can be permanently bound to a keystroke as explained in Section 5.1 [Key Bindings], page 55. Moreover, whenever a command line does not specify one of `ne`'s built in commands, it is assumed to specify the name of a macro to execute. Thus, you can execute macros just by typing their file names. Include a path if the macro file's directory is different from your current directory or your '`~/.ne`' directory.

If the first attempt to open a macro fails, `ne` checks for a macro with the given name in your '`~/.ne`' directory. This allows you to program simple extensions to `ne`'s language. For instance, all automatic preferences macros—which are just specially named macros that contain only commands to set preferences flags—can be executed just by typing their names. For example, if you have an automatic preference for the '`doc`' extension for example, you can set `ne`'s flags exactly as if you loaded a file ending with '`.doc`' by typing the command `doc#ap`.

In general, it is a good idea to save frequently used macros in '`~/.ne`' so that you can invoke them by name without specifying a path regardless of your current directory. On the other hand, if you have a macro that is customized for one document or a set of documents that you store in one directory, then

you might want to save the macro in that directory as well. If you do, then you would want to cd to that directory before you start ne so that you can access that macro without specifying a path.

If your macro has the same name as one of ne's built-in commands, you can only access it with the Macro *name* command. Built-in command names are always found first before ne command interpreter looks for macros.

The system administrator may make some macros available from ne's global directory. See Section 3.1 [Arguments], page 11.

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. Subsequent invocations of the macro will used the cached version.

Warning: while path and file names are case sensitive when initially loading macros, loaded macro names are *not* case sensitive or path sensitive. ne only caches the file name of an already loaded macro, not the path name, and uses a case insensitive comparison. That is, if you invoke '~/foobar/MyMacro', ne remembers it with the case-insensitive name 'mymacro'; a subsequent call for '/usr/MYMACRO' will instead find and use the cached version of '~/foobar/MyMacro'. You can clear the cache by using the UnloadMacros command. See Section 4.6.6 [UnloadMacros], page 35.

The behaviour of macros may vary with different preferences. If the user changes the AutoIndent and WordWrap flags, for example, new lines and new text may not appear in the same way they would have when a macro was recorded. Good general purpose macros avoid such problems by using the PushPrefs command first. This preserves the user's preferences. Then they set any preferences that could affect their behaviour. Once that is taken care of they get on with the actual work for which they were intended. Finally, they use the PopPrefs command to restore the user's preferences. Note that if a macro is stopped before it restores the preferences (either by the user pressing *CONTROL*-\ or by a command failing) then that responsibility falls on the user.

Any changes made to a document by a macro are recorded just as if you had entered the commands yourself. Therefore you can use the Undo command to roll back those changes one at a time. This can be useful especially when developing macros, but you may want to be able to undo all the changes made by a macro with a single Undo command. The AtomicUndo command makes this possible. If you add AtomicUndo + at the start of your macro and AtomicUndo - at the end, then the Undo and Redo commands will handle all changes made by your macro atomically, i.e., as if they had been made by a single command. See Section 4.7.5 [AtomicUndo], page 36.

Finally, any line in a macro that starts with a non-alphabetical character is considered a comment, so you can add comments to a macro by starting a line with '#'.

2.7 More Advanced Features

2.7.1 UTF-8 support

UTF-8 is a character encoding that can represent the whole ISO 10646 character set—two billion characters! ne can load and manipulate UTF-8 files transparently, in particular on systems that provide UTF-8 I/O. See Section 3.11 [UTF-8 Support], page 24.

2.7.2 Bookmarks

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 up to ten bookmarks per document, each designated by a single digit, with the default being '0'. You can set them with the SetBookmark command, and you can return to any set bookmark with the GotoBookmark command. Also, ne sets an automatic bookmark (designated by '-') to your current position in a document whenever you use the GotoBookmark command. You can use this automatic bookmark to return to that previous location with a GotoBookmark - command. Doing so will reset the automatic bookmark, so that subsequent GotoBookmark - commands will switch between those two locations. The special parame-

ters '+1' and '-1' indicate the next or previous set bookmark in conjunction with GotoBookmark and UnsetBookmark, but reference the next or previous unset bookmark when used with SetBookmark. A sequence of GotoBookmark +1 commands lets you easily cycle through all your set bookmarks. See Section 4.10.26 [SetBookmark], page 50, Section 4.10.27 [GotoBookmark], page 50, and Section 4.10.28 [UnsetBookmark], page 50. 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 55.

2.7.3 Automatic Completion

The AutoComplete command helps you extend a given prefix with matching words from your open documents. You can specify the AutoCompete command and prefix on the command line, or you can enter the prefix directly into your document and activate the AutoComplete command. With the cursor at the right end of your prefix, activate the AutoComplete command by entering either the ESCAPE-TAB or the ESCAPE-I key sequence, or the *CONTROL-META-I* key combination, or by selecting AutoComplete from the Extras menu.

If the prefix can be extended unambiguously, the extension will be immediately inserted into your document (this is the case, for instance, if only one word matches the prefix), and a message will tell you whether the extension is an actual word or just the longest possible extension (for instance, if you expand 'fo' and your document contains 'foobar' and 'foofoo' then the partial match will be 'foo'). Otherwise, ne presents you with a list of all matching words: choose the one you want and press RETURN, to select it; otherwise, press F1, ESCAPE or ESCAPE-ESCAPE to cancel the completion operation.

The current state of the CaseSearch flag determines whether the prefix match is case sensitive. Any matching words which only exist in other open documents but not the current one are displayed in bold with an asterisk; think of that as a warning that if you select one of these bold words you will introduce a new word into your current document. Plain words already exist somewhere in your current document. See Section 4.5.11 [AutoComplete], page 33, and Section 4.5.10 [CaseSearch], page 32.

2.7.4 Automatic Bracket Matching

Unless you tell it not to (with the AutoMatchBracket command), ne will highlight any recognized bracket that matches the bracket your cursor is on if that matching bracket is currently visible on your screen. Recognized brackets are '{}', '()', '[]' and '<>'. See Section 4.5.8 [AutoMatchBracket], page 32.

2.7.5 MS-DOS files

ne will detect automagically the presence of MS-DOS line terminators (CR/LFs) and set the CR/LF flag. When the file will be saved, the terminators will be restored correctly. You can change this behaviour using the PreserveCR and CRLF commands. See Section 4.9.19 [PreserveCR], page 43, and Section 4.9.20 [CRLF], page 43.

2.7.6 Binary files

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 the effect of inserting or deleting a NULL. Be careful not to mismatch the state of the binary flag when loading and saving the same file.

2.7.7 File requester

The NoFileReq command deactivates the file requester. It is intended for "tough guys" who always remember the names of their files and can type them at the speed of light (maybe with the help of the completer, which is activated by the TAB key; see Section 3.3 [The Input Line], page 13).

2.7.8 Executing UN*X commands

There are three ways to execute UN*X commands from within ne. The System command can run any UN*X command; you will get back into ne as soon as the command execution terminates. See Section 4.12.10 [System], page 54. The Through (*META-T*) 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's output at the current cursor position. This provides a neat way to pass a part of your document through one of UN*X's many *filter commands* (commands that read from standard input and write to standard output, e.g., sort). See Section 4.4.12 [Through], page 30. Finally, you can use the Suspend (*CONTROL-Z*) command to temporarily stop ne and return to your command shell. See Section 4.12.9 [Suspend], page 54.

2.7.9 Advanced key bindings

For an exhaustive list of the remaining features of ne, see Chapter 3 [Reference], page 11.

3 Reference

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

3.1 Arguments

The main arguments you can give to `ne` are the names of files you want to edit. They will be loaded into separate documents. If you specify `--help` anywhere on the command line, a simple help text describing `ne`'s arguments will be printed.

The `+`*N* option causes `ne` to advance to the *N*th line of the next document loaded. This option is fairly common among editors and text display programs like `vi` and `less`. The *N* itself is optional. Without it, a bare `+` on the command line causes `ne` to advance to the last line of the first document. You can specify a line and column as `+N,M`. Any non-digit can be used to separate the *N* from the *M*. As it only affects the next document loaded, it can appear multiple times on the command line.

The `--binary` option causes `ne` to load the next document in binary mode. Binary mode treats the normal line termination characters as any other character and only breaks lines on NULL characters. Like `+N,M`, `--binary` only affects the next document loaded, and it can appear multiple times on the command line. See Section 4.9.3 [Binary], page 39.

The `--no-config` option skips the reading of the key bindings and menu configuration files (see Chapter 5 [Configuration], page 55). This is essential if you are experimenting with a new configuration and you make mistakes in it.

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

The `--keys` *`filename`* option and the `--menus` *`filename`* option specify a name different from the default one ('`.keys`' and '`.menus`', respectively) for the key bindings and the menu configuration files. Note that `ne` searches for these files first in the current directory, and then in your '`~/.ne`' directory.

The `--ansi` and the `--no-ansi` options manage `ne`'s built-in ANSI sequences. Usually `ne` tries to retrieve from your system some information that is necessary to handle your terminal. If for some reason this is impossible, you can ask `ne` to use a built-in set of sequences that will work on many terminals using the `--ansi` option (to be true, `ne` can be even compiled so that it uses directly the built-in set, but you need not know this). If you want to be sure (usually for debugging purposes) that `ne` is not using the built-in set, you can specify `--no-ansi`.

The `--no-syntax` option disables `ne`'s normal syntax highlighting capability. For most editing situations, this would be unnecessary, but for extremely large files it may be helpful. Syntax highlighting incurs small memory usage and processor overhead penalties for each line of text. The `--no-syntax` option eliminates that overhead.

The `--utf8` and `--no-utf8` options can be used to force or inhibit UTF-8 I/O, overriding the choice imposed by the system locale. Note, however, that in general it is more advisable to set the `LANG` environment variable to a locale supporting UTF-8 (you can usually see the locale list with `locale -a`). See Section 3.11 [UTF-8 Support], page 24.

If you need to open a file whose name starts with '`--`', you can put '`--`' before the filename, which will skip command recognition for the next word.

Finally, `ne` has a *global directory* where the system administrator can store macros, default preferences, and syntax definitions for all users of the system. The location of this directory is defined when `ne` is built, but you can override it by creating and exporting the `NE_GLOBAL_DIR` environment variable prior to invoking `ne`. If you load no files when you start `ne`, or if you invoke the `About` command, it will display a splash screen. The last line on that screen shows the global directory `ne` is using, if it exists, or an error message otherwise.

3.2 The Status Bar

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

The status bar looks more or less like this:

```
L:      31  C:      25   12% iabcwfpvurtBMRPC*@8       20 /foo/bar
```

The numbers after 'L:' and 'C:' are the line and column of the cursor position. The first line and the first column are both number 1. Then, ne shows the percentage of lines before the current line (it will be 0% on the first line, and 100% on the last line).

Following that are a sequence of letters or dashes. These indicate the status of a series of flags which we shall look at later.

The hexadecimal digits following the flags give the code for the character at the cursor, and are displayed optionally (see Section 4.9.10 [HexCode], page 41). If your cursor is at or beyond the right end of the current line, the code disappears.

The file name appearing after the character code is the file name of the current document. The left end of very long file names may be truncated to keep the right end visible. Of course, ne is keeping track internally of the complete file name. It is used by the Save command and as the default input for the SaveAs command. See Section 4.2.3 [Save], page 26, and Section 4.2.4 [SaveAs], page 26.

The displayed line and column numbers, the percentage indicator and the character code change when the cursor moves. This fact can really slow down cursor movement if you are using ne through a slow connection. If you find this to be a problem, it is a good idea to turn off the status bar using either the 'Status Bar' menu item of the 'Prefs' menu or the StatusBar command. See Section 4.9.9 [StatusBar], page 40. Alternatively you can turn on the fast GUI mode using either the 'Fast GUI' menu item of the 'Prefs' menu or the FastGUI command (see Section 4.9.5 [FastGUI], page 39). In fast GUI mode the status bar is not draw in reverse, so some additional optimization can be done when refreshing it.

