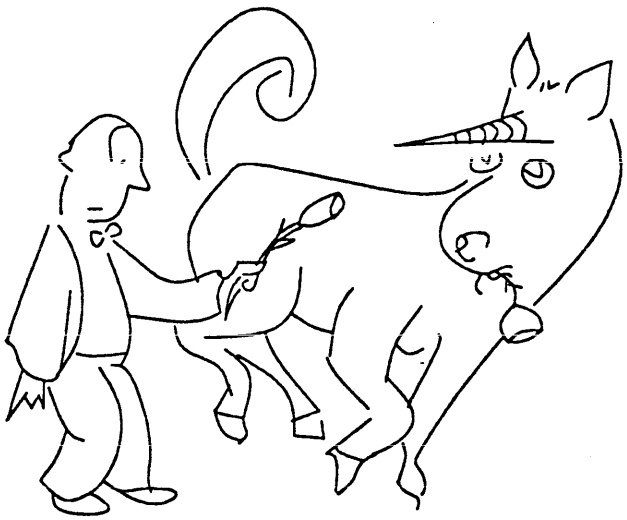

Unix Emacs

James Gosling @ CMU
December, 1981

1. Introduction 3
2. The Screen 3
3. Input Conventions 4
4. Invoking EMACS 4
5. Basic Commands 4
6. Unbound Commands 5
7. Getting Help 5
8. Buffers and Windows 6
9. Compiling programs 6
10. Dealing with collections of files, *tXtE* revisited 7
11. Abbrev mode 7
12. Extensibility 8
12.1. Macros 8
12.2. MLisp -- *Mock Lisp* 9
12.2.1. The syntax of MLisp expressions 9
12.2.2. The evaluation of MLisp expressions 10
12.2.3. MLisp functions 11
12.2.4. Debugging 11
12.3. A Sample MLisp Program 12
12.4. More on Invoking EMACS 14
13. Searching 14
13.1. Simple searches 14
13.2. Regular Expression searches 14

14. Keymaps 16

15. Region Restrictions 18

16. Mode Lines 18

17. Multiple Processes under EMACS 19

17.1. Blocking 22
17.2. Buffer Truncation 22
17.3. Problems 22

18. The EMACS database facility 22

19. Packages 23

19.1. buff -- one-line buffer list 23
19.2. c-mode -- simple assist for C programs 23
19.3. dired -- directory editor 24
19.4. goto -- go to position in buffer 24
19.5. info -- documentation reader 25
19.6. occur -- find occurances of a string 25
19.7. process -- high level process manipulation 25
19.8. pwd -- print and change the working directory 26
19.9. rmail -- a mail management system 26
19.9.1. Sending Mail 26
19.9.2. Reading Mail 27
19.10. scribe -- weak assistance for dealing with Scribe documents 29
19.11. spell -- a simple spelling corrector 30
19.12. tags -- a function tagger and finder 30
19.13. text-mode -- assist for simple text entry 31
19.14. time -- a mode line clock 31
19.15. transp -- transpose words or lines 31
19.16. writeregion -- write region to file 32

20. Command Description 32

21. Options 64

22. Command summary 69

Index 71

1. Introduction

> "What is EMACS? It is a tree falling in the forest with no one to hear it. It is a beautiful flower that smells awful."

This manual attempts to describe the Unix implementation of EMACS, an extensible display editor. It is an *editor* in that it is primarily used for typing in and modifying documents, programs, or anything else that is represented as text. It uses a *display* to interact with the user, always keeping an accurate representation of what is happening visible on the screen that changes in step with the changes made to the document. The feature that distinguishes EMACS from most other editors is its *extensibility*, that is, a user of EMACS can dynamically change EMACS to suit his own tastes and needs.

Calling this editor EMACS is rather presumptuous and even dangerous. There are two major editors called EMACS. The first was written at MIT for their ITS systems as an extension to TECO. This editor is the spiritual father of all the EMACS-like editors; it's principal author was Richard Stallman. The other was also written at MIT, but it was written in MacLisp for Multics by Bernie Greenberg. This editor picks up where ITS EMACS leaves off in terms of its extension facilities. Unix EMACS was called EMACS in the hope that the cries of outrage would be enough to goad the author and others to bring it up to the standards of what has come before.

This manual is organized in a rather haphazard manner. The first several sections were written hastily in an attempt to provide a general introduction to the commands in EMACS and to try to show the method in the madness that is the EMACS command structure. Section 20 (page 32) contains a complete but concise description of all the commands and is in alphabetical order based on the name of the command. Preceding sections generally do not give a complete description of each command, rather they give either the name of the command or the key to which the command is conventionally bound. Section 22 (page 69) lists for each key the command to which it is conventionally bound. The options which may be set with the *set* command are described in section 21, (page 64).

2. The Screen

EMACS divides a screen into several areas called *windows*, at the bottom of the screen there is a one line area that is used for messages and questions from EMACS. Most people will only be using one window, at least until they become more familiar with EMACS. A window is displayed as a set of lines, at the bottom of each window is its *mode line* (For more information on mode lines see section 16, page 18). The lines above the mode line contain an image of the text you are editing in the region around *dot* (or *point*). Dot is the reference around which editing takes place. Dot is a pointer which points at a position *between* two characters. On the screen, the cursor will be positioned on the character that immediatly follows dot. When characters are inserted, they are inserted at the position where dot points; commands exist that delete characters both to the left and to the right of dot. The text on the screen always reflects they way that the text looks *now*.

3. Input Conventions

Throughout this manual, characters which are used as commands are printed in bold face: **X**. They will sometimes have a *control* prefix which is printed as an uparrow character: ↑X is control-X and is typed by holding down the control (often labeled *ctrl* on the top of the key) and simultaneously striking X. Some will have an *escape* (sometimes called *meta*) prefix which is usually printed thus: ESC-X and typed by striking the escape key (often labeled *esc*) then X. And some will have a ↑X prefix which is printed ↑XX which is typed by holding down the control key, striking X, releasing the control key then striking X again.

For example, ESC-↑J is typed by striking ESC then holding down the control key and striking J.

4. Invoking EMACS

EMACS is invoked as a Unix command by typing

```
emacs files
```

to the Shell (the Unix command interpreter). EMACS will start up, editing the named files. You will probably only want to name one file. If you don't specify any names, EMACS will use the same names that it was given the last time that it was invoked. Gory details on the invocation of EMACS can be found in section 12.4, page 14.

5. Basic Commands

Normally each character you type is interpreted individually by EMACS as a command. The instant you type a character the command it represents is performed immediatly.

All of the normal printing characters when struck just insert themselves into the buffer at dot.

To move dot there are several simple commands. ↑F moves dot forward one character. ↑B moves it backward one character. ↑N moves dot to the same column on the next line, ↑P moves it to the same column on the previous line.

String searches may be used to move dot by using the ↑S command to search in the forward direction and ↑R to search in the reverse direction.

Deletions may be performed using ↑H (*backspace*) to delete the character to the left of dot and ↑D to delete the character to the right of dot.

The following table summarizes all of the motion and deletion commands.

	Direction			
	Move		Delete	
Units of Motion	Left	Right	Left	Right
Characters	↑B	↑F	↑H	↑D
Words	ESC-B	ESC-F	ESC-H	ESC-D
Intra line	↑A	↑E		↑K
Inter line	↑P	↑N		

6. Unbound Commands

Even though the number of characters available to use for EMACS commands is large, there are still more commands than characters. You probably wouldn't want to bind them all to keys even if you could. Each command has a long name and by that long name may be bound to a key. For example, ↑F is normally bound to the command named *forward-character* which moves dot forward one character.

There are many commands that are not normally bound to keys. These must be executed with the ESC-X command or by binding them to a key (via the bind-to-key command). Heaven help the twit who rebinds ESC-X.

The ESC-X command will print ": " on the last line of the display and expect you to type in the name of a command. Space and ESC characters may be struck to invoke Tenex style command completion (ie. you type in the first part of the command, hit the space bar, and EMACS will fill in the rest for you -- it will complain if it can't figure out what you're trying to say). If the command requires arguments, they will also be prompted for on the bottom line.

7. Getting Help

EMACS has many commands that let you ask EMACS for help about how to use EMACS. The simplest one is ESC-? (apropos) which asks you for a keyword and then displays a list of those commands whose full name contains the keyword as a substring. For example, to find out which commands are available for dealing with windows, type ESC-?, EMACS will ask "Keyword:" and you reply "window". A list like the following appears:

```
beginning-of-window      ESC-,
delete-other-windows     ↑X1
delete-window            ↑XD
end-of-window            ESC-.
enlarge-window           ↑XZ
line-to-top-of-window    ESC-!
next-window              ↑XN
page-next-window         ESC-↑V
previous-window          ↑XP
shrink-window            ↑X↑Z
split-current-window     ↑X2
```

To get detailed information about some command, the *describe-command* command can be used. It asks for the name of a command, then displays the long documentation for it from the manual. For example, if you wanted more information about the *shrink-window* command, just type "ESC-Xdescribe-command shrink-window" and EMACS will reply:

```
shrink-window                                    ↑X↑Z
    Makes the current window one line shorter, and the window below
    (or the one above if there is no window below) one line taller.
    Can't be used if there is only one window on the screen.
```

If you want to find out what command is bound to a particular key, *describe-key* will do it for you. *Describe-bindings* can be used to make a "wall chart" description of the key bindings in the currently running EMACS, taking into account all of the bindings you have made.

8. Buffers and Windows

There are two fundamental objects in EMACS, *buffers* and *windows*. A buffer is a chunk of text that can be edited, it is often the body of a file. A window is a region on the screen through which a buffer may be viewed. A window looks at one buffer, but a buffer may be on view in several windows. It is often handy to have two windows looking at the same buffer so that you can be looking at two seperate parts of the same file, for example, a set of declarations and a piece of code that uses those declarations. Similarly, it is often handy to have two different buffers on view in two windows.

The commands which deal with windows and buffers are: beginning-of-window (ESC-,), delete-other-windows (↑X1), delete-region-to-buffer (ESC-↑W), delete-window (↑XD), end-of-window (ESC-.), enlarge-window (↑XZ), line-to-top-of-window (ESC-!), list-buffers (↑X↑B), next-window (↑XN), page-next-window (ESC-↑V), previous-window (↑XP), shrink-window (↑X↑Z), split-current-window (↑X2), switch-to-buffer (↑XB), use-old-buffer (↑X↑O) and yank-buffer (ESC-↑Y). See the command description section for more details on each of these.

9. Compiling programs

One of the most powerful features of Unix EMACS is the facility provided for compiling programs and coping with error messages from the compilers. It essential that you understand the standard Unix program *make* (even if you don't use EMACS). This program takes a database (a *makefile*) that describes the relationships among files and how to regenerate (recompile) them. If you have a program that is made up of many little pieces that have to be individually compiled and carefully crafted together into a single executable file, *make* can make your life orders of magnitude easier; it will automatically recompile only those pieces that need to be recompiled and put them together. EMACS has a set of commands that gracefully interact with this facility.