The letters after the line and column number represent the status of the flags associated with the current document. Flags that are off display a '-' instead of a letter. Each flag also has an associated command. The Flags command describes them all when you don't have this manual handy. Here's the list in detail:

'i' appears if the insert flag is true. See Section 4.9.4 [Insert], page 39.

'a' appears if the auto indent flag is true. See Section 4.8.8 [AutoIndent], page 37.

'b' appears if the back search flag is true. See Section 4.5.9 [SearchBack], page 32.

'c' appears if the case sensitive search flag is true. See Section 4.5.10 [CaseSearch], page 32.

'w' appears if the word wrap flag is true. See Section 4.8.7 [WordWrap], page 37.

'f' appears if the free form flag is true. See Section 4.9.6 [FreeForm], page 40.

'p' appears if the automatic preferences flag is true. See Section 4.9.2 [AutoPrefs], page 39.

'v' appears if the verbose macros flag is true. See Section 4.9.18 [VerboseMacros], page 42.

'u' appears if the undo flag is true. See Section 4.7.4 [DoUndo], page 35.

'r' appears if the read only flag is true. See Section 4.9.11 [ReadOnly], page 41.

't/T' appears as 't' if the tabs flag is true, 'T' if the shifttabs flag is also true. See Section 4.9.14 [Tabs], page 41, Section 4.9.16 [ShiftTabs], page 42.

'd' appears if the deltabs flag is true. See Section 4.9.15 [DelTabs], page 41.

'B' appears if the binary flag is true. See Section 4.9.3 [Binary], page 39.

'M'	appears if you are currently marking a block. See Section 4.4.1 [Mark], page 28.
'V'	can appear in place of 'M' if you are currently marking a vertical block. See Section 4.4.2 [MarkVert], page 28.
'R'	appears if you are currently recording a macro. See Section 4.6.1 [Record], page 33.
'P'	appears if the PreserveCR flag is true. See Section 4.9.19 [PreserveCR], page 43.
'C'	appears if the CRLF flag is true. See Section 4.9.20 [CRLF], page 43.
'*'	appears if the document has been modified since the last save, or if the `Modified` command was issued to set this flag. See Section 4.9.29 [Modified], page 45.
'@'	appears if UTF-8 I/O is enabled. See Section 4.9.33 [UTF8IO], page 46.
'A/8/U'	denotes the current buffer encoding—US-ASCII, 8-bit or UTF-8. See Section 4.9.31 [UTF8], page 46.

Note that sometimes `ne` needs to communicate some message to you. The message is usually written over the status bar, where it stays until you do something. Any action such as moving the cursor or inserting a character will restore the normal status bar.

3.3 The Input Line

The bottom line of the screen is usually occupied by the status bar (see Section 3.2 [The Status Bar], page 12). However, whenever `ne` prompts you for a command or file name or asks you to confirm some action, the bottom 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. (Prompts always ends with a colon, so it is easy to distinguish them from *error messages*, which overwrite the status line from time to time.)

`ne` uses the input line in two essentially different ways: *immediate* input and *long* input. You can easily distinguish between these two modes because in immediate input mode the cursor is not on the input line, while for long input mode it is.

Immediate input is used whenever `ne` needs you to specify a simple choice that can be expressed by one character (for example, 'y' or 'n'). When you type the character, `ne` will immediately accept and use your input. Most immediate inputs display a character just after the prompt. This character is the default response, which is used if you just press the RETURN key. Note that immediate input is not case sensitive. Moreover, if a yes/no choice is requested, *anything* other than 'y' will be considered a negative response.

Long input is used when a whole string is required. You can enter and edit your response to long inputs like a line of text in a document. Most key bindings related to line editing work on the command line exactly as they do in a document. This is true even of custom key bindings. Just edit as you are used to. Moreover, the you can paste the first line of the current clip using the keystroke that is bound to the `Paste` command, usually *CONTROL-V*. If your long input is longer than the screen width, the input line scrolls to accommodate your text so you can input very long lines even on small monitors. (There is a limit of 2048 characters.)

The default response to a long input is the response you gave to the previous long input. Your *first action* when presented with a long input will either erase the default response or allow you to edit it. If the first thing you type is a printing character, the default response will be erased. Anything else (cursor movement for example) will allow you to edit it further.

Long input also lets you access your previous long input responses with the up and down cursor commands (or with wider movement commands, such as start/end of file, page up/down, etc.). Once you find a previous input you like, you can edit it further. Long input history is not document specific, so you can recall any of your inputs regardless of which document was active when you entered it. Furthermore, `ne` saves the most recent long inputs in '`~/.ne/.history`' when you end your `ne` session and loads them again when you begin another `ne` session.

When asked to input a number, you can choose between decimal, octal and hexadecimal notation in the standard way: a number starting with '0' is considered in octal, a number starting with '0x' is considered in hexadecimal, and in all other cases decimal base is assumed.

Whenever a file name is requested, you can type a partial file name and *complete* it with the TAB key. ne will scan the current directory (or the directory that you partially specified) and search for the files matching your partial suggestion. The longest prefix common to all such files will be copied on the input line (ne will beep if no completion exists). It's easier done than said—just try. If you press TAB again, you will be brought into the file requester: only the files and directories matching your partial specification will appear, and as usual you will be able to navigate and select a file or escape. See Section 3.5 [The Requester], page 14. Note that ne considers the *last word* on the input line the partial file name to complete, no matter where the cursor is currently (you must use quotes if the name contains spaces, even if it is the only item on the input line).

Complete long input with the RETURN key. You can cancel a long input using F1, ESCAPE, ESCAPE-ESCAPE or any key that is bound to the Escape command. The effect will vary depending on what your were requested to input, but the execution of the command requiring the input will stop.

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 over menus in terms of access speed, but it is not desirable from a user interface point of view. ne has a command line that should be used whenever strange features have to be accessed, or whenever you want to use a command that you are familiar with and that is not bound to any key.

You have two ways to access the command line: by activating the menu and typing a colon (':') or by typing *CONTROL-K* (or any key that is bound to the Exec command; see Section 4.12.4 [Exec], page 53). The first method will work regardless of any key binding configuration if you activate the menus with the ESCAPE key since that key cannot be reconfigured. Of course, there is also a menu entry that does the same job.

Once you activate the command line, the status bar will turn into an input line (see Section 3.3 [The Input Line], page 13) with a 'Command:' prompt waiting for you to do 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 the name of a macro. See Section 2.6 [Basic Macros], page 7, for details.

3.5 The Requester

In various situations, ne needs to ask you to choose one string from several (where "several" can mean a lot). For this kind of event, the *requester* is issued. The requester displays the strings in as many columns as possible and lets you move with the cursor from one string to another. The strings can fill many screens, which are handled as consecutive pages. Most navigation keys work exactly as in normal editing. This is true even of custom key bindings. Thus, for instance, you can page up and down through the list with *CONTROL-P* and *CONTROL-N* (in the standard keyboard configuration).

As with the input line (see Section 3.3 [The Input Line], page 13), 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). Moreover, if you are selecting a file name there is a third possibility: by escaping with the TAB key, the file or directory name that the cursor is currently on will be copied on the input line. This allows you to choose an existing name and modify it.

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

An example of a requester is the list of commands appearing when you use the `Help` command. Another is the list of document words matching a prefix given to the `AutoComplete` command. A third example 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 that always appear in the file requester: '`./`' and '`../`'. The first one represents the current directory and can be used to force a reread of the directory. The second one represents the parent directory and can be used to move up by one directory level.

The requester presents the strings by default in "row major order," which means the second string is on the same row as the first but to its right, at the top of the second column, and so on across each row before filling in the next row down. If you prefer your lists displayed in "column major order"—the first, second, and third strings are in the same column and each column is filled before starting on the next column to the right—then use the `RequestOrder` command to switch that preference. The setting will be stored in your default preferences the next time you save them. See Section 4.9 [Preferences Commands], page 38.

3.6 Syntax Highlighting

Syntax highlighting is particularly useful for programming language text or other types of documents which have a strictly defined syntax. Colors indicate different syntactic categories of text according to the syntax definition in use.

Syntax definitions are stored in separate files. `ne` comes with a suite of syntax definitions for many popular programming languages. When you load a file, `ne` selects the appropriate syntax definition as determined by the filename extension in much the same way autoprefs are loaded. It also contains a built-in table of common filename extensions that share the same syntax definitions. For example, both '`cbl`', and '`cob`' files use the '`cobol`' definition. See the Section 4.9.30 [Syntax], page 45 command for the complete list of built-in extension mappings.

If there is no matching syntax definition for the filename extension, or if the buffer you are editing has no filename yet, or you just want to try a different syntax definition, you can load and use the syntax definition of your choice with the `Syntax` command. It takes the syntax name as a parameter. For example, the name "`c`" works for C syntax files with extensions '`.c`', '`.h`', '`c++`', etc. `ne` searches for the specified syntax definition file in the '`syntax`' subdirectory of your '`~/.ne`' directory first. If not found there, `ne` then looks in the '`syntax`' subdirectory of `ne`'s global directory for the syntax definition file. See Section 3.1 [Arguments], page 11.

With no parameter, the `Syntax` command prompts you for a syntax to load, the offered default being the currently loaded syntax if there is one.

One syntax definition you may find useful for any type of text file is called simply '`tabs`'. It highlights the TABs in your text so you can distinguish them from regular spaces.

You can create your own syntax definitions and store them in your '`~/.ne/syntax`' directory (actually, modifying the colors of an existing definition is much easier; see Chapter 6 [Hints and Tricks], page 59). A complete explanation of syntax specifications is beyond the scope of this document, but the existing definition files should prove to be useful examples. In particular, the '`syntax/c.jsf`' file contains some particularly helpful comments. Syntax definition files have a '`.jsf`' extension. Do not include that extension when using the `Syntax` command.

Syntax highlighting does incur a slight penalty in memory used per line of text, and it also consumes some CPU resources. For small to medium sized files you'll probably never notice. But for extremely large files—on the order of the size of your system's RAM—the difference could be significant. If you invoke `ne` with the `--no-syntax` parameter, `ne` will disable the syntax highlighting mechanism entirely, freeing up the memory and CPU otherwise consumed. (Note that if you are that tight on memory, you may need to disable the undo buffer as well. See Section 4.7.4 [DoUndo], page 35.)

ne uses code from another editor—the GPL-licensed joe—for its syntax highlighting capabilities. Because of this fact, the syntax definition files are identical, even to the '.jsf' extension, which is an acronym for "Joe's Syntax File". It's possible that if both joe and ne are installed on your system that they share the same syntax file directory.

3.7 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.

You activate menus with the F1 key, or in case your keyboard does not have such a key, ESCAPE, ESCAPE-ESCAPE or any key that is bound to the Escape command. Move around the menus pressing with 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 in the current menu 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 corresponds to a single command. In explaining what each menu item allows you to do, we shall simply refer you to the section that explains the command relative to the menu item.

If you plan to change ne's menu (see Section 5.2 [Changing Menus], page 56), you should take a look at the file 'default.menus' that comes with ne's distribution. It contains a complete menu configuration that clones the standard one.

3.7.1 File

The File menu contains standard items that allow loading and saving files. Quitting ne (which doesn't save changes) or exiting ne (which does save changes) is also possible.

'Open...' See Section 4.2.1 [Open], page 26.

'Open New...'
See Section 4.2.2 [OpenNew], page 26.

'Save' See Section 4.2.3 [Save], page 26.

'Save As...'
See Section 4.2.4 [SaveAs], page 26.

'Quit Now'
See Section 4.3.1 [Quit], page 27.

'Save&Exit'
See Section 4.3.2 [Exit], page 27.

'About' See Section 4.12.1 [About], page 52.

3.7.2 Documents

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

'New' See Section 4.3.3 [NewDoc], page 27.

'Clear' See Section 4.3.4 [Clear], page 27.

'Close' See Section 4.3.5 [CloseDoc], page 27.

'Next' See Section 4.3.6 [NextDoc], page 27.

'Prev' See Section 4.3.7 [PrevDoc], page 27.

'Select...'
See Section 4.3.8 [SelectDoc], page 27.

3.7.3 Edit

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

‘`Mark Block`’
See Section 4.4.1 [Mark], page 28.

‘`Cut`’ See Section 4.4.4 [Cut], page 29.

‘`Copy`’ See Section 4.4.3 [Copy], page 28.

‘`Paste`’ See Section 4.4.5 [Paste], page 29.

‘`Erase`’ See Section 4.4.7 [Erase], page 29.

‘`Through`’ See Section 4.4.12 [Through], page 30.

‘`Delete Line`’
See Section 4.11.9 [DeleteLine], page 52.

‘`Delete EOL`’
See Section 4.11.10 [DeleteEOL], page 52.

‘`Mark Vert`’
See Section 4.4.2 [MarkVert], page 28.

‘`Paste Vert`’
See Section 4.4.6 [PasteVert], page 29.

‘`Open Clip...`’
See Section 4.4.9 [OpenClip], page 29.

‘`Save Clip...`’
See Section 4.4.10 [SaveClip], page 30.

3.7.4 Search

The Search menu contains commands related to searching for specific contents or locations within a document.

‘`Find...`’ See Section 4.5.1 [Find], page 30.

‘`Find RegExp...`’
See Section 4.5.2 [FindRegExp], page 30.

‘`Replace...`’
See Section 4.5.3 [Replace], page 31.

‘`Replace Once...`’
See Section 4.5.4 [ReplaceOnce], page 31.

‘`Replace All...`’
See Section 4.5.5 [ReplaceAll], page 31.

‘`Repeat Last`’
See Section 4.5.6 [RepeatLast], page 32.

‘`Goto Line...`’
See Section 4.10.5 [GotoLine], page 47.

‘`Goto Col...`’
See Section 4.10.6 [GotoColumn], page 47.

‘`Goto Mark...`’
See Section 4.10.7 [GotoMark], page 47.

'Match Bracket'
: See Section 4.5.7 [MatchBracket], page 32.

'Set Bookmark'
: See Section 4.10.26 [SetBookmark], page 50.

'Unset Bookmark'
: See Section 4.10.28 [UnsetBookmark], page 50.

'Goto Bookmark'
: See Section 4.10.27 [GotoBookmark], page 50.

3.7.5 Macros

The Macros menu contains commands related to creating and using macros.

'Record' See Section 4.6.1 [Record], page 33.

'Stop' See Section 4.6.1 [Record], page 33.

'Replace...'
: See Section 4.5.3 [Replace], page 31.

'Play Once'
'Play Many...'
: See Section 4.6.2 [Play], page 33.

'Play Macro...'
: See Section 4.6.3 [Macro], page 34.

'Open Macro...'
: See Section 4.6.4 [OpenMacro], page 34.

'Save Macro...'
: See Section 4.6.5 [SaveMacro], page 34.

3.7.6 Extras

This menu contains a few special items that don't fit in obvious ways into other menus.

'Exec...' See Section 4.12.4 [Exec], page 53.

'Suspend' See Section 4.12.9 [Suspend], page 54.

'Help...' See Section 4.12.6 [Help], page 53.

'Refresh' See Section 4.12.8 [Refresh], page 53.

'Undo' See Section 4.7.1 [Undo], page 35.

'Redo' See Section 4.7.2 [Redo], page 35.

'Undel Line'
: See Section 4.7.3 [UndelLine], page 35.

'Center' See Section 4.8.1 [Center], page 36.

'Shift Right'
'Shift Left'
: See Section 4.4.8 [Shift], page 29.

'Paragraph'
: See Section 4.8.2 [Paragraph], page 36.

‘Adjust View’
‘Center View’
See Section 4.10.23 [AdjustView], page 49.

‘ToUpper’ See Section 4.8.3 [ToUpper], page 37.

‘ToLower’ See Section 4.8.4 [ToLower], page 37.

‘Capitalize’
See Section 4.8.5 [Capitalize], page 37.

3.7.7 Navigation

The Navigation menu contains commands related moving around in a document.

‘Move Left’
See Section 4.10.1 [MoveLeft], page 46.

‘Move Right’
See Section 4.10.2 [MoveRight], page 47.

‘Line Up’ See Section 4.10.3 [LineUp], page 47.

‘Line Down’
See Section 4.10.4 [LineDown], page 47.

‘Prev Page’
See Section 4.10.8 [PrevPage], page 47.

‘Next Page’
See Section 4.10.9 [NextPage], page 48.

‘Page Up’ See Section 4.10.10 [PageUp], page 48.

‘Page Down’
See Section 4.10.11 [PageDown], page 48.

‘Start Of File’
See Section 4.10.19 [MoveSOF], page 49.

‘End Of File’
See Section 4.10.18 [MoveEOF], page 49.

‘Start Of Line’
See Section 4.10.15 [MoveSOL], page 48.

‘End Of Line’
See Section 4.10.14 [MoveEOL], page 48.

‘Top Of Screen’
See Section 4.10.16 [MoveTOS], page 48.

‘Bottom Of Screen’
See Section 4.10.17 [MoveBOS], page 48.

‘Incr Up’ See Section 4.10.21 [MoveIncUp], page 49.

‘Incr Down’
See Section 4.10.22 [MoveIncDown], page 49.

‘Prev Word’
See Section 4.10.12 [PrevWord], page 48.

‘Next Word’
See Section 4.10.13 [NextWord], page 48.

3.7.8 Prefs

The Prefs menu contains commands related to setting, storing, and using your preferred document flags.

'Tab Size...'
: See Section 4.9.13 [TabSize], page 41.

'Tabs as Spaces'
: See Section 4.9.14 [Tabs], page 41.

'Insert/Over'
: See Section 4.9.4 [Insert], page 39.

'Free Form'
: See Section 4.9.6 [FreeForm], page 40.

'Status Bar'
: See Section 4.9.9 [StatusBar], page 40.

'Hex Code'
: See Section 4.9.10 [HexCode], page 41.

'Fast GUI'
: See Section 4.9.5 [FastGUI], page 39.

'Word Wrap'
: See Section 4.8.7 [WordWrap], page 37.

'Right Margin'
: See Section 4.8.6 [RightMargin], page 37.

'Auto Indent'
: See Section 4.8.8 [AutoIndent], page 37.

'Request Order'
: See Section 4.9.8 [RequestOrder], page 40.

'Preserve CR'
: See Section 4.9.19 [PreserveCR], page 43.

'Save CR/LF'
: See Section 4.9.20 [CRLF], page 43.

'Load Prefs...'
: See Section 4.9.24 [LoadPrefs], page 44.

'Save Prefs...'
: See Section 4.9.25 [SavePrefs], page 44.

'Load AutoPrefs'
: See Section 4.9.26 [LoadAutoPrefs], page 44.

'Save AutoPrefs'
: See Section 4.9.27 [SaveAutoPrefs], page 44.

'Save Def Prefs'
: See Section 4.9.28 [SaveDefPrefs], page 45.

3.8 Regular Expressions

Regular expressions are a powerful way of specifying complex search and replace operations. ne supports the full regular expression syntax on US-ASCII and 8-bit buffers, but has to impose a restriction on character sets when searching in UTF-8 text. See Section 3.11 [UTF-8 Support], page 24.

3.8.1 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 that 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 behaviour; better to quote the special character anyway, regardless of where is appears.

The following are the characters and character sequences that 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.

Note that when searching in UTF-8 text, a character set may contain US-ASCII characters only.

'`[^ ... ]`' '`[^`' 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. Also in this case, when searching in UTF-8 text a complemented character set may contain US-ASCII characters only.

'`^`' 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`' that 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 that matches only '`$`', and '`\[`' is a regular expression that 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`' *et cetera*, 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 that 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. These are US-ASCII letters, numbers and the underscore, independently on the buffer encoding.

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

3.8.2 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, to put a backslash in the replacement string, you have to use '`\\`'.

3.9 Automatic Preferences

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

The use of autoprefs is definitely straightforward. You simply use the '`Save AutoPrefs`' menu item (or the `SaveAutoPrefs` command; see Section 4.9.27 [SaveAutoPrefs], page 44) when the current document has the given extension and the current configuration suits your tastes. The internal state of a series of options will be recorded as a macro containing commands that reproduce the current configuration. The macro is then saved in the '`~/.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 for TEX files '`tex#ap`', and so on.

Macros are generated with short or long command names depending on the status of the verbose macros flag. See Section 4.9.18 [VerboseMacros], page 42.

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

3.10 Emergency Save

When ne is interrupted by an abnormal event (for instance, the crash of your terminal), it will try to save all unsaved documents in its current directory. Named documents will have their names prefixed with a '#'. Unnamed documents will be given names made up of hexadecimal numbers obtained by some addresses in memory that will make them unique.

3.11 UTF-8 Support

Since version 1.30, ne can manipulate UTF-8 files and supports UTF-8 when communicating with the user. At startup, ne fetches the system locale description, and checks whether it contains the string 'utf8' or 'utf-8'. In this case, it starts communicating with the user using UTF-8. This behaviour can be modified either using a suitable command line option (see see Section 3.1 [Arguments], page 11), or using Section 4.9.33 [UTF8IO], page 46. This makes it possible to display and read from the keyboard a wide range of characters.

Independently of the input/output encoding, ne keeps track of the encoding of each buffer. ne does not try to select a particular coding on a buffer, unless it is forced to do so, for instance because a certain character is inserted. Once a buffer has a definite encoding, however, it keeps it forever.

More precisely, every buffer may be in one of three *encoding modes*: US-ASCII, when it is entirely composed of US-ASCII characters; 8-bit, if it contains also other characters, but it is not UTF-8 encoded; and finally, UTF-8, if it is UTF-8-encoded.

The behaviour of ne in US-ASCII and 8-bit mode is similar to previous versions: each byte in the buffer is considered a separate character.

There are, however, two important differences: first, if I/O is not UTF-8 encoded, *any* encoding of the ISO-8859 family will work flawlessly, as ne merely reads bytes from the keyboard and displays bytes on the screen. On the contrary, in the case of UTF-8 input/output ne must take a decision as to which encoding is used for non-UTF-8 buffers, and presently this is hardwired to ISO-8859-1. Second, since version 1.34, 8-bit buffers use localized casing and character type functions. This means that case-insensitive searches or case foldings will work with, say, Cyrillic characters, provided that your locale is set correctly.

In UTF-8 mode, instead, ne interprets the bytes in the buffer in a different way—several bytes may encode a single character. The whole process is completely transparent to the user, but if you really want to look at the buffer content, you can switch to 8-bit mode (see see Section 4.9.31 [UTF8], page 46).

For most operations, UTF-8 support should be transparent. However, in some cases, in particular when mixing buffers with different encodings, ne will refuse to perform certain operations because of incompatible encodings.

The main limitation of UTF-8 buffers is that when searching for a regular expression in a UTF-8 text, character sets may only contain US-ASCII characters (see see Section 3.8 [Regular Expressions], page 20). You can, of course, partially emulate a full UTF-8 character set implementation specifying the possible alternatives using '|' (but you have no ranges).

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 commands directly.