The ↑X↑E (*execute*) command writes all modified buffers and executes the *make* program: The output of *make* will be placed into a buffer called *Error log* which will be visible in some window on the screen. As soon as *make* has finished EMACS parses all of its output to find all the error messages and figure out the files and lines referred to. All of this information is squirreled away for later use by the ↑X↑N command.

The ↑X↑N (*next*) command takes the next error message from the set prepared by ↑X↑E and does three things with it:

- Makes the message itself visible at the top of a window. The buffer will be named *Error log*.
- Does a *visit* (see the ↑X↑V command) on the file in which the error occurred.
- Sets dot to the beginning of the line where the compiler saw the error. This setting of dot takes into account changes to the file that may have been made since the compilation was attempted. EMACS perfectly compensates for any changes that may have been made and always positions the text on the correct line (well, correct as far as the compiler was concerned; the compiler itself may have been a trifle confused about where the error occurred)

If you've seen all the error messages ↑X↑N will say so and do nothing else.

So, the general scenario for dealing with programs is:

- Build a *make* database to describe how your program is to be compiled.
- Compile your program from within EMACS by typing ↑X↑E.
- If there were errors, step through them by typing ↑X↑N, correcting the error, and typing ↑X↑N to get the next.
- When you run out of error messages, type ↑X↑E to try the compilation again.
- When you finally manage to get your beast to compile without any errors, type ↑C to say goodbye to EMACS.
- You'll probably want to use *sdb*, the symbolic debugger, to debug your program.

10. Dealing with collections of files, ↑X↑E revisited

The ↑X↑E command doesn't always execute the *make* program: if it is given a non-zero argument it will prompt for a Unix command line to be executed in place of *make*. All of the other parts of ↑X↑E are unchanged, namely it still writes all modified buffers before executing the command and parses the output of the command execution for line numbers and file names.

This can be used in some very powerful ways. For example, consider the *grep* program. Typing "↑U↑X↑E`grep -n MyProc *.c`ESC" will scan all C programs in the current directory and look for all occurrences of the string "MyProc". After *grep* has finished you can use EMACS (via the ↑X↑N command) to examine and possibly change every instance of the string from a whole collection of files. This makes the task of changing all calls to a particular procedure much easier. **Note:** this only works with the version of *grep* in /usr/jag/bin which has been modified to print line numbers in a format that EMACS can understand.

There are many more uses. The *lint* program, for example. Scribe users might find "`cat MyReport.otl`" to be useful.

A file name/line number pair is just a string embedded someplace in the text of the error log that has the form "FileName, `line` LineNumber". The FileName may or may not be surrounded by quotes ("). The critical component is the string "`, line `" that comes between the file name and the line number. Roll your own file scanning programs, it can make your life much easier.

11. Abbrev mode

Abbrev mode allows the user to type abbreviations into a document and have EMACS automatically expand them. If you have an abbrev called "rhp" that has been defined to expand to the string "rhinocerous party" and have turned on abbrev mode then typing the first non-alphanumeric character after having typed "rhp" causes the string "rhp" to be replaced by "rhinocerous party". The capitalization of the typed in abbreviation controls the capitalization of the expansion: "Rhp" would expand as "Rhinocerous party" and "RHP" would expand as "Rhinocerous Party".

Abbreviations are defined in *abbrev tables*. There is a global abbrev table which is used regardless of which buffer you are in, and a local abbrev table which is selected on a buffer by buffer basis, generally depending on the major mode of the buffer.

Define-global-abbrev takes two arguments: the name of an abbreviation and the phrase that it is to expand to. The abbreviation will be defined in the global abbrev table. Define-local-abbrev is like define-global-abbrev except that it defines the abbreviation in the current local abbrev table.

The use-abbrev-table command is used to select (by name) which abbrev table is to be used locally in this buffer. The same abbrev table may be used in several buffers. The mode packages (like electric-c and text) all set up abbrev tables whose name matches the name of the mode.

The switch *abbrev-mode* must be turned on before EMACS will attempt to expand abbreviations. When abbrev-mode is turned on, the string "abbrev" appears in the mode section of the mode line for the buffer. Use-abbrev-table automatically turns on abbrev-mode if either the global or new local abbrev tables are non-empty.

All abbreviations currently defined can be written out to a file using the write-abbrev-file command. Such a file can be edited (if you wish) and later read back in to define the same abbreviations again. Read-abbrev-file reads in such a file and screams if it cannot be found, quietly-read-abbrev-file doesn't complain (it is primarily for use in startups so that you can load a current-directory dependant abbrev file without worrying about the case where the file doesn't exist).

12. Extensibility

Unix EMACS has two extension features: macros and a built in Lisp system. Macros are used when you have something quick and simple to do, Lisp is used when you want to build something fairly complicated like a new language dependant mode.

12.1. Macros

A *macro* is just a piece of text that EMACS remembers in a special way. When a macro is *executed* the characters that make up the macro are treated as though they had been typed at the keyboard. If you have some common sequence of keystrokes you can define a macro that contains them and instead of retyping them just call the macro. There are two ways of defining macros:

The easiest is called a *keyboard* macro. A keyboard macro is defined by typing the start-remembering command (↑X() then typing the commands which you want to have saved (which will be executed as you type them so that you can make sure that they are right) then typing the stop-remembering command (↑X)). To execute the keyboard macro just type the execute-keyboard-macro command (↑Xe). You can only have one keyboard macro at a time. If you define a new keyboard macro the old keyboard macro vanishes into the mist.

Named macros are slightly more complicated. They have names, just like commands and MLisp functions and can be called by name (or bound to a key). They are defined by using the define-string-macro command (which must be executed by typing ESC-Xdefine-string-macro since it isn't usually bound to a key) which asks

for the name of the macro and it's body. The body is typed in as a string in the prompt area at the bottom the the screen and hence all special characters in it must be quoted by prefixing them with ↑Q. A named macro may be executed by typing ESC-X**name-of-macro** or by binding it to a key with bind-to-key.

The current keyboard macro can be converted into a named macro by using the define-keyboard-macro command which takes a name a defines a macro by that name whose body is the current keyboard macro. The current keyboard macro ceases to exist.

12.2. MLisp -- *Mock Lisp*

Unix EMACS contains an interpreter for a language that in many respects resembles Lisp. The primary (some would say only) resemblance between *Mock Lisp* and any real Lisp is the general syntax of a program, which many feel is Lisp's weakest point. The differences include such things as the lack of a **cons** function and a rather peculiar method of passing parameters.

12.2.1. The syntax of MLisp expressions

There are four basic syntactic entities out of which MLisp expressions are built. The two simplest are integer constants (which are optionally signed strings of digits) and string constants (which are sequences of characters bounded by double quote ["""] characters -- double quotes are included by doubling them: """""" is a one character string. The third are names which are used to refer to things: variables or procedures. These three are all tied together by the use of procedure calls. A procedure call is written as a left parenthesis, "(", a name which refers to the procedure, a list of whitespace separated expressions which serve as arguments, and a closing right parenthesis, ")". An expression is simply one of these four things: an integer constant, a string constant, a name, or a call which may itself be recursivly composed of other expressions.

String constants may contain the usual C excape sequences, "\n" is a newline, "\t" is a tab, "\r" is a carriage return, "\b" is a backspace, "\e" is the escape (033) character, "*nnn*" is the character whose octal representation is *nnn*, and "↑*c*" is the control version of the character *c*.

For example, the following are legal MLisp expressions:

1	The integer constant 1.
"hi"	A two character string constant
"\↑X\↑F"	A two character string constant
"""what?"""	A seven character string constant
(+ 2 2)	An invocation of the "+" function with integer arguments 2 and 2. "+" is the usual addition function. This expression evaluates to the integer 4.
(setq bert (* 4 12))	An invocation of the function *setq* with the variable *bert* as its first argument and and expression that evaluates the product of 4 and 12 as its second argument. The evaluation of this expression assigns the integer 48 to the variable *bert*.
(visit-file "mbox")	An invocation of the function *visit-file* with the string "mbox" as its first argument. Normally the *visit-file* function is tied to the key ↑X↑B. When it is invoked interactively,

either by typing ↑X↑B or ESC-Xvisit-file, it will prompt in the minibuf for the name of the file. When called from MLisp it takes the file name from the parameter list. All of the keyboard-callable function behave this way.

Names may contain virtually any character, except whitespace or parens and they cannot begin with a digit, "'" or "-".

12.2.2. The evaluation of MLisp expressions

Variables must be declared (*bound*) before they can be used. The declare-global command can be used to declare a global variable; a local is declared by listing it at the beginning of a **progn** or a function body (ie. immediatly after the function name or the word **progn** and before the executable statements). For example:

```
(defun
    (foo i
        (setq i 5)
    )
)
```

defines a rather pointless function called *foo* which declares a single local variable *i* and assigns it the value 5. Unlike real Lisp systems, the list of declared variables is not surrounded by parenthesis.

Expressions evaluate to values that are either integers, strings or markers. Integers and strings are converted automaticly from one to the other type as needed: if a function requires an integer parameter you can pass it a string and the characters that make it up will be parsed as an integer; similarly passing an integer where a string is required will cause the integer to be converted. Variables may have either type and their type is decided dynamically when the assignment is made.

Marker values indicate a position in a buffer. They are not a character number. As insertions and deletions are performed in a buffer, markers automatically follow along, maintaining their position. Only the functions *mark* and *dot* return markers; the user may define ones that do and may assign markers to variables. If a marker is used in a context that requires an integer value then the ordinal of the position within the buffer is used; if a marker is used in a context that requires a string value then the name of the marked buffer is used. For example, if **there** has been assigned some marker, then **(pop-to-buffer there)** will pop to the marked buffer. **(goto-character there)** will set dot to the marked position.

A procedure written in MLisp is simply an expression that is bound to a name. Invoking the name causes the associated expression to be evaluated. Invocation may be triggered either by the evaluation of some expression which calls the procedure, by the user typing it's name to the ESC-X command, or by striking a key to which the procedure name has been bound.

All of the commands listed in section 20 (page 32) may be called as MLisp procedures. Any parameters that they normally prompt the user for are taken as string expressions from the argument list in the same order as they are asked for interactivly. For example, the *switch-to-buffer* command, which is normally tied to the ↑XB key, normally prompts for a buffer name and may be called from MLisp like this: (switch-to-buffer *string-expression*).

12.2.3. MLisp functions

An MLisp function is defined by executing the *defun* function. For example:

```
(defun
    (silly
        (insert-string "Silly!")
    )
)
```

defines a function called *silly* which, when invoked, just inserts the string "Silly!" into the current buffer.

MLisp has a rather strange (relative to other languages) parameter passing mechanism. The *arg* function, invoked as (arg *i prompt*) evaluates the *i*th argument of the invoking function if the invoking function was called interactivly or, if the invoking function was not called interactivly, *arg* uses the prompt to ask you for the value. Consider the following function:

```
(defun
    (in-parens
        (insert-string "(")
        (insert-string (arg 1 "String to insert? "))
        (insert-string ")")
    )
)
```

If you type ESC-Xin-parens to invoke *in-parens* interactivly then EMACS will ask in the minibuffer "String to insert? " and then insert the string typed into the current buffer surrounded by parenthesis. If *in-parens* is invoked from an MLisp function by **(in-parens "foo")** then the invocation of *arg* inside *in-parens* will evaluate the expression "foo" and the end result will be that the string "(foo)" will be inserted into the buffer.

The function *interactive* may be used to determine whether or not the invoking function was called interactivly. *Nargs* will return the number of arguments passed to the invoking function.

This parameter passing mechanism may be used to do some primitive language extension. For example, if you wanted a statement that executed a statement *n* times, you could use the following:

```
(defun
    (dotimes n
        (setq n (arg 1))
        (while (> n 0)
            (setq n (- n 1))
            (arg 2)
        )
    )
)
```

Given this, the expression **(dotimes 10 (insert-string "<>"))** will insert the string "<>" 10 times. [*Note*: The prompt argument may be omitted if the function can never be called interactivly].

12.2.4. Debugging

Unfortunatly, debugging MLisp functions is something of a black art. The biggest problem right now is that if an MLisp function goes into an infinite loop there is no way to stop it[1].

There is no breakpoint facility. All that you can do is get a stack trace whenever an error occurs by setting

---

[1] Unless you are running the "interruptable" version of EMACS, in which case typing ↑G sill stop whatever is going on. This version of EMACS has the problem that by typing ↑G you may cause EMACS to lose track of what is on the screen.

the *stack-trace-on-error* variable. With this set, any time that an error occurs a dump of the MLisp execution call stack and some other information is dumped to the "Stack trace" buffer.

12.3. A Sample MLisp Program

The following piece of MLisp code is the Scribe mode package. Other implementations of EMACS, on ITS and on Multics have *modes* that influence the behaviour of EMACS on a file. This behaviour is usually some sort of language-specific assistance. In Unix EMACS a *mode* is no more that a set of functions, variables and key-bindings. This mode package is designed to be useful when editing Scribe source files.

```
(defun
```

The apply-look function makes the current word "look" different by changing the font that it is printed in. It positions dot at the beginning of the word so you can see where the change will be made and reads a character from the tty. Then it inserts "@c[" (where c is the character typed) at the front of the word and "]" at the back. Apply-look gets tied to the key ESC-l *so typing* ESC-l i *when the cursor is positioned on the word "begin" will change the word to "@i[begin]".*

```
    (apply-look go-forward
        (save-excursion c
            (if (! (eolp)) (forward-character))
            (setq go-forward -1)
            (backward-word)
            (setq c (get-tty-character))
            (if (> c ' ')
                (progn (insert-character '@')
                    (insert-character c)
                    (insert-character '[')
                    (forward-word)
                    (setq go-forward (dot))
                    (insert-character ']')
                )
            )
        )
        (if (= go-forward (dot)) (forward-character))
    )
)

(defun
```

This function is called to set a buffer into Scribe mode

```
    (scribe-mode
        (remove-all-local-bindings)
```

If the string "LastEditDate= """ exists in the first 2000 characters of the document then the following string constant is changed to the current date. The intent of this is that you should stick at the beginning of your file a line like: "@string(LastEditDate= "Sat Jul 11 17:59:01 1981")". This will automatically get changed each time you edit the file to reflect that last date on which the file was edited.

```
        (if (! buffer-is-modified)
            (save-excursion
                (error-occurred
                    (goto-character 2000)
                    (search-reverse "LastEditDate=""")
                    (search-forward """")
                    (set-mark)
                    (search-forward """")
                    (backward-character)
                    (delete-to-killbuffer)
                    (insert-string (current-time))
                    (setq buffer-is-modified 0)
                )
            )
        )
        (local-bind-to-key "justify-paragraph" "\ej")
        (local-bind-to-key "apply-look" "\el")
        (setq right-margin 77)
        (setq mode-string "Scribe")
        (setq case-fold-search 1)
        (use-syntax-table "text-mode")
        (modify-syntax-entry "w    -'")
        (use-abbrev-table "text-mode")
        (setq left-margin 1)
        (novalue)
    )
)

(novalue)
```

12.4. More on Invoking EMACS

When EMACS is invoked, it does several things that are not of too much interest to the beginning user.

1. EMACS looks for a file called "**.emacs_pro**" in your home directory, if it exists then it is loaded, with the *load* command. This is the mechanism used for user profiles -- in your **emacs_pro** file, place the commands needed to customize EMACS to suit your taste. If a user has not set up an **emacs_pro** file then EMACS will use a site-specific default file for initialization. At CMU this file is named /usr/local/lib/emacs/maclib/profile.ml

2. EMACS will then interprete its command line switches. "-l<filename>" loads the given file (only one may be named), "-e<funcname>" executes the named function (again, only one may be named). -l and -e are executed in that order, after the user profile is read, but before and file visits are done. This is intended to be used along with the csh alias mechanism to allow you to invoke EMACS packages from the shell (that is, assuming that there is anyone out there who still uses the shell for anything other than to run under EMACS!). For example: "**alias rmail emacs -lrmail -ermail-com**" will cause the csh "rmail" command to invoke EMACS running rmail. Exiting rmail will exit EMACS.

3. If neither *argv* nor *argc* have yet been called (eg. by your startup or by the command line named package) then the list of arguments will be considered as file names and will be visited; if there are no arguments then the arguments passed to the last invocation of EMACS will be used.

4. Finally, EMACS invokes it's keyboard command interpreter, and eventually terminates.

13. Searching

EMACS is capable of performing two kinds of searches[2]. There are two parallel sets of searching and replacement commands that differ only in the kind of search performed.

13.1. Simple searches

The commands *search-forward*, *search-reverse*, *query-replace-string* and *replace-string* all do simple searches. That is, the search string that they use is matched directly against successive substrings of the buffer. The characters of the search string have no special meaning. These search forms are the easiest to understand and are what most people will want to use. They are what is conventionally bound to ↑S, ↑R, ESC-Q and ESC-R.

13.2. Regular Expression searches

The commands *re-search-forward*, *re-search-reverse*, *re-query-replace-string*, *re-replace-string* and *looking-at* all do regular expression searches. The search string is interpreted as a regular expression and matched against the buffer according to the following rules:

1. Any character except a special character matches itself. Special characters are '\' '[' '.' and sometimes '↑' '*' '$'.

---

[2] *Regular* and *Vanilla* for those of you with no taste

2. A '.' matches any character except newline.

3. A '\' followed by any character except those mentioned in the following rules matches that character.

4. A '\w' Matches any word character, as defined by the syntax tables.

5. A '\W' Matches any non-word character, as defined by the syntax tables.

6. A '\b' Matches at a boundary between a word and a non-word character, as defined by the syntax tables.

7. A '\B' Matches anywhere but at a boundary between a word and a non-word character, as defined by the syntax tables.

8. A '\`' Matches at the beginning of the buffer.

9. A '\'' Matches at the end of the buffer.

10. A '\<' Matches anywhere before dot.

11. A '\>' Matches anywhere after dot.

12. A '\=' Matches at dot.

13. A nonempty string *s* bracketed "**[** *s* **]**" (or "**[↑** *s* **]**" matches any character in (or not in) *s*. In *s*, '\' has no special meaning, and ']' may only appear as the first letter. A substring *a-b*, with *a* and *b* in ascending ASCII order, stands for the inclusive range of ASCII characters.

14. A '\' followed by a digit *n* matches a copy of the string that the bracketed regular expression beginning with the *n* th '**\(**' matched.

15. A regular expression of one of the preceeding forms followed by '*****' matches a sequence of 0 or more matches of the regular expression.

16. A regular expression, *x*, bracketed "**\(** *x* **\)**" matches what *x* matches.

17. A regular expression of this or one of the preceeding forms, *x*, followed by a regular expression of one of the preceeding forms, *y* matches a match for *x* followed by a match for *y*, with the *x* match being as long as possible while still permitting a *y* match.

18. A regular expression of one of the preceeding forms preceded by '**↑**' (or followed by '**$**'), is constrained to matches that begin at the left (or end at the right) end of a line.

19. A sequence of regular expressions of one of the preceeding forms seperated by '\|'s matches any one of the regular expressions.

20. A regular expression of one of the preceeding forms picks out the longest amongst the leftmost matches if searching forward, rightmost if searching backward.

21. An empty regular expression stands for a copy of the last regular expression encountered.

In addition, in the replacement commands, *re-query-replace-string* and *re-replace-string*, the characters in the replacement string are specially interpreted:

- Any character except a special character is inserted unchanged.
- A '\' followed by any character except a digit causes that character to be inserted unchanged.
- A '\' followed by a digit *n* causes the string matched by the *n*th bracketed expression to be inserted.
- An '&' causes the string matched by the entire search string to be inserted.

The following examples should clear a little of the mud:

Pika Matches the simple string "Pika".

Whiskey.*Jack Matches the string "Whiskey", followed by the longest possible sequence of non-newline characters, followed by the string "Jack". Think of it as finding the first line that contains the string "Whiskey" followed eventually on the same line by the string "Jack"

[a-z][a-z]* Matches a non-null sequence of lower case alphabetics. Using this in the *re-replace-string* command along with the replacement string "**(&)**" will place parenthesis around all sequences of lower case alphabetics.

Guiness\|Bass Matches either the string 'Guiness' or the string 'Bass'.

\Bed\b Matches 'ed' found as the suffix of a word.

\bsilly\W*twit\b
Matches the sequence of words 'silly' and 'twit' seperated by arbitrary punctuation.

14. Keymaps

When a user is typing to EMACS the keystrokes are interpreted using a *keymap*. A keymap is just a table with one entry for each character in the ASCII character set. Each entry either names a function or another keymap. When the user strikes a key, the corresponding keymap entry is examined and the indicated action is performed. If the key is bound to a function, then that function will be invoked. If the key is bound to another keymap then that keymap is used for interpreting the next keystroke.

There is always a global keymap and a local keymap, as keys are read from the keyboard the two trees are traversed in parallel (you can think of keymaps as FSMs, with keystrokes triggering transitions). When either of the traversals reaches a leaf, that function is invoked and interpretation is reset to the roots of the trees.

The root keymaps are selected using the *use-global-map* or *use-local-map* commands. A new empty keymap is created using the *define-keymap* command.

The contents of a keymap can be changed by using the *bind-to-key* and *local-bind-to-key* commands. These

two commands take two arguments: the name of the function to be bound and the keystroke sequence to which it is to be bound. This keystroke sequence is interpreted relative to the current local or global keymaps. For example, **(bind-to-key "define-keymap" "\↑Zd")** binds the *define-keymap* function to the keystroke sequence '↑Z' followed by 'd'.

A named keymap behaves just like a function, it can be bound to a key or executed within an MLisp function. When it is executed from within an MLisp function, it causes the next keystroke to be interpreted relative to that map.