4.1 General Guidelines

Every command in `ne` has a long and a short name. Except in a very few cases, the short name is given by two or three letters that are the initials of the words that form the long name. For instance, `SearchBack` has short name `SB`, `SaveDefPrefs` has the short name `SDP`, and `AdjustView`'s short name is `AV`. There are some exceptions however. The most frequently used commands such as `Exit` have one-letter short names (`X`). Also some commands use a different short name to avoid clashes with a more common command's short name. For example, `StatusBar`'s short name is `ST` rather than `SB` to avoid clashes with `SearchBack`'s short name.

A command always has at most one argument. This is a chosen limitation that allows `ne`'s parsing of commands and macros to be very fast. Moreover, it nullifies nearly all problems related to delimiters, escape characters, and the like. 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's 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. A flag modifier is an optional number that 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 you need to *change* a flag. For instance, you may be in insert mode and you want to overstrike, or vice versa. Absolute settings (those with a number) are useful essentially for macros. It is reasonable to use 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 time you will be prompted to type the argument on the command line.

As for the input line, for numeric arguments you can choose between decimal, octal and hexadecimal notation in the standard way: a number starting with '`0`' is considered in octal, a number starting with '`0x`' is considered in hexadecimal, and in all other cases decimal base is assumed.

When a number represents how many times `ne` should repeat an action, it is always understood that the command will terminate when the conditions for applying it are no longer true. For instance, the `Paragraph` command accepts the number of paragraphs to format. But if not enough paragraphs exists in the text, only the available ones will be formatted.

This easily allows performing operations on an entire document by specifying preposterously huge numbers as arguments. `ToUpper 200000000` will make all the words in the document upper case. (At least, one would hope so!) Note that this is much faster than recording a macro with the command `ToUpper` in it and playing it many times because in the former case 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*-\).

To handle situations such as an argument string starting with a space, `ne` implements a simple mechanism whereby you can enclose any string argument in double quotes. If the first non-blank character after the command and last character of the command line are double quotes, the quotes will be removed

and whatever is left will be used as the string argument. For example, the `Find` command to find a space could be entered on the command line or in a macro as `Find " "`. The only case needing special treatment is when a string starts and ends with double quotes. The command `Find ""quote""` would locate the next occurrence of the string '`"quote"`' (including the double quotes). However, `Find onequote"` wouldn't require special treatment because the command argument doesn't both start and end with a double quote.

4.2 File Commands

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

4.2.1 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 4.9.7 [NoFileReq], page 40.)

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

4.2.2 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 4.2.1 [Open], page 26.

4.2.3 Save

Syntax: `Save`
Abbreviation: `S`

saves the current document using its default file name.

If the current document is unnamed, the file requester will open and you will be prompted to select a file. (You can inhibit the file requester opening by using the `NoFileReq` command; see Section 4.9.7 [NoFileReq], page 40.)

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

4.2.4 SaveAs

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

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

If the optional *filename* argument is not specified, the file requester will open and you will be prompted to select a file. (You can inhibit the file requester opening by using the `NoFileReq` command; see Section 4.9.7 [NoFileReq], page 40.)

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 commands allow manipulation of the circular list of documents in `ne`.

4.3.1 Quit

Syntax: `Quit`
Abbreviation: `Q`

closes all documents and exits. If any documents are modified, you have to confirm the action.

4.3.2 Exit

Syntax: `Exit`
Abbreviation: `X`

saves all modified documents, closes them and exits. If any documents 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 that 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 a copy of the preferences of the current document.

4.3.4 Clear

Syntax: `Clear`
Abbreviation: `CL`

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

4.3.5 CloseDoc

Syntax: `CloseDoc`
Abbreviation: `CD`

closes the current document. The document is removed from `ne`'s list and, if it is the only existing document, `ne` exits. If the document was modified since it was last saved, you have to confirm the action.

4.3.6 NextDoc

Syntax: `NextDoc`
Abbreviation: `ND`

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

4.3.7 PrevDoc

Syntax: `PrevDoc`
Abbreviation: `PD`

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

4.3.8 SelectDoc

Syntax: `SelectDoc`
Abbreviation: `SD`

displays a requester containing the names of all the documents in memory. You select whichever document you want to become the current document.

If you escape from the requester the requester goes away and you are returned to your original current document.

SelectDoc is especially useful if you have a large number of documents open (say, more than 10). Otherwise, NextDoc and PrevDoc should be enough. See Section 4.3.6 [NextDoc], page 27, and Section 4.3.7 [PrevDoc], page 27.

4.4 Clip Commands

These commands control the clipping system. A *clip* is a snippet of text separate from any document, which you can save to a file or insert into a document. You can select text in a document and copy it to a clip, optionally deleting it from your text. You can also load text directly from a file into a clip. ne can have any number of clips, which are distinguished by an integer. Most clip commands act on the current clip, which can be selected with ClipNumber. Clips can be copied and pasted in two ways—normally (as lines of text) or vertically (as a rectangular block of characters).

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 and cursor together define the range of text over which clips (Cut, Copy, Erase) and left and right shifts operate .

If you invoke Mark with no arguments, it will set 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.

4.4.2 MarkVert

Syntax: MarkVert [0|1]
Abbreviation: MV

is the same as Mark, but the region manipulated by the cut/paste commands is the rectangle having as vertices the cursor and the mark. If you invoke MarkVert with no arguments, it will set the mark. If you specify 0 or 1, the mark will be canceled or set to the current position, respectively. Moreover, a capital 'V', rather than a capital 'M', will appear on the status bar.

For example, if you have the following text:

```
aaaBbbccc
aaabbbccc
aaabbbCcc
```

and you set a vertical mark at 'B' then move the cursor to 'C', you can cut or copy all of the 'B's.

If you have made a vertical cut or copy, it's very likely you will want to use PasteVert rather than the usual Paste to reinsert the text in a rectangle. See Section 4.4.6 [PasteVert], page 29.

4.4.3 Copy

Syntax: Copy [*n*]
Abbreviation: C

copies the contents of the characters lying 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 4.4.11 [ClipNumber], page 30. 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 into the current document at the cursor position. If you don't specify the clip number, the current clip is used; Specify which clip is current with Section 4.4.11 [ClipNumber], page 30.

4.4.6 PasteVert

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

vertically pastes 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 Shift

Syntax: `Shift [<|>][n][t|s]]`
Abbreviation: `SH`

shifts the text on lines between the mark and the cursor either right ('`>`', the default) or left ('`<`') by adding or removing white space on each line. The adjustment size, specified as an unsigned integer '`n`', is in units of the current tab size ('`t`') or spaces ('`s`'). The default is 1. Adjustments start at the left edge of a vertical mark, or column 1 otherwise. If the mark is not currently set, only the current line is affected.

`Shift` will insert tab characters only if the document's `Tabs` flag and the `ShiftTabs` flag are both set—in which case an upper case '`T`' will appear in the status bar. If either of the `Tabs` or `ShiftTabs` flags is unset (i.e there is no upper case '`T`' in the status bar) `Shift` will only insert spaces.

In the case of left shifts, if any indicated line has insufficient leading white space for the requested adjustment to be made, then `Shift` reports an error and makes no changes.

4.4.9 OpenClip

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

loads the given file name as the current clip, just as if you cut or copied it from the current document; see Section 4.4.3 [Copy], page 28.

If the optional *filename* argument is not specified, the file requester will open and you will be prompted to select a file. (You can inhibit the file requester opening by using the `NoFileReq` command; see Section 4.9.7 [NoFileReq], page 40.)

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

4.4.10 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 will open and you will be prompted to select a file. (You can inhibit the file requester opening by using the `NoFileReq` command; see Section 4.9.7 [NoFileReq], page 40.)

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

4.4.11 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.

4.4.12 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` 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: a simple, fast exact matching search (optionally ignoring case), and a very flexible and powerful, but slower, regular expression search based on the GNU `regex` library (again, optionally case insensitive).

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 4.5.9 [SearchBack], page 32, and Section 4.5.10 [CaseSearch], page 32.

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 current document for the given extended regular expression (see Section 3.8 [Regular Expressions], page 20) . 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 4.5.9 [SearchBack], page 32, and Section 4.5.10 [CaseSearch], page 32.

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 to the first match of the most recent find string or regular expression and prompts you for 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 stopping the search ('`Last`');
- stopping the search immediately ('`Quit`');
- replacing *all* occurrences of the find string with the given string ('`All`');
- reversing the search direction ('`Backward`' or '`Forward`'); this choice will also modify the value of the back search flag. See Section 4.5.9 [SearchBack], page 32.

`Replace` 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, there are some special features you can use in the replace string (see Section 3.8 [Regular Expressions], page 20) . See Section 4.5.2 [FindRegExp], page 30.

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 with Section 4.6.1 [Record], page 33, there is no trace in the macro of your interaction with `ne` during the replacement process. When the macro is played, you will again have 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 31, Section 4.5.5 [ReplaceAll], page 31, and Section 4.5.6 [RepeatLast], page 32.

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 it exists, 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 replaced by `ReplaceAll`. See Section 4.7.1 [Undo], page 35.

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).

`RepeatLast` is especially 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. Avoiding interaction during macros is the primary reason the commands `ReplaceAll` and `ReplaceOnce` are provided.

4.5.7 MatchBracket

Syntax: `MatchBracket`
Abbreviation: `MB`

moves the cursor to the bracket associated with the bracket the cursor is on. If the cursor is not on a bracket, or there is no bracket associated with the current one, an error message is issued. Recognized brackets are '`{}`', '`()`', '`[]`' and '`<>`'. See Section 4.5.8 [AutoMatchBracket], page 32.

4.5.8 AutoMatchBracket

Syntax: `AutoMatchBracket [0..15]`
Abbreviation: `AMB`

sets the auto match bracket mode. When the cursor is on a recognized bracket ('`{}`', '`()`', '`[]`' or '`<>`') and the associated matching bracket is on the screen, that matching bracket will be indicated according to the mode. The mode is either zero for no bracket matching, or the sum of 1 (altered foreground and background brightness), 2 (inverse), 4 (bold), and 8 (underline). If no mode is specified, `ne` prompts your for one. The default mode is 1. See Section 4.5.7 [MatchBracket], page 32.

4.5.9 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 invoke `SearchBack` 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 also can be set through interactions with the `Replace` command. See Section 4.5.3 [Replace], page 31.

4.5.10 CaseSearch

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

sets the case sensitivity flag. When this flag is true, the search commands distinguish between the upper and lower case letters. By default the flag is false.

If you invoke `CaseSearch` 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.

4.5.11 AutoComplete

Syntax: `AutoComplete [`*prefix*`]`
Abbreviation: `AC`

attempts to extend the *prefix* using matching words from your open documents, and inserts the extended text into your document. If the *prefix* can be extended unambiguously, the matching text is immediately inserted into your document. Otherwise, `ne` displays a selection of all words in open documents that match *prefix*, and inserts the word you select into the current document. Matching words from the current document display normally; those which only exist in other open documents are bold and with a trailing asterisk. If no *prefix* is given on the command line, or if `AutoComplete` is selected from the `Extras` menu or using a keyboard shortcut, the word characters to the immediate left of the cursor in the current document are used as the *prefix*. Note that if no word characters are to the left of the cursor, or the *prefix* given on the command line is an empty string (`""`), then all words in all your open documents are displayed. Prefix matches may be case sensitive or not depending on the current document's `CaseSearch` flag state. See Section 4.5.10 [CaseSearch], page 32.

4.6 Macros Commands

Macros are lists of commands. Any series of operations that 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 '`#`' are considered comments; lines starting with other nonalphabetical characters are presently ignored). But the real power of macros is that they be recorded during the normal usage of `ne`. When the recording terminates, the operations that have been recorded can be saved for later use. Note that each document has a current macro (the last macro that 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, `ne` starts recording your actions in a new macro. When it becomes false, the macro recording is stopped, and the macro can be played or saved via Section 4.6.2 [Play], page 33, or Section 4.6.5 [SaveMacro], page 34.

If you call invoke `Record` 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 both starting and stopping macro recording to a single key while still being able to specify "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 containing 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.

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 4.9.7 [NoFileReq], page 40.)

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

Note that macros whose names do 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 by that name in the current directory. If this search also fails, `ne` looks in '`~/.ne`' and finally in `ne`'s global directory (defined when `ne` was built, or in a place specified by your `NE_GLOBAL_DIR` environment variable) for a macro file by that name.

Warning: 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 35. 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 even just `MaCrO` will use the cached version of `~/foobar/macro`. Note that the cache table is global to `ne` and not specific to any single document. This greatly improves efficiency when macros are used repeatedly.

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 33.

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 4.9.7 [NoFileReq], page 40.)

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 in a file with the given 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 4.9.7 [NoFileReq], page 40.)

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

`SaveMacro` 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. Note that `SaveMacro` converts `InsertChar` commands into a possibly smaller number of `InsertString` commands. This makes macros easier to read and edit. See Section 4.11.1 [InsertChar], page 51, and Section 4.11.2 [InsertString], page 51.

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 macros it has to play.

`UnloadMacros` is especially useful 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. After you undo a number of actions, you can `Redo` all or some of them; see Section 4.7.2 [Redo], page 35. However, if you take any new actions after having `Undo`ne some, you can no longer `Redo` those `Undo`ne actions. See Section 4.7.2 [Redo], page 35.

4.7.2 Redo

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

redoes the last *n* actions undone by `Undo` (as long as you don't take any actions that change the text between the `Undo` and `Redo` commands). If *n* is not specified, it is assumed to be one. You can only `Redo` actions that have been `Undo`ne. See Section 4.7.1 [Undo], page 35.

4.7.3 UndelLine

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

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

`UndelLine` 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 `UndelLine` works independently of the status of the undo flag. See Section 4.7.4 [DoUndo], page 35.

4.7.4 DoUndo

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

sets the flag that 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 invoke `DoUndo` 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 '`U`' will be upper case if the flag is true and the `AtomicUndo` level is non-zero.)

The usefulness of this option relies in the fact that 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 we see no reason to not keep the undo flag always true, and this is indeed the default.

4.7.5 AtomicUndo

Syntax: `AtomicUndo [0|+|-]`
Abbreviation: `AU`

increases, decreases, sets or clears the `AtomicUndo` level. The normal level is zero. All current document changes made while the `AtomicUndo` level is above zero are treated as a single change by the `Undo` and `Redo` commands. If no parameter is given, a level of 0 is set to 1; otherwise the current non-zero level is decremented. If 0 is given, the level is reset to zero. Parameters of '`+`' and '`-`' respectively increment and decrement the level, which in no case can be negative. If the level is above zero, the `DoUndo` flag in the status bar, which is normally a lower-case '`u`', becomes upper case '`U`'.

Two other actions will reset the `AtomicUndo` level to zero: invoking the `Undo` command, and disabling the undo system with the `DoUndo` command. You cannot set a non-zero `AtomicUndo` level unless the undo system is enabled.

Note: macros that you wish to undo and redo atomically—i.e., as if they were single commands—should begin with `AtomicUndo +` and end with `AtomicUndo -` so that they can call and/or be called by other macros.

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 or a line with leading white space incongruous with that of preceeding lines.

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 37.

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 37.

`ne`'s notion of a paragraph includes the current non-blank line (regardless of its leading white space) and all subsequent non-blank lines that have identical (to each other's—not to the first line's) leading white space. Therefore your paragraphs can have various first line indentations and left margins. `Paragraph` also takes into account characters commonly used at the left edge of block comments ('`/`', '`*`', '`#`', spaces, and tabs) or quoted email ('`>`'), and attempts to preserve those on the left edge when possible.

After the `Paragraph` command completes, your cursor will be positioned on the first non-blank character after the last reformatted paragraph (or, if there is no such character, at the end of the document).

`Paragraph` does not insert "smart" spaces after full stops and colons, nor does it do other "smart" things such as justification. If you need such facilities, you should consider using a text formatter. TEX for example is usually an excellent choice.

4.8.3 ToUpper

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

shifts to upper case the letters from the cursor position up to the end of a word, and moves to the first 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 only want to upper case 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 `ToUpper` 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 37.

4.8.5 Capitalize

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

acts exactly like `ToUpper`, but capitalizes, that is, makes the first letter upper case and the other ones lower case. See Section 4.8.3 [ToUpper], page 37.

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 37.

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 zero 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 when you attempt to insert characters beyond the right margin. See Section 4.8.6 [RightMargin], page 37. `ne` will attempt to preserve certain invariant characters at the left edge of paragraphs typically used to indicate comments in source code or quoted passages in email text. See Section 4.8.2 [Paragraph], page 36.

If you invoke `WordWrap` 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 invoke `AutoIndent` 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 35.

4.9 Preferences Commands

These commands allow you to set your preferences, that is, the value of a series of flags that 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 either by writing them out to a file (saved as a macro that suitably sets the flags) or by pushing them onto a "preferences stack". 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 and the turbo parameter are global to ne, and are not saved. 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 '~/.ne/.default#ap'.

Note that there is an automatic preferences system, which automagically loads a preferences file related to the extension of the file name. Automatic preferences files are kept in your '~/.ne' directory. 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 4.9.2 [AutoPrefs], page 39.

4.9.1 Flags

Syntax: Flags
Abbreviation: FLAG

displays a list of all the status flags for ne and their associated commands. It is not recorded when recording a macro.

```
FLAG COMMAND      ABBR  DESCRIPTION
  i   Insert        I    inserts new characters (vs. replacing)
  a   AutoIndent    AI   aligns cursor under previous line after <Return>
  b   SearchBack    SB   searches search backward rather than forward
  c   CaseSearch    CS   searches are case sensitive
  w   WordWrap      WW   breaks long lines as you type
  f   FreeForm      FF   allows cursor to move beyond the end of lines
  p   AutoPrefs     AP   use automatic preferences based on file extension
  v   VerboseMacros VM   record macros using use long command names
  u   DoUndo        DU   record edits for later undoing
  r   ReadOnly      RO   changes are not allowed
  t/T Tabs          TAB  TAB key inserts TABs instead of spaces
  T   ShiftTabs     ST   Shift may insert TABs (only if 't' is also set)
  d   DelTabs       DT   BS and DEL may remove tabs worth of space
  B   Binary        B    affects file loading/saving
  M   Mark          M    mark set for line-oriented block operations
  V   MarkVert      MV   like mark, but block is rectangle
  R   Record        REC  actions are being recorded in a macro
  P   PreserveCR    PCR  affects how <CR> chars are loaded from files
  C   CRLF          CRLF use CR/LF as line terminator
  *   Modified      MOD  document has been modified since last saved
  @   UTF8IO        U8IO I/O (keyboard and terminal) are UTF-8 encoded
A/8/U UTF8          U8   the document encoding (ASCII, 8-bit or UTF-8)
```

The `RequestOrder` and `AutoMatchBracket` flags' states are not indicated on the status bar. See Section 4.9.8 [RequestOrder], page 40 and Section 4.5.8 [AutoMatchBracket], page 32 respectively.

4.9.2 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 your '`~/.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 3.9 [Automatic Preferences], page 23.

If you invoke `AutoPrefs` 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.

4.9.3 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, line feeds, 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 itself. Thus, binary editing within `ne` should be considered not a normal activity, but rather an exceptional one.

If you invoke `Binary` 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.

4.9.4 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 also governs the behaviour of the `InsertChar` and `InsertString` commands.

If you invoke `Insert` 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.

4.9.5 FastGUI

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

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

If you invoke `FastGUI` with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively.

The `FastGUI` setting is saved in your '`~/.ne/.default#ap`' file when you use the `SaveDefPrefs` command or the '`Save Def Prefs`' menu. It is not saved by the `SaveAutoPrefs` command.

4.9.6 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 there is no text present (however, you cannot move inside the space expansion of a TAB character).

If you invoke `FreeForm` 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 that has engaged users 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.

4.9.7 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 invoke `NoFileReq` 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 RequestOrder

Syntax: `RequestOrder [0|1]`
Abbreviation: `RQO`

sets the request order flag. When this flag is true, the requester displays entries in column order. Otherwise entries are displayed by rows.

If you invoke `RequestOrder` with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively.

The `RequestOrder` setting is saved in your '`~/.ne/.default#ap`' file when you use the `SaveDefPrefs` command or the '`Save Def Prefs`' menu. It is not saved by the `SaveAutoPrefs` command.

4.9.9 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 we are 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 that do not allow to set a scrolling region can produce annoying flashes at the bottom of the screen.

If you invoke `StatusBar` with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively.

The `StatusBar` setting is saved in your '`~/.ne/.default#ap`' file when you use the `SaveDefPrefs` command or the '`Save Def Prefs`' menu. It is not saved by the `SaveAutoPrefs` command.

4.9.10 HexCode

Syntax: `HexCode [0|1]`
Abbreviation: `HC`

sets the hex code flag. When this flag is true, the hexadecimal code of the character currently under the cursor is displayed on the status line.

4.9.11 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 that you cannot write to. See Section 4.2.1 [Open], page 26.

If you invoke `ReadOnly` 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.

4.9.12 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 61.) 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 you can accelerate the recognition of the ESCAPE key by hitting it twice in a row.

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

4.9.13 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.

4.9.14 Tabs

Syntax: `Tabs [0|1]`
Abbreviation: `TAB`

sets the `Tabs` flag. When this flag is set, TAB key and the `InsertTab` command will insert literal *TAB* characters. Otherwise it will insert enough spaces to have the same visual effect.

In normal editing, the TAB key invokes the command "`InsertTab 1`". Unlike most others, the TAB key cannot be mapped to other commands. Thus the `Tabs` flag provides the only customization `ne` offers for the TAB key.

If set, either a lower case '`t`' or upper case '`T`' will appear in the status bar depending on the state of the `ShiftTabs` flag. (The `ShiftTabs` flag is irrelevant if the `Tabs` flag is off.) See Section 4.9.16 [ShiftTabs], page 42.

4.9.15 DelTabs

Syntax: `DelTabs [0|1]`
Abbreviation: `DT`

sets the DelTabs flag. When this flag is set, a 'd' will appear on the status bar, and the BACKSPACE and DEL keys will remove a tab's worth of *space* characters if a *TAB* character could have occupied the same whitespace in the current line as the removed spaces. This is the deletion counterpart to the Tabs flag. See Section 4.9.14 [Tabs], page 41.

4.9.16 ShiftTabs

Syntax: ShiftTabs [0|1]
Abbreviation: SHT

sets the ShiftTabs flag. ShiftTabs has an effect only when the Tabs flag is set, in which case an upper case 'T' appears in the status line. When this flag and the Tabs flag are both set, left and right Shift commands may use tab characters to adjust leading white space. Otherwise only spaces are used. See Section 4.4.8 [Shift], page 29.

4.9.17 Turbo

Syntax: Turbo [*steps*]
Abbreviation: TUR

sets the turbo parameter. Iterated actions and global replaces will update at most *steps* lines of the screen (or at most twice the number of visible rows if *steps* is zero); 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 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 that 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 zero, which means twice the number of lines of the screen; for several reasons this does seem to be a good value.

4.9.18 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 edit the macro manually, but it can slow down command parsing.

If you invoke VerboseMacros 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.

The VerboseMacros setting is saved in your '~/.ne/.default#ap' file when you use the SaveDefPrefs command or the 'Save Def Prefs' menu. It is not saved by the SaveAutoPrefs command.

4.9.19 PreserveCR

Syntax: `PreserveCR [0|1]`
Abbreviation: `PCR`

sets the preserve carriage returns flag. When a file is loaded into a buffer for which this flag is false, both CR (carriage return) and NL (new line) characters are treated as line terminators. If the flag is true, CR characters do not act as line terminators but are instead preserved in the buffer. This flag has no effect except when loading a file into a buffer.