The following sample uses the keymap to partially simulate the *vi* editor. Different keymaps are used to simulate the different modes in *vi*: command mode and insertion mode.

```
(defun
    (insert-before                  ; Enter insertion mode
        (use-global-map "vi-insertion-mode"))

    (insert-after                   ; Also enter insertion mode, but after
                                    ; the current character
        (forward-character)
        (use-global-map "vi-insertion-mode"))

    (exit-insertion                 ; Exit insertion mode and return to
                                    ; command mode
        (use-global-map "vi-command-mode"))

    (replace-one
        (insert-character (get-tty-character))
        (delete-next-character))

    (next-skip
        (beginning-of-line)
        (next-line)
        (skip-white-space))

    (prev-skip
        (beginning-of-line)
        (previous-line)
        (skip-white-space))

    (skip-white-space
        (while (& (! (eolp)) (| (= (following-char) ' ') (= (following-char) '↑i')))
            (forward-character)))

    (vi                             ; Start behaving like vi
        (use-global-map "vi-command-mode"))
)

; setup vi mode tables
(define-keymap "vi-command-mode")
(define-keymap "vi-insertion-mode")

(use-global-map "vi-insertion-mode"); Setup the insertion mode map
(bind-to-key "execute-extended-command" "\↑X")
(progn i
    (setq i ' ')
    (while (< i 0177)
        (bind-to-key "self-insert" i)
        (setq i (+ i 1))))
(bind-to-key "self-insert" "\011")
(bind-to-key "newline" "\015")
(bind-to-key "self-insert" "\012")
(bind-to-key "delete-previous-character" "\010")
```

```
(bind-to-key "delete-previous-character" "\177")
(bind-to-key "exit-insertion" "\033")

(use-global-map "vi-command-mode"); Setup the command mode map
(bind-to-key "execute-extended-command" "\^X")
(bind-to-key "next-line" "\^n")
(bind-to-key "previous-line" "\^p")
(bind-to-key "forward-word" "w")
(bind-to-key "backward-word" "b")
(bind-to-key "search-forward" "/")
(bind-to-key "search-reverse" "?")
(bind-to-key "beginning-of-line" "0")
(bind-to-key "end-of-line" "$")
(bind-to-key "forward-character" " ")
(bind-to-key "backward-character" "\^h")
(bind-to-key "backward-character" "h")
(bind-to-key "insert-after" "a")
(bind-to-key "insert-before" "i")
(bind-to-key "replace-one" "r")
(bind-to-key "next-skip" "+")
(bind-to-key "next-skip" "\^m")
(bind-to-key "prev-skip" "-")
(use-global-map "default-global-keymap")
```

15. Region Restrictions

The portion of the buffer which EMACS considers visible when it performs editing operations may be restricted to some subregion of the whole buffer.

The *narrow-region* command sets the restriction to encompass the region between dot and mark. Text outside this region will henceforth be totally invisible. It won't appear on the screen and it won't be manipulable by any editing commands. It will, however, be read and written by file manipulation commands like *read-file* and *write-current-file*. This can be useful, for instance, when you want to perform a replacement within a few paragraphs: just narrow down to a region enclosing the paragraphs and execute *replace-string*.

The *widen-region* command sets the restriction to encompass the entire buffer. It is usually used after a *narrow-region* to restore EMACS's attention to the whole buffer.

Save-restriction is only useful to people writing MLisp programs. It is used to save the region restriction for the current buffer (and ***only*** the region restriction) during the execution of some subexpression that presumably uses region restrictions. The value of **`(save-excursion expressions...)`** is the value of the last expression evaluated.

16. Mode Lines

A *mode line* is the line of descriptive text that appears just below a window on the screen. It usually provides a description of the state of the buffer and is usually shown in reverse video. The standard mode line shows the name of the buffer, an '*' if the buffer has been modified, the name of the file associated with the buffer, the *mode* of the buffer, the current position of dot within the buffer expressed as a percentage of the buffer size and and indication of the nesting within *recursive-edit*'s which is shown by wrapping the mode line in an appropriate number of '[' ']' pairs.

It is often the case that for some silly or practical reason one wants to alter the layout of the mode line, to show more, less or different information. EMACS has a fairly general facility for doing this. Each buffer has associated with it a format string that describes the layout of the mode line for that buffer whenever it appears in a window. The format string is interpreted in a manner much like the format argument to the C printf subroutine. Unadorned characters appear in the mode line unchanged. The '%' character and the following format designator character cause some special string to appear in the mode line in their place. The format designators are:

b	Inserts the name of the buffer.
f	Inserts the name of the file associated with the buffer.
m	Inserts the value of the buffer-specific variable *mode-string*.
M	Inserts the value of the variable *global-mode-string*.
p	Inserts the position of "dot" as a percentage.
*****	Inserts an '*' if the buffer has been modified.
[	Inserts (recursion-depth) '['s.
]	Inserts (recursion-depth) ']'s.

If a number *n* appears between the '%' and the format designator then the inserted string is constrained to be exactly *n* characters wide. Either by padding or truncating on the right.

At CMU the default mode line is built using the following format:

```
" %[Buffer: %b%* File: %f  %M (%m)  %p%]"
```

The following variables are involved in generating mode lines:

mode-line-format This is the buffer specific variable that provides the format of a buffers mode line.

default-mode-line-format
This is the value to which *mode-line-format* is initialized when a buffer is created.

mode-string This buffer-specific string variable can be inserted into the mode line by using '%m' in the format. This is it's only use by EMACS. Usually, mode packages (like 'lisp-mode' or 'c-mode') put some string into *mode-string* to indicate the **mode** of the buffer. It is the appearance of this piece of descriptive information that gives the mode line its name.

global-mode-string This is similar to *mode-string* except that it is global -- the same string will be inserted into all mode lines by '%M'. It is usually used for information of global interest. For example, the time package puts the current time of day and load average there.

17. Multiple Processes under EMACS

EMACS has the ability to handle multiple interactive subprocesses. The following is a sketchy description of this capability.

In general, you will *not* want to use any of the functions described in the rest of this section. Instead, you should be using one of the supplied packages that invoke them, see 19.7 page 25. For example, the "shell" command provides you with a window into an interactive shell and the "time" package puts the current time and load average (continuously updated) into the mode line.

Multiple interactive processes can be started under EMACS (using "start-process" or "start-filtered-process"). Processes are tied to a buffer at inception and are thereafter known by this buffer name. Input can be sent to a process from the region or a string, and output from processes is normally attached to the end of the process buffer. There is also the ability to have EMACS call an arbitrary MLISP procedure to process the output each time it arrives from a process (see "start-filtered-process").

Many of the procedures dealing with process management use the concept of "current-process" and "active-process". The current-process is usually the most recent process to have been started. Two events can cause the current-process to change:

1. When the present current-process dies, the most recent of the remaining processes is popped up to take its place.

2. The current-process can be explicitly changed using the "change-current-process" command.

The active-process refers to the current-process, unless the current buffer is a live process in which case it refers to the current buffer.

Below is list of the current mlisp procedures for using processes:

active-process **[unbound]**: (active-process) -- Returns the name of the active process as defined in the section describing the process mechanism.

change-current-process **[unbound]**: (change-current-process "process-name") -- Sets the current process to the one named.

continue-process **[unbound]**: (continue-process "process-name") -- Continue a process stopped by *stop-process.*

current-process **[unbound]**: (current-process) -- Returns the name of the current process as defined in the section describing the process mechanism.

eot-process **[unbound]**: (eot-process "process-name") -- Send an EOT to the process.

int-process **[unbound]**: (int-process "process-name") -- Send an interrupt signal to the process.

kill-process **[unbound]**: (kill-process "process-name") -- Send a kill signal to the process.

list-processes **[unbound]**: (list-processes) -- Analagous to "list-buffers". Processes which have died only appear once in this list before completely disappearing.

process-output **[unbound]**: (process-output) -- Can only be called by the *on-output-procedure* to procure the output generated by the process whose name is given by *MPX-process*. Returns the output as a string.

process-status **[unbound]**: (process-status "process-name") -- Returns -1 if "process-name" isn't a process, 0 if the process is stopped, and 1 if the process is running.

quit-process **[unbound]**: (quit-process "process-name") -- Send a quit signal to the process.

region-to-process **[unbound]**: (region-to-process "process-name") -- The region is wrapped up and sent to the process.

Variable *silently-kill-processes*: If ON EMACS will kill processes when it exits *without* asking any questions. Normally, if you have processes running when EMACS exits, the question "You have processes on the prowl, should I hunt them down for you" is asked. (default OFF)

start-filtered-process **[unbound]**: (start-filtered-process "command" "buffer-name" "on-output-procedure") -- Does the same thing as start-process except that things are set up so that "on-output-procedure" is automatically called whenever output has been received from this process. This procedure can access the name of the process producing the output by refering to the variable *MPX-process*, and can retrieve the output itself by calling the procedure *process-output*.

> **The filter procedure must be careful to avoid generating side-effects (eg. *search-forward*). Moreover, if it attempts to go to the terminal for information, output from other processes may be lost.**

start-process **[unbound]**: (start-process "command" "buffer-name") -- The home shell is used to start a process executing the command. This process is tied to the buffer "buffer-name" unless it is null in which case the "Command execution" buffer is used. Output from the process is automatically attached to the end of the buffer. Each time this is done, the mark is left at the end of the output (which is the end of the buffer).

stop-process **[unbound]**: (stop-process "process-name") -- Tell the process to stop by sending it a stop signal. Use *continue-process* to carry on.

string-to-process [unbound]: (string-to-process "process-name" "string") -- The string is sent to the process.

17.1. Blocking

When too many characters are sent to a process in one gulp, the send will be blocked until the process has removed sufficient characters from the buffer. The send will then be automatically continued. Normally this process is invisible to the EMACS user, but if the process has been stopped, the send will not be unblocked and further attempts to send to the process will result in an overwrite error message.

17.2. Buffer Truncation

EMACS does not allow process buffers to grow without bound. When a process buffer exceeds the value of the variable *process-buffer-length*, 500 characters are erased from the beginning of the buffer. The default value for *process-buffer-length* is 10,000.

17.3. Problems

The most obvious problem with allowing multiple interactive processes is that it is too easy to start up useless jobs which drag everyone down. Also when checkpointing is done, all buffers including the process buffers are checkpointed. So if you have a one line buffer keeping time, it will take more system time to checkpoint it than it will to keep it updated once a minute.

In addition to anti-social problems, there are some real bugs remaining:

- Sometimes when starting a process, it will inexplicably expire immediately. This often happens to the first process you fire up.

- Subprocesses are assumed to not want to try fancy things with the terminal. EMACS doesn't know how to handle this and for now more or less ignores stty requests from processes. This means that csh cannot be used from within EMACS. Running chat and ftp can also cause problems. Someday, EMACS should try to handle stty's.

- The worst problem is that background processes started outside EMACS will cause EMACS to hang when they finally finish. This might get fixed if I want to think about it.

- If EMACS does crash or hang, you will find several orphan processes left hanging around. It is best to do a ps and get rid of them.

18. The EMACS database facility

Unix EMACS provides a set of commands for dealing with databases of a rather primitive form. These databases are intended to be used in *help* facilities to find documentation for a given keyword, but they have many other uses: managed mailboxes or nodes in an *info* tree.

A *database* is a set of (key, content) pairs which may be retrieved or stored based on the key. Both the key and the content may be arbitrary strings of characters. The content may be long, but there are restrictions on the aggragate length of the keys.

A *database search list* is a list of databases. When a key is looked up in a database search list the databases in the search list are examined in order for one containing the key. The content corresponding to the first key that matches is returned. When a key is to have its content changed only the first database in the search list is used.

The commands available for dealing with databases are:

extend-database-search-list **[unbound]**: (extend-database-search-list dbname filename) adds the given data base file to the data base search list (dbname). If the database is already in the search list then it is left, otherwise the new database is added at the beginning of the list of databases.

fetch-database-entry **[unbound]**: (fetch-database-entry dbname key) takes the entry in the data base corresponding to the given key and inserts it into the current buffer.

list-databases **[unbound]**: (list-databases) lists all data base search lists.

put-database-entry **[unbound]**: (put-database-entry dbname key) takes the current buffer and stores it into the named database under the given key.

19. Packages

This chapter contains a description of a few of the packages that have been written for EMACS in MLisp. To load some package, just type "ESC-X load *PackageName*". The title of each following section contains the name of the package before the '--'.

19.1. buff -- one-line buffer list

Loading the *buff* package replaces the binding for ↑X-↑B (usually *list-buffers*) with *one-line-buffer-list*.

one-line-buffer-list Gives a one-line buffer list in the mini-buffer. If the buffer list is longer than one line, it will print a line at a time and wait for a character to be typed before moving to the next line. Buffers that have been changed since they were last saved are prefixed with an asterisk (*), buffers with no associated file are prefixed with a hash-mark (#), and empty buffers are prefixed with an at-sign (@).

19.2. capword -- capitalize the current word

The *capword* package defines one function, *capitalize-word*, which is conveniently bound to to ESC-C (although this is *not* done by the package.) *Capitalize-word* is quite similar to *case-word-upper* or *case-word-lower*, except that it first sets the entire current word in lower case, then sets the first letter in upper case. For example, if the current word is "eMaCs", typing ESC-C will change it to "Emacs".

19.3. c-mode -- simple assist for C programs

begin-C-comment (ESC-') Initiates the typing in of a comment. Moves the cursor over to the comment column, inserts "/* " and turns on autofill. If ESC-' is typed in the first column, the the comment begins there, otherwise it begins where ever *comment-column* says it should.

end-C-comment (ESC-') Closes off the current comment.

indent-C-procedure
(ESC-j) Takes the current function (the one in which dot is) and fixes up its indentation by running it through the "indent" program.

19.4. dired -- directory editor

The *dired* package implements the *dired* command which provides some simple convenient directory editing facilities. When you run *dired* it will ask for the name of a directory, displays a listing of it in a buffer, and processes commands to examine files and possibly mark them for deletion. When you're through with *dired* it actually deletes the marked files, after asking for confirmation. The commands it recognizes are:

d	Marks the current file for deletion. A 'D' will appear at the left margin. It does not actually delete the file, it just marks it. The deletion will be performed when *dired* is exited. It also makes the next file be the current one.
u	Removes the deletion mark from the current file. This is the command to use if you change your mind about deleting a file. It also makes the next file be the current one.
RUBOUT	Removes the deletion mark from the line preceeding the current one. If you mark a file for deletion with 'd' the current file will be advanced to the next line. RUBOUT undoes both the advancing and the marking for deletion.
e, v	Examine a file put putting it in another window and doing a recursive-edit on it. To resume *dired* type ↑C.
r	Removes the current file from the directory listing. It doesn't delete the file, it just gets rid of the directory listing entry. Use it to remove some of the clutter on your screen.
q, ↑C	Exits *dired*. For each file that has been marked for deletion you will be asked for confirmation. If you answer 'y' the file will be deleted, otherwise not.
n, ↑N	Moves to the next entry in the directory listing.
p, ↑P	Moves to the previous entry in the directory listing.
↑V	Moves to the next page in the directory listing.
ESC-v	Moves to the previous page in the directory listing.
ESC-<	Moves to the beginning of the directory listing.
ESC->	Moves to the end of the directory listing.

19.5. goto -- go to position in buffer

goto-line — Moves the cursor to beginning of the indicated line. The line number is taken from the prefix argument if it is provided, it is prompted for otherwise. Line numbering starts at 1.

goto-percent — Moves dot to the indicated percentage of the buffer. The percentage is taken from the prefix argument if it is provided, it is prompted for otherwise. **(goto-percent *n*)** goes to the character that is *n%* from the beginning of the buffer.

19.6. info -- documentation reader

This is a Unix Emacs version of the ITS INFO structured documentation reader. Loading this package defines one function, *info*, which when invoked presents you with a menu of topics. Among other things, it documents itself. To find out how to use *info*, type "h" while it is running. (To return to Emacs from *info*, type "q".

If you left *info* while visiting some node, giving the *info* command again (to the same invocation of Emacs) will return you to the node you were at, rather than to the directory node. You can return to the directory node, if you wish, by typing "d".

Loading *info*, and running it the first time, both a slow operations (several seconds on an unloaded system.) Once initialized, however, subsequent calls will go swiftly.

19.7. occur -- find occurances of a string

The *occur* package allows one to find the occurances of a string in a buffer. It contains one function

Occurances — When invoked, prompts with "Search for all occurances of: ". It then lists (in a new buffer) all lines contain the string you type following *dot*. Possible options (listed at the bottom of the screen) allow you to page through the listing buffer or abort the function.

In addition, a global variable controls the action of the function:

&Occurances-Extra-Lines
is a global variable that controls how many extra surrounding lines are printed in addition to the line containing the string found. If this variable is 0 then NO additional lines are printed. If this variable is greater than 0 then it will print that many lines above and below the line on which the string was found. When printing more than one line per match in this fashion, it will also print a seperator of '----------' so you can tell where the different matches begin and end. At the end of the buffer it prints '<<<End of Occur>>>'.

19.8. process -- high level process manipulation

The process package provides high level access to the process control features of Unix EMACS. It allows you to interact with a shell through an EMACS window, just as though you were talking to the shell normally.

shell — The *shell* command is used to either start or reenter a shell process. When the shell command is executed, if a shell process doesn't exist then one is created (running the standard "sh") tied to a buffer named "shell'. In any case, the shell buffer becomes the

current one and dot is positioned at the end of it. In that buffer output from the shell and programs run with it will appear. Anything typed into it will get sent to the subprocess when the *return* key is struck. This lets you interact with a shell using EMACS, and all of it's editing capability, as an intermediary. You can scroll backwards over a session, pick up pieces of text from other places and use them as input, edit while watching the execution of some program, and much more...

lisp — The *lisp* command is exactly the same as the *shell* command except that it starts up "cmulisp" in the "lisp" buffer. You can have both a shell and a lisp process going at the same time. You can even have as many shells going as you want, but this package doesn't support it.

grab-last-line — (ESC-=) This command takes the last string typed as input to the process and brings it back, as though you had typed it again. So if you muff a command, just type ESC-=, edit the line, and hit *return* again.

lisp-kill-output — (↑X↑K) [this only applies to *lisp* processes] Erases the output from the last command. If you don't want to see the output of the last command any more, just type ↑X↑K and it will go away.

pr-newline — (↑M -- return) Takes the text of the current line and sends it as input to the process tied to the current buffer. Actually, if dot is on the last line of the buffer, it takes the region from mark to the end of the buffer and sends it as input (output from a process causes the mark to be set after the inserted text); if dot is not on the last line, just the text of that line is shipped (presuming that your prompt is "$ ").

send-eot — (↑D) If dot is at the end of the buffer, then ↑D behaves just as it does outside of EMACS -- it sends an EOT to the subprocess (end of file to some folks). If dot isn't at the end of the buffer, then it does the usual character deletion.

send-int-signal — (\177 -- rubout) Sends an INT (Interrupt) signal to the subprocess, which should make it stop whatever it is doing.

send-quit-signal — (↑\) Sends a QUIT signal to the subprocess, making it stop whatever it is doing and produce a core dump.

19.9. pwd -- print and change the working directory

pwd — Prints the current working directory in the mode line, just like the shell command "pwd".

cd — Changes the current working directory, just like the shell command "cd". You should beware that *cd* only changes the current directory for EMACS, if it has already spawned a subprocess (a shell, for example) then a *cd* from within EMACS has no effect on the shell.

19.10. rmail -- a mail management system

EMACS may be used to send and receive electronic mail. The *rmail* command (Usually invoked as "ESC-Xrmail") is used for reading mail, *smail* is used for sending mail.

19.10.1. Sending Mail

When sending mail, either by using the *smail* command or from within *rmail*, EMACS constructs a buffer that contains an outline of the message to be sent and allows you to edit it. All that you have to do is fill in the blanks. When you exit from *smail* (by typing ↑C usually -- when you're editing the message body you will be in a recursive-edit) the message will be sent to the destinations and blindcopied to you. Several commands are available to help you in composing the message:

justify-paragraph (ESC-j) Fixes up the line breaks in the current paragraph according to the current left and right margins.

exit-emacs (↑C) Exits mail composition and attempts to send the mail. If all goes well the mail composition window will disappear and a confirmation message will appear at the bottom of the screen. If there is some sort of delivery error you will be placed back into the composition window and a message will appear. **Bug:** when delivery is attempted and there are errors in the delivery, the message will have been delivered to the acceptable addresses and not to the others. This makes retrying the message difficult since you have to manually eliminate the addresses to which the message has already been sent.

mail-abort-send (↑X↑A) Aborts the message. If you're part-way through composing a message and decide that you don't want to send it, ↑X↑A will throw it away, after asking for confirmation.

mail-noblind-exit (↑X↑C) Exits *smail* and send the message, just as ↑C will, except that a blind copy of the message will not be kept.

exit-emacs (↑X↑F) Same as ↑C.

exit-emacs (↑X↑S) Same as ↑C.

mail-append (↑Xa) Positions dot at the end of the body and sets margins and abbrev tables appropriatly.

mail-cc (↑Xc) Positions dot to the "cc:" field, creating it if necessary.

mail-insert (↑Xi) Inserts the body of the message that was most recently looked at with rmail into the body of the message being composed. If, for instance, what you want to do is forward a message to someone, just read the message with rmail, then compose a message to the person you want to forward to, and type ↑Xi.

mail-subject (↑Xs) Positions dot to the "subject:" field of the message.

mail-to (↑Xt) Positions dot to the "to:" field of the message.

19.10.2. Reading Mail

The *rmail* command provides a facility for reading mail from within EMACS. When it is running there are usually two windows on the screen: one shows a summary of all the messages in your mailbox and the other displays the "current" message. The summary window may contain something like this:

```
  02621525335022 29 Oct 1981  research!dmr     [empty]
B 02621525335030 29 Oct 1981  =>Unix-Wizards  A plea for understanding
  02621525335040 31 Oct 1981  CSVAX.dmr        rc etymology
  02621525335072 3 Nov 1981   EHF              fyi
```

```
 A  02621352421000 3 Nov 1981   JIM              copyrights
  B 02621353040000 3 Nov 1981   =>JIM            Re: copyrights
    02621646433000 [empty]      [empty]          [empty]
  B 02621647417000 4 Nov 1981   =>research!ikey Emacs
>N  02622024522003 5 November   flaco            cooking class
```

This is broken into five columns, as indicated by the underlining.