If you invoke `PreserveCR` 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 '`P`' will appear on the status bar if the flag is true.

4.9.20 CRLF

Syntax: `CRLF [0|1]`
Abbreviation: `CRLF`

sets the CR/LF flag. When a file is saved from a buffer for which this flag is true, both a CR (carriage return) and a NL (new line) character are output as line terminators. This flag has no effect except when saving a file.

This flag is automatically set if you load a file that has at least one CR/LF sequence into it.

If you invoke `CRLF` 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 '`C`' will appear on the status bar if the flag is true.

4.9.21 VisualBell

Syntax: `VisualBell [0|1]`
Abbreviation: `VB`

sets the visual bell flag. When this flag is true, the terminal will flash (if possible) instead of beeping.

If you invoke `VisualBell` with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively.

4.9.22 PushPrefs

Syntax: `PushPrefs [`*n*`]`
Abbreviation: `PUSHP`

pushes *n* copies of the user preferences onto a stack. If not specified, *n* defaults to one. Use the `PopPrefs` command to pop preferences off the stack and restore the values. See Section 4.9.23 [PopPrefs], page 43. Note that the preferences stack is global, not buffer-specific, so you could `PushPrefs` one buffer's preferences, switch buffers, then `PopPrefs` those preferences, thereby altering the preferences for the second buffer. The maximum preferences stack depth is 32.

`PushPrefs` and `PopPrefs` are useful in macros that require certain preferences to work properly. A macro can `PushPrefs`, change any preferences necessary, do its work, then `PopPrefs` to restore the users previous preferences settings.

`PushPrefs` saves the following values on the preferences stack:

```
AutoIndent  DelTabs   NoFileReq    StatusBar  VisualBell
AutoPrefs   DoUndo    PreserveCR   ShiftTabs  WordWrap
Binary      FreeForm  ReadOnly     Tabs
CaseSearch  HexCode   RightMargin  TabSize
ClipNumber  Insert    SearchBack   UTF8Auto
```

4.9.23 PopPrefs

Syntax: `PopPrefs [`*n*`]`
Abbreviation: `POPP`

pops *n* sets of preferences from the preferences stack (where they were placed previously by `PushPrefs`) and applies those preferences to the current buffer. See Section 4.9.22 [PushPrefs], page 43. If not specified, *n* defaults to one. Note that the preferences stack is global, not buffer specific. Therefore you could `PushPrefs` one buffer's preferences, switch buffers, then `PopPrefs` those settings altering the preferences for the second buffer. The maximum preferences stack depth is 32.

`PushPrefs` and `PopPrefs` are useful in macros that require certain preferences to work properly. A macro can `PushPrefs`, change any preferences necessary, do its work, then `PopPrefs` to restore the users previous preferences settings.

PopPrefs restores the following values from the preferences stack:

```
AutoIndent  DelTabs   NoFileReq    StatusBar  VisualBell
AutoPrefs   DoUndo    PreserveCR   ShiftTabs  WordWrap
Binary      FreeForm  ReadOnly     Tabs
CaseSearch  HexCode   RightMargin  TabSize
ClipNumber  Insert    SearchBack   UTF8Auto
```

4.9.24 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 4.9.7 [NoFileReq], page 40.) 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 only option modifiers. You can manually edit a preferences file for special purposes, such as filtering out specific settings. See Chapter 6 [Hints and Tricks], page 59.

4.9.25 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 4.9.7 [NoFileReq], page 40.) If you escape from the file requester, you can input the file name on the command line.

4.9.26 LoadAutoPrefs

Syntax: `LoadAutoPrefs`
Abbreviation: `LAP`

loads the preferences file in '`~/.ne`' associated with the current document's file name extension. If the current file name has no extension, the default preferences are loaded. See Section 4.9.2 [AutoPrefs], page 39.

4.9.27 SaveAutoPrefs

Syntax: `SaveAutoPrefs`
Abbreviation: `SAP`

saves the current preferences on the file in '`~/.ne`' associated with the current document's file name extension. If the current file name has no extension, an error message is issued. See Section 4.9.2 [AutoPrefs], page 39.

4.9.28 SaveDefPrefs

Syntax: `SaveDefPrefs`
Abbreviation: `SDP`

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

4.9.29 Modified

Syntax: `Modified [0|1]`
Abbreviation: `MOD`

sets the modified flag. This flag is set automatically whenever a buffer is modified, and is used to determine which buffers need to be saved when `ne` exits. Normally you would not alter this flag, but when a buffer is inadvertently modified and you don't want the changes saved, `Modified` provides a way to make `ne` consider the buffer unchanged.

If you invoke `Modified` 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 asterisk ('`*`') will appear on the status bar if the flag is true.

4.9.30 Syntax

Syntax: `Syntax [`*name*`|*]`
Abbreviation: `SY`

loads the syntax with the given name, and colors the current buffer accordingly.

If the optional *name* argument is not specified, you are prompted for one. The current one, if set, is suggested as the default. The special *name* * turns off syntax highlighting for the current document. Otherwise, *name* must match a syntax definition either in your '`~/.ne/syntax`' directory or in a directory named '`syntax`' inside `ne`'s global directory. Additionally, `ne` has a table mapping common suffixes to syntax names. If there is no syntax with a given name, `ne` will try to remap the name using the following table (the string before the colon is the name of the syntax file):

```
ada: adb, ads
asm: s
c: c++, cc, cpp, h, h++ hpp, l, lex, y, yacc
cobol: cbl, cob
csh: tcsh
diff: patch
fortran: f, for
html: htm
java: js
lisp: el, lsp
mason: mas
ocaml: ml, mli
pascal: p, pas
perl: pl, pm
ps: eps
python: py, sage
rexx: rex
ruby: rb
sh: bash, bash_login, bash_logout, bash_profile, bashrc, ksh,
    profile, rc
skill: il
tex: latex, dtx, sty
texinfo: texi, txi
troff: 1
```

```
verilog: v, vh, vhd
xml: xsd
```

4.9.31 UTF8

Syntax: `UTF8 [0|1]`
Abbreviation: `U8`

sets the UTF-8 flag. When this flag is true, `ne` considers the current buffer as UTF-8 coded. Note that this flag is set automatically upon file loading (if possible) if you required automatic detection. See Section 4.9.32 [UTF8Auto], page 46.

If you invoke `UTF8` with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively. When you try to set this flag, the buffer will be checked for UTF-8 compliance, and you will get an error message in case of failure. When you try to reset it, the buffer is set to ASCII or 8-bit, depending on its content. A '`U`' will appear on the status bar if the flag is true. Alternatively, an '`A`' or an '`8`' will be displayed to denote whether the buffer is composed exclusively by US-ASCII characters, or also by other 8-bit characters (whose encoding is likely to be part of the ISO-8859 family). Note that each time this command modifies the buffer encoding, it also resets the undo buffer.

4.9.32 UTF8Auto

Syntax: `UTF8Auto [0|1]`
Abbreviation: `U8A`

sets the UTF-8 automatic-detection flag. When this flag is true, `ne` will try to guess whether a file just loaded is UTF-8 encoded. Moreover, when a non US-ASCII character is inserted in a pure US-ASCII buffer, ne will automatically switch to UTF-8. See Section 4.9.31 [UTF8], page 46. The flag is true by default if `ne` detects UTF-8 I/O at startup. See Section 4.9.33 [UTF8IO], page 46.

If you invoke `UTF8Auto` with no arguments, it will toggle the flag. If you specify 0 or 1, the flag will be set to false or true, respectively.

4.9.33 UTF8IO

Syntax: `UTF8IO [0|1]`
Abbreviation: `U8IO`

sets the UTF-8 input/output flag. This flag is set automatically depending on your locale setting, and is used to determine whether communication with the user (keyboard and terminal) should be UTF-8 encoded. Normally you would not alter this flag, but sometimes `ne` may make the wrong guess (e.g., when you are remotely connected).

If you invoke `UTF8IO` 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 '`@`' will appear on the status bar if the flag is true.

4.10 Navigation Commands

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

4.10.1 MoveLeft

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

moves the cursor to 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 to 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 to the *line*th line of the file. If *line* is zero or greater than the number of lines in the file, the cursor is moved to the last line.

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

4.10.6 GotoColumn

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

moves the cursor to 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 response is the current column number.

4.10.7 GotoMark

Syntax: `GotoMark`
Abbreviation: `GM`

moves the cursor to the current mark, if it exists. See Section 4.4.1 [Mark], page 28.

`GotoMark` is mainly useful if you forgot where you started marking. If you want to record positions in a file and jump to them later, you may want to use a bookmarks. See Section 4.10.26 [SetBookmark], page 50.

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 to 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 to 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 PageUp

Syntax: `PageUp [n]`
Abbreviation: `PUP`

pages the screen backward by *n* screens. If *n* is not specified, it is assumed to be one.

4.10.11 PageDown

Syntax: `PageDown [n]`
Abbreviation: `PDN`

pages the screen forward by *n* screens. If *n* is not specified, it is assumed to be one.

4.10.12 PrevWord

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

moves the cursor to the first character of 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 to the start of that word).

4.10.13 NextWord

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

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

4.10.14 MoveEOL

Syntax: `MoveEOL`
Abbreviation: `EOL`

moves the cursor to the end of the current line (`EOL` = end of line).

4.10.15 MoveSOL

Syntax: `MoveSOL`
Abbreviation: `SOL`

moves the cursor to the start of the current line (`SOL` = start of line).

4.10.16 MoveTOS

Syntax: `MoveTOS`
Abbreviation: `TOS`

moves the cursor to the top line of the screen (`TOS` = top of screen).

4.10.17 MoveBOS

Syntax: `MoveBOS`
Abbreviation: `BOS`

moves the cursor to the lowest line currently visible (`BOS` = bottom of screen).

4.10.18 MoveEOF

Syntax: `MoveEOF`
Abbreviation: `EOF`

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

4.10.19 MoveSOF

Syntax: `MoveSOF`
Abbreviation: `SOF`

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

4.10.20 MoveEOW

Syntax: `MoveEOW`
Abbreviation: `EOW`

moves the cursor one character past the end of the current word.

`MoveEOW` is extremely useful in macros, because it allows you to copy precisely the word the cursor is on. See Chapter 6 [Hints and Tricks], page 59.

4.10.21 MoveIncUp

Syntax: `MoveIncUp`
Abbreviation: `MIU`

moves the cursor incrementally towards the beginning of the document. More precisely, if the cursor is not on the start of the line it lies on, then it is moved to the start of that line. Otherwise, if it is on the first line of the screen, then it is moved to the start of the document; otherwise, it is moved to the first line of the screen.

4.10.22 MoveIncDown

Syntax: `MoveIncDown`
Abbreviation: `MID`

moves the cursor incrementally towards the end of the document. More precisely, if the cursor is not on the end of the line it lies on, then it is moved to the end of that line. Otherwise, if it is on the last line of the screen, then it is moved to the end of the document; otherwise, it is moved to the last line of the screen.

4.10.23 AdjustView

Syntax: `AdjustView [T|M|B|L|C|R] [`*n*`]`
Abbreviation: `AV`

shifts the view (text visible in the terminal window) horizontally or vertically without changing the cursor's position in the document. View adjustments are constrained by the current TAB size and the length and width of the current document. If called with no arguments '`T`' is assumed.

'`T`', '`M`', and '`B`' cause vertical shifts so that the current line becomes the top, middle, or bottom-most visible line respectively.

'`L`', '`C`', and '`R`' cause horizontal shifts, making the current column the left-most, center, or right-most visible positions.