- The first column contains some flags: '>' indicates the current message, 'B' indicates that the message is a blindcopy (ie. A copy of a message that you sent to someone else), 'A' indicates that you've answered the message, and 'N' indicates that the message is new.
- The second column contains a long string of digits that is internal information for the mail system.
- The third contains the date on which the mail was sent.
- The forth contains the sender of the message, unless it is a blindcopy, in which case it contains the destination (indicated by the "=>").
- The fifth column contains the subject of the message.

When in the summary window *Rmail* responds to the following commands:

rmail-shell (!) Puts you into a command shell so that you can execute Unix commands. Resume mail reading by typing ↑C.

execute-extended-command
(:) An emergency trap-door for executing arbitrary EMACS commands. You should never need this.

rmail-first-message
(<) Look at the first message in the message file.

rmail-last-message (>) Look at the last message in the message file.

rmail-help (?) Print a very brief help message

exit-emacs (↑C) Leave rmail. Changes marked in the message file directory (eg. deletions) will be made.

rmail-search-reverse
(↑R) Prompts for a search string and positions at the first message, scanning in reverse, whose directory entry contains the string.

rmail-search-forward
(↑S) Prompts for a search string and positions at the first message, scanning forward, whose directory entry contains the string.

rmail-append (a) Append the current message to a file.

rmail-previous-page
(b) Moves backward in the window that contains the current message.

rmail-delete-message
(d) Flag the current message for deletion. It won't actually be deleted until you leave rmail.

rmail-next-page (f) Moves forward in the window that contains the current message. To read a message that is longer than the window that contains it, just keep typing f and rmail will show you successive pages of it.

rmail-goto-message
(g) Moves to the *n*th message.

smail (m) Lets you send some mail.

rmail-next-message
(n) Moves to the next message.

rmail-previous-message
(p) Moves to the previous message.

exit-emacs (q) the same as ↑C

rmail-reply (r) Constructs a reply to the current message.

rmail-skip (s) Moves to the *n*th message relative to this one.

rmail-undelete-message
(u) If the current message was marked for deletion, u removes that mark.

19.11. scribe -- weak assistance for dealing with Scribe documents

Scribe mode binds *justify-paragraph* to ESC-j, defines *apply-look* and binds it to ESC-l, turns on autofill, sets the right margin to 77 and updates the LastEditDate to the current date. It also binds *index-entry* to ESC-I, and *scribe-command* to ESC-S.

If the string "LastEditDate="" exists somewhere in the first 2000 characters of the document then then the region extending from it to the next "" is replaced by the current date and time. You're intended to stick in your document something like:

```
@String(LastEditDate="Sat Nov 28 11:17:29 1981")
```

EMACS will automatically maintain the date. The date will only change in the file you make some changes, the mere act of starting scribe-mode does not cause the date change to be permanent.

Apply-look reads a single character and then surrounds the current word with "@*c*[" and "]". So, if you've just typed "begin", typing ESC-l-i will change it to "@i[begin]", which appears in the document as "*begin*". This use of the word "look" comes from the *Bravo* text editor.

Index-entry takes a number of words and creates a Scribe index entry for that phrase, on a separate line. The current dot and mark are not modified. If the command is given with no prefix-argument, the current word is used as the index item. If a positive argument *n* is given, *n* words starting with the current word are used as the index phrase; a negative argument *n* causes the *n* words ending with the current word to be used. The easiest way to learn what the real rules are is to try it out; if you make a mistake, you can try again

without having to change the cursor position, then delete the wrong index entries once you've got a right one.

Scribe-command is used to create a *Begin -- End* bracket pair for a specified scribe command. You are prompted for the name of the command (e.g., **Index, Itemize, Description**, etc.) For example, **ESC-S Itemize** would insert

```
@Begin(Itemize)

@End(Itemize)
```

and would leave the cursor on the blank line inside the begin--end brackets. If you always create scribe commands in this way, you'll never have unbalanced begin--ends in your scribe files.

19.12. spell -- a simple spelling corrector

The spell package implements the single function *spell*. It provides a simple facility for doing spelling correction. If you invoke *spell* it will scan your file looking for spelling errors, then it will go through a dialogue to let you fix them up. For each misspelled word EMACS will show you the word, some context around it and ask you what to do. If you type 'e' or '↑G' the spelling corrector will exit. If you type ' ' it will ignore the word. If you type 'r' it will ask for the text to use in replacing the word and perform a query-replace. ***Bug:*** This uses the Unix *spell* command which believes that its input is a source for the Unix standard text formatter troff/nroff; Spell misbehaves on Scribe .mss files.

19.13. tags -- a function tagger and finder

The tags package closely resembles the tags package found in Twenex EMACS. The database used by the tag package (called a tagfile) correlates function definitions to the file in which the definitions appear. The primary function of the tag package is to allow the user to specify the name of a function, and then have EMACS locate the definition of that function. The commands implemented are:

add-tag — Adds the current line (it should be the definition line for some function) to the current tagfile.

goto-tag — *goto-tag* takes a single string argument which is usually the name of a function and visits the file containing that function with the first line of the function at the top of the window. The string may actually be a substring of the function name (actually, any substring of the first line of the function definition). If *goto-tag* is given a numeric argument then rather than asking for a new string it will use the old string and search for the next occurrence of that string in the tagfile. This is used for stepping through a set of tags that contain the same string.

This is the most commonly used command in the tag package so it is often bound to a key: Twenex EMACS binds it to ESC-., but the Unix tag package doesn't bind it to anything, it presumes that the user will bind it (I use ↑X↑G).

make-tag-table — Takes a list of file names (with wildcards allowed) and builds a tagfile for all the functions in all of the files. It determines the language of the contents of the file from the extension. This command may take a while on large directories, be prepared to wait. A common use is to type "make-tag-table *.c".

recompute-all-tags — Goes through your current tag file and for each file mentioned refinds all of the tags. This

is used to rebuild an entire tag file if you've made very extensive changes to the files mentioned and the tag package is no longer able to find functions. The tagfile contains *hints* to help the system locate the tagged function, as you make changes to the various files the hints become out of date. Periodically (no too often!) you should recompute the tagfile.

visit-function — Takes the function name at or before dot, does a *goto-tag* on that name, then puts you into a recursive-edit to look at the function definition. To get back to where you were, just type ↑C. This is used when you're editing something, have dot positioned at some function invocation, then want to look at the function.

visit-tag-table — Normally the name of the tagfile is ".tags" in the current directory. If you want to use some other tagfile, visit-tag-table lets you do that.

19.14. text-mode -- assist for simple text entry

Implements the *text-mode* command which ties ESC-j to *justify-paragraph* and sets up autofill with a left margin of 1 and a right margin of 77.

19.15. time -- a mode line clock

This package only implements one user-visible function, *time*, which puts the current time of day and load average (continuously updating!) in the mode line of each window. It uses global-mode-string and the subprocess control facility. Major!

19.16. transp -- transpose words or lines

The *transp* package allows transposition of word and lines (similar to the function of *transpose-character*.)

transpose-word — Takes the two words preceding *dot* and exchanges them. (If *dot* is within a word, it is counted as preceeding *dot*.)

transpose-line — Takes the two lines preceding *dot* and exchanges them. (If *dot* is within a line, it is counted as preceeding *dot*.)

There are also several global variables to control the *transpose-line* function:

&Default-Transpose-Direction
(default 1) Tells transpose-line which other line to transpose with the current on. If this is set to 1 (actually your favorite non-zero number will do) then transpose-line will use the line above the current one and if it is 0 transpose-line will use the line below the current one.

&Default-Transpose-Follow
(default 0) If this is set Non-zero it will cause transpose-line to leave the cursor(dot) on the line that got transposed, and if this is set to Zero it will stay at the same place in the file!

&Default-Transpose-Magic
(default 0) This variable controls some magic inside the transpose Line function. If it is set to zero, transpose-line will behave as controlled by the settings of the above variables. If

this is set Non-Zero then the magic is controlled by the cursor position when transpose-line is invoked. If the cursor(dot) is somewhere in the middle of a line, then it behaves as if this variable were 0. If the cursor is at the end of a line, or at the beginning of a line, the magic will happen. If the cursor is at the beginning of the line transpose-line will override the above variable settings and assert that you want to transpose with the above line and that you want to follow the line you were on. If the cursor is at the end of a line transpose-line will assume that you want to transpose with the next line and that you want to follow the line you were on. The main reason for this magic is so that you can blip lines up and down in your buffer real easily.

19.17. writeregion -- write region to file

This package only implements one function, write-region-to-file, which takes the region between dot and mark and writes it to the named file.

20. Command Description

This chapter describes (in alphabetical order) all of the commands which are defined in the basic Unix EMACS system. Other commands may be defined by loading packages. Each description names the command and indicates the default binding.

! **[unbound]**

(! e_1) MLisp function that returns not e_1.

!= **[unbound]**

(!= e_1 e_2) MLisp function that returns true iff e_1 != e_2.

% **[unbound]**

(% e_1 e_2) MLisp function that returns e_1 % e_2 (the C mod operator).

& **[unbound]**

(& e_1 e_2) MLisp function that returns e_1 & e_2.

*** **[unbound]**

(* e_1 e_2) MLisp function that returns e_1 * e_2.

+ **[unbound]**

(+ e_1 e_2) MLisp function that returns e_1 + e_2.

-
(- e_1 e_2) MLisp function that returns e_1 - e_2. [unbound]

/
(/ e_1 e_2) MLisp function that returns e_1 / e_2. [unbound]

<
(< e_1 e_2) MLisp function that returns true iff e_1 < e_2. [unbound]

<<
(<< e_1 e_2) MLisp function that returns e_1 << e_2 (the C shift left operator). [unbound]

<=
(<= e_1 e_2) MLisp function that returns true iff e_1 <= e_2. [unbound]

=
(= e_1 e_2) MLisp function that returns true iff e_1 = e_2. [unbound]

>
(> e_1 e_2) MLisp function that returns true iff e_1 > e_2. [unbound]

>=
(>= e_1 e_2) MLisp function that returns true iff e_1 >= e_2. [unbound]

>>
(>> e_1 e_2) MLisp function that returns e_1 >> e_2 (the C shift right operator). [unbound]

↑
(↑ e_1 e_2) MLisp function that returns e_1 ↑ e_2 (the C XOR operator). [unbound]

abort-operation ↑G

EMACS gives up on what it is trying to do now and goes back to standard input mode. Rings the bell. Use ↑G whenever EMACS is in a state you don't like, for example, asking you for a string to be searched for when you decide that you don't want to search for a string.

active-process [unbound]

(active-process) -- Returns the name of the active process as defined in the section describing the process mechanism.

append-region-to-buffer [unbound]

Appends the region between dot and mark to the named buffer. Neither the original text in the destination buffer nor the text in the region between dot and mark will be disturbed.

append-to-file [unbound]

Takes the contents of the current buffer and appends it to the named file. If the files doesn't exist it will be created.

apropos ESC-?

Prompts for a keyword and then prints a list of those commands whose short description contains that keyword. For example, if you forget which commands deal with windows, just type "ESC-?**window**ESC".

arg [unbound]

(arg i [prompt]) evaluates to the i'th argument of the invoking function or prompts for it if called interactively [the prompt is optional, if it is omitted, the function cannot be called interactivly]. For example,

```
(arg 1 "Enter a number: ")
```

Evaluates to the value of the first argument of the current function, if the current function was called from MLisp. If it was called interactively then it is prompted for. As another example, given:

```
(defun (foo (+ (arg 1 "Number to increment? ") 1)))
```

then (foo 10) returns 11, but typing "ESC-Xfoo" causes emacs to ask "Number to increment? ". Language purists will no doubt cringe at this rather primitive parameter mechanism, but what-the-hell... it's amazingly powerful.

argc [unbound]

Is an MLisp function that returns the number of arguments that were passed to EMACS when it was invoked from the Unix shell. If either *argc* or *argv* are called early enough then EMACS's startup action of visiting the files named on the command line is suppressed.

argument-prefix ↑U

When followed by a string of digits ↑U causes that string of digits to be interpreted as a numeric argument which is generally a repetition count for the following command. For example, ↑U10↑N moves down 10 lines (the 10'th next). A string of *n* ↑U's followed by a command provides an argument to that command of 4^n. For example, ↑U↑N moves down four lines, and ↑U↑U↑N moves down 16. Argument-prefix should *never* be called from an MLisp function.

argv **[unbound]**

(argv *i*) returns the *i*th argument that was passed to EMACS when it was invoked from the Unix Shell. If EMACS were invoked as "emacs blatto" then (argv 1) would return the string "blatto". If either argc or argv are called early enough then EMACS's startup action of visiting the files named on the command line is suppressed.

auto-execute **[unbound]**

Prompt for and remember a command name and a file name pattern. When a file is read in via *visit-file* or *read-file* whose name matches the given pattern the given command will be executed. The command is generally one which sets the mode for the buffer. Patterns must be of the form "*string" or "string*": "*string" matches any filename whose suffix is "string"; "string*" matches any filename prefixed by "string". For example, **auto-execute c-mode *.c** will put EMACS into C mode for all files with the extension ".c".

autoload **[unbound]**

(autoload command file) defines the associated *command* to be autoloaded from the named *file*. When an attempt to execute the command is encountered, the file is loaded and then the execution is attempted again. the loading of the file must have redefined the command. Autoloading is useful when you have some command written in MLisp but you don't want to have the code loaded in unless it is actually needed. For example, if you have a function named box-it in a file named box-it.ml, then the command **(autoload "box-it" "box-it.ml")** will define the box-it command, but won't load its definition from box-it.ml. The loading will happen when you try to execute the box-it command.

backward-balanced-paren-line **[unbound]**

Moves dot backward until either

- The beginning of the buffer is reached.
- An unmatched open parenthesis, '(', is encountered. That is, unmatched between there and the starting position of dot.
- The beginning of a line is encountered at "parenthesis level zero". That is, without an unmatched ')' existing between there and the starting position of dot.

The definitions of parenthesis and strings from the syntax table for the current buffer are used.

backward-character **↑B**

Move dot backwards one character. Ends-of-lines and tabs each count as one character. You can't move back to before the beginning of the buffer.

backward-paragraph ESC-[

Moves to the beginning of the current or previous paragraph. Blank lines, and Scribe and nroff command lines separate paragraphs and are not parts of paragraphs.

backward-paren **[unbound]**

Moves dot backward until an unmatched open parenthesis, '(', or the beginning of the buffer is found. This can be used to aid in skipping over Lisp S-expressions. The definitions of parenthesis and strings from the syntax table for the current buffer are used.

backward-sentence ESC-A

Move dot backward to the beginning of the preceeding sentence; if dot is in the middle of a sentence, move to the beginning of the current sentence. Sentences are seperated by a '.', '?' or '!' followed by whitespace.

backward-word ESC-B

If in the middle of a word, go to the beginning of that word, otherwise go to the beginning of the preceding word. A word is a sequence of alphanumerics.

baud-rate **[unbound]**

An MLisp function that returns what EMACS thinks is the baud rate of the communication line to the terminal. The baud rate is (usually) 10 times the number of characters transmitted per second. (Baud-rate) can be used for such things as conditionally setting the display-file-percentage variable in your EMACS profile: (setq display-file-percentage (> (baud-rate) 600))

beginning-of-file ESC-<

Move dot to just before the first character of the current buffer.

beginning-of-line ↑A

Move dot to the beginning of the line in the current buffer that contains dot; that is, to just after the preceeding end-of-line or the beginning of the buffer.

beginning-of-window ESC-,

Move dot to just in front of the first character of the first line displayed in the current window.

bind-to-key **[unbound]**

Bind a named macro or procedure to a given key. All future hits on the key will cause the named macro or procedure to be called. The key may be a control key, and it may be prefixed by ↑X or ESC. For example, if you want ESC-= to behave the way ESC-X*print* does, then typing ESC-X*bind-to-key print* ESC-= will do it.

bobp [unbound]

(bobp) is an MLisp predicate which is true iff dot is at the beginning of the buffer.

bolp [unbound]

(bolp) is an MLisp predicate which is true iff dot is at the beginning of a line.

buffer-size [unbound]

(buffer-size) is an MLisp function that returns the number of characters in the current buffer.

c-mode [unbound]

Incompletely implemented.

c= [unbound]

(c= e_1 e_2) MLisp function that returns true iff e_1 is equal to e_2 taking into account the character translations indicated by case-fold-search and word-mode-search. If word-mode-search is in effect, then upper case letters are "c=" to their lower case equivalents.

case-region-capitalize [unbound]

Capitalize all the words in the region between dot and mark by making their first characters upper case and all the rest lower case.

case-region-invert [unbound]

Invert the case of all alphabetic characters in the region between dot and mark.

case-region-lower [unbound]

Change all alphabetic characters in the region between dot and mark to lower case.

case-region-upper [unbound]

Change all alphabetic characters in the region between dot and mark to upper case.

case-word-capitalize [unbound]

Capitalize the current word (the one above or to the left of dot) by making its first character upper case and all the rest lower case.

case-word-invert [unbound]

Invert the case of all alphabetic characters in the current word (the one above or to the left of dot).

case-word-lower **[unbound]**

Change all alphabetic characters in the current word (the one above or to the left of dot) to lower case.

case-word-upper **[unbound]**

Change all alphabetic characters in the current word (the one above or to the left of dot) to upper case.

change-current-process **[unbound]**

(change-current-process "process-name") -- Sets the current process to the one named.

change-directory **[unbound]**

Changes the current directory (for EMACS) to the named directory. All future file write and reads (↑X↑S, ↑X↑V, etc.) will be interpreted relative to that directory.

char-to-string **[unbound]**

Takes a numeric argument and returns a one character string that results from considering the number as an ascii character.

Command prefix (also known as META) **ESC**

The next character typed will be interpreted as a command based on the fact that it was preceded by ESC. The name **meta** for the ESC character comes from funny keyboards at Stanford and MIT that have a Meta-shift key which is used to extend the ASCII character set. Lacking a Meta key, we make do with prefixing with an ESC character. You may see (and hear) commands like ESC-V referred to as Meta-V. Sometimes the ESC key is confusingly written as $, so ESC-V would be written as $V. ESC is also occasionally referred to as *Altmode*, from the labeling of a key on those old favorites, model 33 teletypes.

command-prefix **↑X**

The next character typed will be interpreted as a command based on the fact that it was preceded by ↑X.

compile-it **↑X↑E**

Make is a standard Unix program which takes a description of how to compile a set of programs and compiles them. The output of *make* (and the compilers it calls) is placed in a buffer which is displayed in a window. If any errors were encountered, EMACS makes a note of them for later use with ↑X↑N. Presumably, a data base has been set up for *make* that causes the files which have been edited to be compiled. ↑X↑E then updates the files that have been changed and *make* does the necessary recompilations, and EMACS notes any errors and lets you peruse them with ↑X↑N.

If ↑X↑E is given a non-zero argument, then rather than just executing *make* EMACS will prompt for a Unix command line to be executed. Modified buffers will still be written out, and the output will still go to the *Error log* buffer and be parsed as error messages for use with ↑X↑N. One of the most useful applications of this feature involves the *grep* program. "↑U↑X↑Egrep -n MyProc *.cESC" will scan through all C source files looking for the string "MyProc" (which could be the name of a procedure). You can then use ↑X↑N to step through all places in all the files where the string was found. Note: The version of *grep* in my bin directory, /usr/jag/bin/grep, must be used: it prints line numbers in a format that is understood by EMACS. (ie. "*FileName*, line *LineNumber*)

concat [unbound]

Takes a set of string arguments and returns their concatenation.

continue-process [unbound]

(continue-process "process-name") -- Continue a process stopped by *stop-process*.

copy-region-to-buffer [unbound]

Copies the region between dot and mark to the named buffer. The buffer is emptied before the text is copied into it; the region between dot and mark is left undisturbed.

current-buffer-name [unbound]

MLisp function that returns the current buffer name as a string.

current-column [unbound]

(current-column) is an MLisp function that returns the printing column number of the character immediately following dot.

current-file-name [unbound]

MLisp function that returns the file name associated with the current buffer as a string. If there is no associated file name, the null string is returned.

current-indent [unbound]

(current-indent) is an MLisp function the returns the amount of whitespace at the beginning of the line which dot is in (the printing column number of the first non-whitespace character).

current-process [unbound]

(current-process) -- Returns the name of the current process as defined in the section describing the process mechanism.

current-time [unbound]

MLisp function that returns the current time of day as a string in the format described in CTIME(3), with the exception that the trailing newline will have been stripped off. (substr (current-time) -4 4) is the current year.

declare-global [unbound]

Takes a list of variables and for each that is not already bound a global binding is created. Global bindings outlive all function calls.

define-buffer-macro [unbound]

Take the contents of the current buffer and define it as a macro whose name is associated with the buffer. This is how one redefines a macro that has been edited using edit-macro.

define-global-abbrev [unbound]

Define (or redefine) an abbrev with the given name for the given phrase in the global abbreviation table.

define-keyboard-macro [unbound]

Give a name to the current keyboard macro. A keyboard macro is defined by using the ↑X(and ↑X) command; define-keyboard-macro takes the current keyboard macro, squirrels it away in a safe place, gives it a name, and erases the keyboard macro. define-string-macro is another way to define a macro.

define-keymap [unbound]

(define-keymap "mapname") defines a new, empty, keymap with the given name. See the section on keymaps, 14 page 16, for more information.

define-local-abbrev [unbound]

Define (or redefine) an abbrev with the given name for the given phrase in the local abbreviation table. A local abbrev table must have already been set up with *use-abbrev-table.*

define-string-macro [unbound]

Define a macro given a name and a body as a string entered in the minibuffer. Note: to get a control character into the body of the macro it must be quoted with ↑Q. define-keyboard-macro is another way to define a macro.

defun [unbound]