A optional number *n* immediately after '`T`', '`B`', '`L`', or '`R`' indicate the number or rows or columns to shift the view toward the top, bottom, left, or right of the window.

Horizontal and vertical adjustment specifications may be combined, so that for example '`AdjustView TL`' shifts the view so that the current position becomes the top left-most character on screen (within the limits of the current TAB size). Likewise, '`AdjustView B3R5`' shifts the view three lines toward the bottom and five columns (excepting TAB size) toward the right.

4.10.24 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 is very useful in order to gain some keystrokes on systems with very few keys. See also Section 4.10.25 [ToggleSEOL], page 50, Section 4.10.19 [MoveSOF], page 49, and Section 4.10.18 [MoveEOF], page 49.

4.10.25 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 is very useful in order to gain some keystrokes on systems with very few keys. See also Section 4.10.24 [ToggleSEOF], page 50, Section 4.10.15 [MoveSOL], page 48, and Section 4.10.14 [MoveEOL], page 48.

4.10.26 SetBookmark

Syntax: `SetBookmark [`*`n`*`|+1|-1|-]`
Abbreviation: `SBM`

sets a document bookmark to the current cursor position. Each document has 10 available bookmarks designated '`0`' to '`9`', plus the automatic bookmark designated by '`-`'. If no option is given, '`0`' is assumed. Values of *n* from '`0`' to '`9`' set the *n*th bookmark, while '`+1`' and '`-1`' indicate respectively the next and previous available unset bookmarks. You can also set the '`-`' automatic bookmark, but it will be reset automatically to the current position whenever a `GotoBookmark` command is issued.

4.10.27 GotoBookmark

Syntax: `GotoBookmark [`*`n`*`|+1|-1|-]`
Abbreviation: `GBM`

moves the cursor to the designated bookmark if that bookmark is set; see Section 4.10.26 [SetBookmark], page 50. Each document has 10 available bookmarks designated '`0`' to '`9`', plus the automatic bookmark designated by '`-`'. If no option is given, '`0`' is assumed. The optons '`+1`' and '`-1`' indicate respectively the next and previous set bookmarks, so that repeated `GotoBookmark +1` commands will cycle through all currently set bookmarks. When successful, the '`-`' automatic bookmark is set to the position in the document from which the command was issued, so that `GotoBookmark -` returns you to the location from which you last issued a successful `GotoBookmark` command. Subsequent repeated `GotoBookmark -` commands will toggle you between the two locations.

4.10.28 UnsetBookmark

Syntax: `UnsetBookmark [`*`n`*`|+1|-1|-|*]`
Abbreviation: `UBM`

unsets either the *n*th bookmark, the next (*+1*) or previous (*-1*) set bookmarks, the automatic (-) bookmark, or all (*) bookmarks, making it as if they had never been set; see Section 4.10.26 [SetBookmark], page 50. If no option is specified, *n* is assumed to be zero. While you can unset the automatic bookmark '`-`', it will be reset automatically to the current position whenever a `GotoBookmark` command is issued. Each document's valid bookmark designations are 0 to 9, and the '`-`' automatic bookmark.

4.11 Editing Commands

These commands allow modifying a document directly.

4.11.1 InsertChar

Syntax: `InsertChar [code]`
Abbreviation: `IC`

inserts a character whose ASCII code is *code* at the current cursor position. *code* can be either decimal, hexadecimal if preceded by '`0x`', or octal if preceded by '`0`'. In any case, *code* must be different from 0. All the currently active preferences options (insert, word wrapping, auto indent, *et cetera*) are applied.

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

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 4.11.8 [InsertLine], page 52.

Note also that `SaveMacro` converts `InsertChar` commands into a possibly smaller number of `InsertString` commands. This makes macros easier to read and edit. See Section 4.6.5 [SaveMacro], page 34.

4.11.2 InsertString

Syntax: `InsertString [text]`
Abbreviation: `IS`

inserts *text* at the current cursor position. If the optional argument *text* is omitted, you will be prompted for it on the command line. All the currently active preferences options (insert, word wrapping, auto indent, *et cetera*) are applied.

Note that `SaveMacro` converts `InsertChar` commands into a possibly smaller number of `InsertString` commands. This makes macros easier to read and edit. See Section 4.6.5 [SaveMacro], page 34.

4.11.3 InsertTab

Syntax: `InsertTab [n]`
Abbreviation: `IT`

inserts either *n* literal *TAB* characters or one or more spaces sufficient to advance the current cursor position *n* tab stops depending on the `Tabs` flag. See Section 4.9.14 [Tabs], page 41, Section 4.9.13 [TabSize], page 41.

4.11.4 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 words, the carriage return between the two lines will be deleted. Note that if the cursor is past the end of the current line, no action will be performed.

4.11.5 DeletePrevWord

Syntax: `DeletePrevWord [n]`
Abbreviation: `DPW`

deletes text from the current position to the first character of 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 delete to the start of that word).

4.11.6 DeleteNextWord

Syntax: `DeleteNextWord [n]`
Abbreviation: `DNW`

deletes text from the current position to the next word *n* times. If the optional *n* argument is not specified, it is assumed to be one.

4.11.7 Backspace

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

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

4.11.8 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.

4.11.9 DeleteLine

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

deletes *n* lines starting from the current cursor position, putting the last one in the temporary buffer, from which it can be undeleted. See Section 4.7.3 [UndelLine], page 35. 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`.

4.11.10 DeleteEOL

Syntax: `DeleteEOL`
Abbreviation: `DE`

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

`DeleteEOL` 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 the copyright splash screen and places a simple information line containing the version and build date of `ne` on the status bar. Press any key to dismiss this screen.

4.12.2 Alert

Syntax: `Alert`
Abbreviation: `AL`

beeps or flashes, depending on the value of the visual bell flag.

4.12.3 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).

4.12.4 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 (though the command you type is).

`Exec` 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 34.

4.12.5 Flash

Syntax: `Flash`
Abbreviation: `FL`

acts as `Beep`, but interchanging the words "beep" and "flash". Same comments apply. See Section 4.12.3 [Beep], page 53.

4.12.6 Help

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

displays some help about the command *name* (both the short and the long versions of the command names 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.

Invocations of the `Help` command are never registered while recording macros so that you can safely access the help system while recording. See Section 4.6.1 [Record], page 33.

4.12.7 NOP

Syntax: `NOP`
Abbreviation: `NOP`

does nothing. Mainly useful for inhibiting standard key bindings.

4.12.8 Refresh

Syntax: `Refresh`
Abbreviation: `REF`

refreshes the display. `Refresh` 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.

`Refresh` has the side effect of checking to see if your window size has changed, and will modify the display to take that into account.

4.12.9 Suspend

Syntax: `Suspend`
Abbreviation: `SU`

suspends `ne` and returns you to a shell prompt; usually, the shell command `fg` is used to resume `ne`.

4.12.10 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, control returns to `ne`.

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

4.12.11 Escape

Syntax: `Escape`
Abbreviation: `ESC`

toggles the menus on and off, 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 33.

4.12.12 KeyCode

Syntax: `KeyCode`
Abbreviation: `KC`

prompts you to press a key, and reports on the status line the key code `ne` associates with that key. This can be useful while configuring your '`~/.ne/.keys`' file. It also reports the input class for that key. Input class codes are: ALPHA, COMMAND, RETURN, TAB, IGNORE, and INVALID.

5 Configuration

In this chapter we shall 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.

5.1 Key Bindings

`ne` allows you to associate any keystroke with any command. To accomplish this task, you have to create a (possibly UTf-8) file named '`.keys`' in your '`~/.ne`' directory. You can change the default name (possibly specifying a complete path) using the `--keys` argument (see Section 3.1 [Arguments], page 11).

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. Each line starting with '`SEQ`' binds a character sequence to a key. All other lines are considered comments. 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 the file '`default.keys`' that comes with `ne`'s distribution: it contains a complete, commented definition of `ne`'s standard bindings that you can modify with a trial-and-error approach.) The easiest way to see the code `ne` uses for a given key is by using the Section 4.12.12 [KeyCode], page 54 command. It prompts you to press a key, then reports the code for that key on the status bar.

You can write just the hexadecimal 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.

command can be any `ne` command, including `Escape` (which allows reconfiguring the menu activator) and `Macro`, which allows binding 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 34.

Note that you cannot *ever* redefine RETURN or 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.

Besides the "standard" combinations (e.g., *CONTROL-letter*), it possible to program combinations based on the META key (a.k.a. ALT). The situation in this case is a bit more involved, because depending on the terminal emulator you are using, the effect of the META key can be widely different. For instance, `xterm` raises the eighth bit of a character, so, for instance,

```
KEY 81 MoveSOF
```

binds *CONTROL-META-a* to the action of moving to the start of the document. However, `gnome-terminal` will emit the character of ASCII code 1 prefixed with ESC instead ("`\x1b\x01`"). To handle this case, `ne` provides codes from 180 on for *simulated META sequences*: for instance,

```
KEY 181 MoveSOF
```

binds the abovementioned sequence to the same action as before. In general, the code 180+*x* corresponds to the sequence ESC followed by the ASCII character of code *x*. Note that some of these sequences may be disabled, if they conflict with existing sequences of your terminal (for instance, ESC followed by '`O`' is always disabled because it prefixes several built-in keyboard sequences).

As a final note, we remark that typing *META-a* on `gnome-terminal` will produce an ESC followed by 'a' ("\x1ba"). Since it is obviously easier to press just META rather than META and CONTROL at the same time, it is a good idea to associate the same sequence also to this combination, using

```
KEY 1E1 MoveSOF
```

Moreover, this setting provides the user with a second choice: one can press ESCAPE followed by a letter instead of using modifiers.

This is the approach used by default in `ne`: this way, CONTROL with META plus a letter should always work, and META should work sometimes (of course, if you're sure to use always the same kind of emulator you can bind more features). Again, the best place to look at it's '`default.keys`'.

As stated above, each line starting with '`SEQ`' binds a character sequence to a key code. The format for a '`SEQ`' binding is

```
SEQ "sequence" hexcode
```

where "*sequence*" is a double-quoted string of characters (which can include escaped hexadecimals) followed by a hexadecimal key code as described above for '`KEY`' definitions.

You should rarely need this, as properly configured systems already do this for most keys. However, some key combinations (CONTROL in conjunction with cursor keys for example) are usually not defined. If you know the character sequence your system generates for such a combination, you may use '`SEQ`' to bind that sequence to a particular key code if that sequence isn't already defined on your system. For example, CONTROL-"cursor-left" may generate the sequence `\x1b[1;5D`. The following lines bind that sequence to the F10 key code '`14A`', then bind that key code to the '`HELP`' command.

```
SEQ  "\x1b[1;5D"  14A
KEY  14A          HELP
```

Sequences are inherently terminal- or terminal emulator-specific, so their utility will vary depending on how many emulators you use. At least they give you the posibility to use keys or key combinations that aren't covered by `curses`.

The key binding file is parsed at startup. If something does not work, `ne` exits displaying an error message. If you want `ne` to skip parsing the key binding file (for instance, to correct the broken file), just give `ne` the `--no-config` argument. See Section 3.1 [Arguments], page 11.

5.2 Changing Menus

`ne` allows you to change the contents of its menus. To accomplish this task, you have to create a file named '`.menus`' in your home directory, or in '`~/.ne`'. You can change the default name (possibly specifying a complete path) using the `--menus` argument (see Section 3.1 [Arguments], page 11).

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 "File"
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 accommodated, but you should consider that many terminals are 80 columns 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 55). Please do not say things in the menus that 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 `ne` to skip the menu configuration phase (for instance, to correct the broken file), just give `ne` the `--no-config` argument. See Section 3.1 [Arguments], page 11.

6 Hints and Tricks

Use F1 or ESCAPE-ESCAPE, 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 4.9.12 [EscapeTime], page 41). This means that you will experience annoying delays when using menus. If you have no F1 key, use ESCAPE-ESCAPE, or redefine a keystroke assigning the command `Escape`, and you will be able to use that keystroke instead of ESCAPE. Unfortunately, some GUI-based terminals (most notably, `gnome-terminal`) use F1 for their own purposes; in that case, you can assign the `Escape` command to another key (see Chapter 5 [Configuration], page 55).

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 several other shortcuts. If the built-in META bindings do not work, you must discover which is the effect of the META in your terminal emulator. Indeed, it is possible in theory to configure about 150 shortcuts. See Chapter 5 [Configuration], page 55. In any case, prefixing a key with ESCAPE has the same effect as holding down META, so with the standard key bindings you can, for instance, advance by word with ESCAPE followed by *f*.

Mac users should turn on "Delete sends CTRL-H" in the `Terminal` *settings.*

If you are a Mac user, you need to check the "Delete sends CTRL-H" option in the '`Advanced`' tab of the `Terminal` application settings.

`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`*`/`' to specify the home directory of another user.

It is easy to correct bad colors.

Sometimes, due to different opinions about the best default foreground and background colors, some of the color choices in a syntax file might be unreadable (for instance, '`dim white`' on a terminal with a white background). Just copy the guilty syntax specification file to the '`~/.ne/syntax`' directory, and change the color names at the start of the file.

Use the '`tabs`' *syntax to distinguish* TAB*s from* SPACE*s.*

When you're struggling to clean up a mix of TABs and SPACEs, temporarily switching to the '`tabs`' syntax may help. The command `Syntax tabs` makes TAB characters show up in a different background color from SPACEs. Once you've gotten your white space issues straightened out, you can switch back to the syntax appropriate for your current file type.

`ne` *does interactive filename completion.*

When you have to specify a file name as last element of a long input, you can invoke the completer using TAB. If you hit it twice in a row, you will enter the file requester, where you can navigate and escape back to the command line, either with F1, which will let you edit again your previous input, or with TAB, which will copy your current selection over your previous file name. In other words, you can freely alternate completion, editing and browsing.

Disable the status line for slow connections.

`ne` tries to emit as few characters as possible when updating the screen. However, for each key you type it is likely that the status bar has to be updated. If your connection is very slow, you can disable the status bar to get a quicker response (see Section 4.9.9 [StatusBar], page 40).

The ESCAPE delay when activating menus can be avoided.

If you press after ESCAPE any key that 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 to speed up the menu activation (note that '`:`' would not work, because it would activate the command line). Alternatively, you can just type ESCAPE twice in a row.

Use turbo mode for lengthy operations.

Turbo mode (see Section 4.9.17 [Turbo], page 42) allows performing very complex operations without updating the screen until the operations are complete. This can be a major plus if you are editing very long files, or if your terminal is slow. If the default value (0, which means twice the number of visible rows) 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: in particular, if you use them on a UTF-8 text, `ne` has to transform them into an equivalent (but more complex) expression that cannot match partially a UTF-8 sequence, and this expansion makes the search even slower.

Use the correct 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 that, when executed, produces the current configuration. However, you could want, for instance, to never change the insert/overwrite state. In this case, just edit the autoprefs files with `ne` and delete the line containing the command setting the insert flag. When the autoprefs are loaded later, 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 that 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`.

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 his knowledge of the limitations.

The design goal of `ne` was to write an editor that is easy to use at first sight, powerful, and completely configurable. Making `ne` run on any terminal that `vi` could handle was also a basic issue, because there is no use getting accustomed to a new tool if you cannot use it when you really need it. Finally, using resources sparingly was considered essential.

`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 user interface. The same can be said about the input line, because it is always preceded by a (possibly highlighted) prompt ending with a colon.

`ne` has no sophisticated visual updating system similar to, for instance, the one of `curses`. All updating is done while manipulating the text, and only if the turbo flag is set can some iterated operations delay the update. (In this case, `ne` keeps track in a very rough way of the part of the screen that 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 reasonably fast connections, the responsiveness of the editor is greatly enhanced by the direct update. And since we update the screen in parallel with the internal representation, we can exploit our knowledge to output a very small number of characters per modification. As it is typical in `ne`, when such design tradeoffs arise, preference is given to the solution that 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 text. There is a doubly linked list of line descriptors that 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 contains a null, so there is no need for a list of free chunks. The point is that the free characters lying 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 its league. No attempt is made to support ten thousand 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 (C90) compiler is assumed.

If the system has a `terminfo` database and the related functions (which are usually contained in `curses` library), `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) that `termcap` lacks. If `terminfo` is not available, `ne` can use a `termcap` database, or, as a last resort, a built-in set of ANSI control sequences. Some details about this can be found in Chapter 10 [Portability Problems], page 67.

`ne` does not allow redefinition of the ESCAPE, TAB or RETURN keys, nor of the interrupt character *CONTROL-*`\`. This decision has been made 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 activating 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 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 ':', or by ESCAPE, again. See Chapter 6 [Hints and Tricks], page 59.

Note that, as has been stated several times, the custom key bindings also work when doing a long input, navigating through the menus or browsing the requester. However, this is only partially true. To keep the code size and complexity down, 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 restrictions in its terminal handling. It does not support highlighting on terminals that use a magic cookie. Supporting such terminals correctly 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 that 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: macros are simply scripts that can be played back automatically. This makes them very useful for everyday use in a learn/play context, but rather inflexible for extending the capabilities of the editor. However, it is not reasonable to incorporate in an editor an interpreter for a custom language. Rather, a system-wide macro language should control the editor *via* interprocess communication. This is the way of the REXX language, and it is likely that future versions of ne will support optionally macros written in REXX.

ne has been written with sparing resource use as a basic goal. Every possible effort has been made to reduce the use of CPU time and memory, the number of system calls, and the number of characters output to the terminal. 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. The update of files using syntax highlighting is as lazy as possible: modifications cause just the update of the current line, and the rest of the screen is updated only when you move away. The search algorithm is a simplified version of the Boyer-Moore algorithm that provides high performance with a minimal setup time. An effort has been taken to move to the text segment all data that do not change during the program execution. When the status bar is switched off, additional optimizations reduce the cursor movement to a minimum.

A word should be said about lists. Clearly, handling the text as a single block with an insertion gap (a la emacs) allows you to gain some memory. 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.

In practice, ne occupies less memory than any memory-based editor we are aware of. (Of course, this does not take into account some sophisticated features of ne, such as unlimited undo/redo, which can cause major memory consumption.)

8 The Encoding Mess

The original `ne` handled 8-bit text files, and assumed that every byte coming from the keyboard could be output to the terminal. No other assumption was made—for instance, the up/down casing functions did not assume a particular encoding for non-US-ASCII characters. This choice had a significant advantage: `ne` could handle easily several different encodings, with minor nuisances for the end user.

Since version 1.30, `ne` supports UTF-8. It can use UTF-8 for its input/output, and it can also interpret one or more buffers as containing UTF-8 encoded text, acting accordingly. Note that the buffer content is actual UTF-8 text—`ne` does not use wide characters. As a positive side-effect, `ne` can support fully the ISO-10646 standard, but nonetheless non-UTF-8 texts occupy exactly one byte per character.

More precisely, *any* piece of text in `ne` is classified as US-ASCII, 8-bit or UTF-8. A US-ASCII text contains only US-ASCII characters. An 8-bit text sports a one-to-one correspondence between characters and bytes, whereas an UTF-8 text is interpreted in UTF-8. Of course, this rises a difficult question: *when* should a buffer be classified as UTF-8?

Character encodings are a mess. There is nothing we can do to change this fact, as character encodings are *metadata that modify data semantics*. The same file may represent different texts of different lengths when interpreted with different encodings. Thus, there is no safe way of guessing the encoding of a file.

`ne` stays on the safe side: it will never try to convert a file from an encoding to another one. It can, however, interpret data contained in a buffer depending on an encoding: in other words, encodings are truly treated as metadata. You can switch off UTF-8 at any time, and see the same buffer as a standard 8-bit file.

Moreover, `ne` uses a *lazy* approach to the problem: first of all, unless the UTF-8 automatic detection flag is set (see Section 4.9.32 [UTF8Auto], page 46), no attempt is ever made to consider a file as UTF-8 encoded. Every file, clip, command line, etc., is firstly scanned for non-US-ASCII characters: if it is entirely made of US-ASCII characters, it is classified as US-ASCII. An US-ASCII piece of text is compatible with anything else—it may be pasted in any buffer, or, if it is a buffer, it may accept any form of text. Buffers classified as US-ASCII are distinguished by an '`A`' on the status bar.

As soon as a user action forces a choice of encoding (e.g., an accented character is typed, or an UTF-8-encoded clip is pasted), `ne` fixes the mode to 8-bit or UTF-8 (when there is a choice, this depends on the value of the Section 4.9.32 [UTF8Auto], page 46 flag). Of course, in some cases this may be impossible, and in that case an error will be reported.

All this happens behind the scenes, and it is designed so that in 99% of the cases there is no need to think of encodings. In any case, should `ne`'s behaviour not match your needs, you can always change at run time the level of UTF-8 support.

9 History

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 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 it possible to develop on that platform again. For `ne` 1.0, an effort has been made to provide a `terminfo` emulation using GNU's `termcap`. The development eventually moved to Linux.

Todd Lewis got involved with `ne` when the University of North Carolina's Chapel Hill campus migrated its central research computers from MVS to UNIX in 1995. The readily available UNIX editors had serious weaknesses in their user interfaces, especially from the standpoint of MVS users who were not too excited about having to move their projects to another platform while learning an entirely new suite of tools. `ne` offered an easily understood interface with enough capabilities to keep these new UNIX users productive. Todd installed and has maintained NE at UNC since then, making several improvements to the code to meet his users' needs. In early 1999 his code base and mine were merged to become version 1.17.

Support for syntax highlighting was added in 2009 with code and techniques heavily borrowed from the GNU-licensed editor `joe`, which was written by Joseph H. Allen. Much of the work to incorporate this code into `ne` was undertaken by Daniele Filaretti, an undergraduate student working under the direction of Sebastiano at the Università degli Studi di Milano.

10 Portability Problems

This chapter is devoted to the description of the (hopefully very few) problems that could arise when porting `ne` to other flavors of UN*X.

The fact that only POSIX calls have been used (see Chapter 7 [Motivations and Design], page 61) 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()`, `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 to use them. You just have to compile using one of the options explained in the '`makefile`' and in the '`README`'. Should you need comprehensive information on GNU `termcap`, you can find the distribution files on any `ftp` site that 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 that 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 the 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 print them in reverse video; moreover, all characters between 127 and 160 are visualized as a reversed question mark (this works particularly well with ISO Latin 1, but Windows users might not like it). This behavior can be easily changed by modifying the `out()` function in '`term.c`'.

Note that it is certainly possible that some system features not standardized by POSIX interfere with `ne`'s use of the I/O stream. Such problems should be dealt with locally by using the system facilities rather than by horribly `#ifdef`'ing the source code. An example is given in Chapter 6 [Hints and Tricks], page 59.

11 Acknowledgments

A lot of people contributed to this project. Part of the code comes from `emacs` and `joe`. Many people, in particular at the silab (the Milan University Computer Science Department Laboratory), helped in beta testing the first versions. Daniele Filaretti worked at the integration of syntax-highlighting code from `joe`. John Gabriele suggested several new features and relentlessly tested them.

Comments, complaints, desiderata are welcome.

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

vigna@di.unimi.it


Todd M. Lewis
CB 1150 2210 ITS Franklin
University of North Carolina
Chapel Hill, NC 27599-1150
USA

utoddl@email.unc.edu
```

Command Index

P

Q

R

S

T

U

V

W

Concept Index

S

T

U

V

W

Table of Contents