(defun (name expressions...)...) is an MLisp function that defines a new MLisp function with the given name and a body composed of the given expressions. The value of the function is the value of the last expression. For example:

```
(defun
    (indent-line                ; this function just sticks a tab at
        (save-excursion         ; the beginning of the current line
            (beginning-of-line) ; without moving dot.
            (insert-string "    ")
        )
    )
)
```

delete-buffer [unbound]

Deletes the named buffer.

delete-macro [unbound]

Delete the named macro.

delete-next-character ↑D

Delete the character immediatly following dot; that is, the character on which the terminals cursor sits. Lines may be merged by deleting newlines.

delete-next-word ESC-D

Delete characters forward from dot until the next end of a word. If dot is currently not in a word, all punctuation up to the beginning of the word is deleted as well as the word.

delete-other-windows ↑X1

Go back to one-window mode. Generally useful when EMACS has spontaneously generated a window (as for ESC-? or ↑X↑B) and you want to get rid of it.

delete-previous-character ↑H

Delete the character immediatly preceding dot; that is, the character to the left of the terminals cursor. If you've just typed a character, ↑H (*backspace*) will delete it. Lines may be merged by deleting newlines.

delete-previous-character RUBOUT

Delete the character immediatly preceding dot; that is, the character to the left of the terminals cursor. If you've just typed a character, RUBOUT will delete it. Lines may be merged by deleting newlines.

delete-previous-word ESC-H

If not in the middle of a word, delete characters backwards (to the left) until a word is found. Then delete the word to the left of dot. A word is a sequence of alphanumerics.

delete-region-to-buffer ESC-↑W

Wipe (kill, delete) all characters between dot and the mark. The deleted text is moved to a buffer whose name is prompted for, which is emptied first.

delete-to-killbuffer ↑W

Wipe (kill, delete) all characters between dot and the mark. The deleted text is moved to the kill buffer, which is emptied first.

delete-white-space [unbound]

Deletes all whitespace characters (spaces and tabs) on either side of dot.

delete-window ↑XD

Removes the current window from the screen and gives it's space to it's neighbour below (or above) and makes the current window and buffer those of the neighbour.

describe-bindings **[unbound]**

Places in the *Help* window a list of all the keys and the name of the procedure that they are bound to. This listing is suitable for printing and making you own quick-reference card for your own customized version of EMACS.

describe-command **[unbound]**

Describe the named extended command. An "extended command" is the first word that you type to the ESC-X command. "ESC-X**describe-command describe-command**" will print the documentation for the describe-command extended command.

describe-key **[unbound]**

Describe the given key. ESC-X**describe-key** ESC-X will print the documentation for the ESC-X key.

describe-variable **[unbound]**

Describe the named variable. A "variable" is something that you can set with the ESC-X**set** command or print with the ESC-X**print** command. They let the user fine-tune EMACS to their own taste. ESC-X**describe-variable right-margin** will print documentation about the right-margin setting.

describe-word-in-buffer ↑X↑D

Takes the word nearest the cursor and looks it up in a data base and prints the information found. This data base contains short one-line descriptions of all of the Unix standard procedures and Franz Lisp standard functions. The idea is that if you've just typed in the name of some procedure and can't quite remember which arguments go where, just type ↑X↑D and EMACS will try to tell you.

digit **[unbound]**

Heavy wizardry: you don't want to know. "digit" should eventually disappear.

dot **[unbound]**

(dot) is an MLisp function that returns the number of characters to the left of dot plus 1 (ie. if dot is at the beginning of the buffer, (dot) returns 1). The value of the function is an object of type "marker" -- if it is assigned to a variable then as changes are made to the buffer the variable's value continues to indicate the same position in the buffer.

dump-syntax-table [unbound]

Dumps a readable listing of a syntax table into a buffer and makes that buffer visible.

edit-macro [unbound]

Take the body of the named macro and place it in a buffer called *Macro edit*. The name of the macro is associated with the buffer and appears in the information bar at the bottom of the window. The buffer may be edited just like any other buffer (this is, in fact, the intent). After the macro body has been edited it may be redefined using *define-buffer-macro*.

end-of-file ESC->

Move dot to just after the last character of the buffer.

end-of-line ↑E

Move dot to the end of the line in the current buffer that contains dot; that is, to just after the following end-of-line or the end of the buffer.

end-of-window ESC-.

Move dot to just after the last character visible in the window.

enlarge-window ↑XZ

Makes the current window one line taller, and the window below (or the one above if there is no window below) one line shorter. Can't be used if there is only one window on the screen.

eobp [unbound]

(eobp) is an MLisp predicate that is true iff dot is at the end of the buffer.

eolp [unbound]

(eolp) is an MLisp predicate that is true iff dot is at the end of a line.

eot-process [unbound]

(eot-process "process-name") -- Send an EOT to the process.

erase-buffer [unbound]

Deletes all text from the current buffer. Doesn't ask to make sure if you really want to do it.

erase-region [unbound]

Erases the region between dot and mark. It is like delete-to-killbuffer except that it doesn't move the text to the kill buffer.

error-message **[unbound]**

(error-message "string-expression") Sends the *string-expression* to the screen as an error message where it will appear at the bottom of the screen. EMACS will return to keyboard level.

error-occured **[unbound]**

(error-occured expressions...) executes the given expressions and ignores their values. If all executed successfully, error-occured returns false. Otherwise it returns true and all expressions after the one which encountered the error will not be executed.

exchange-dot-and-mark **↑X↑X**

Sets dot to the currently marked position and marks the old position of dot. Useful for bouncing back and forth between two points in a file; particularly useful when the two points delimit a region of text that is going to be operated on by some command like ↑W (erase region).

execute-extended-command **ESC-X**

EMACS will prompt in the minibuffer (the line at the bottom of the screen) for a command from the extended set. These deal with rarely used features. Commands are parsed using a Twenex style command interpreter: you can type ESC or space to invoke command completion, or '?' for help with what you're allowed to type at that point. This doesn't work if it's asking for a key or macro name.

execute-keyboard-macro **↑XE**

Takes the keystrokes remembered with ↑X(and ↑X) and treats them as though they had been typed again. This is a cheap and easy macro facility. For more power, see the define-string-macro, define-keyboard-macro and bind-to-key commands.

execute-mlisp-buffer **[unbound]**

Parse the current buffer as as a single MLisp expression and execute it. This is what is generally used for testing out new functions: stick your functions in a buffer wrapped in a *defun* and use execute-mlisp-buffer to define them.

execute-mlisp-line **ESC-ESC**

Prompt for a string, parse it as an MLisp expression and execute it.

execute-monitor-command **↑X!**

Prompt for a Unix command then execute it, placing its output into a buffer called *Command execution* and making that buffer visible in a window. The command will not be able to read from its standard input (it will be connected to /dev/null). For now, there is no way to execute an interactive subprocess.

exit-emacs ↑C

Exit EMACS. Will ask if you're sure if there are any buffers that have been modified but not written out.

exit-emacs ↑X↑C

Exit EMACS. Will ask if you're sure if there are any buffers that have been modified but not written out.

exit-emacs ESC-↑C

Exit EMACS. Will ask if you're sure if there are any buffers that have been modified but not written out.

expand-mlisp-variable **[unbound]**

Prompts for the name of a declared variable then inserts the name as text into the current buffer. This is very handly for typing in MLisp functions. It's also fairly useful to bind it to a key for easy access.

expand-mlisp-word **[unbound]**

Prompt for the name of a command then insert the name as text into the current buffer. This is very handly for typing in MLisp functions. It's also fairly useful to bind it to a key for easy access.

extend-database-search-list **[unbound]**

(extend-database-search-list dbname filename) adds the given data base file to the data base search list (dbname). If the database is already in the search list then it is left, otherwise the new database is added at the beginning of the list of databases.

fetch-database-entry **[unbound]**

(fetch-database-entry dbname key) takes the entry in the data base corresponding to the given key and inserts it into the current buffer.

file-exists **[unbound]**

(file-exists *fn***)** returns 1 if the file named by *fn* exists and is writable, 0 if it does not exist, and -1 if it exists and is readable but not writable.

filter-region **[unbound]**

Take the region between dot and mark and pass it as the standard input to the given command line. Its standard output replaces the region between dot and mark. Use this to run a region through a Unix style-filter.

following-char **[unbound]**

(following-char) is an MLisp function that returns the character immediatly following dot. The null character (0) is returned if dot is at the end of the buffer. Remember that dot is not 'at' some character, it is between two characters.

forward-balanced-paren-line **[unbound]**

Moves dot forward until either

- The end of the buffer is reached.
- An unmatched close parenthesis, ')', is encountered. That is, unmatched between there and the starting position of dot.
- The beginning of a line is encountered at "parenthesis level zero". That is, without an unmatched '(' existing between there and the starting position of dot.

The definitions of parenthesis and strings from the syntax table for the current buffer are used.

forward-character **↑F**

Move dot forwards one character. Ends-of-lines and tabs each count as one character. You can't move forward to after the end of the buffer.

forward-paragraph **ESC-]**

Moves to the end of the current or following paragraph. Blank lines, and Scribe and nroff command lines separate paragraphs and are not parts of paragraphs.

forward-paren **[unbound]**

Moves dot forward until an unmatched close parenthesis, ')', or the end of the buffer is found. This can be used to aid in skipping over Lisp S-expressions. The definitions of parenthesis and strings from the syntax table for the current buffer are used.

forward-sentence **ESC-E**

Move dot forward to the beginning of the next sentence. Sentences are seperated by a '.', '?' or '!' followed by whitespace.

forward-word **ESC-F**

Move dot forward to the end of a word. If not currently in the middle of a word, skip all intervening punctuation. Then skip over the word, leaving dot positioned after the last character of the word. A word is a sequence of alphanumerics.

get-tty-buffer **[unbound]**

Given a prompt string it reads the name of a buffer from the tty using the minibuf and providing command completion.

get-tty-character [unbound]

Reads a single character from the terminal and returns it as an integer. The cursor is not moved to the message area, it is left in the text window. This is useful when writing things like query-replace and incremental search.

get-tty-command [unbound]

(get-tty-command *prompt*) prompts for the name of a declared function (using command completion & providing help) and returns the name of the function as a string. For example, the *expand-mlisp-word* function is simply (insert-string (get-tty-command ": expand-mlisp-word ")).

get-tty-string [unbound]

Reads a string from the terminal using its single string parameter for a prompt. Generally used inside MLisp programs to ask questions.

get-tty-variable [unbound]

(get-tty-variable *prompt*) prompts for the name of a declared variable (using command completion & providing help) and returns the name of the variable as a string. For example, the *expand-mlisp-variable* function is simply (insert-string (get-tty-variable ": expand-mlisp-variable ")).

getenv [unbound]

(getenv "*varname*") returns the named shell environment variable. for example, (getenv "HOME") will return a string which names your home directory.

goto-character [unbound]

Goes to the given character-position. (goto-character 5) goes to character position 5.

if [unbound]

(if test thenclause elseclause) is an MLisp function that executes and returns the value of *thenclause* iff *test* is true; otherwise it executes *elseclause* if it is present. For example:

```
(if (eolp)
    (to-col 33)
)
```

will tab over to column 33 if dot is currently at the end of a line.

illegal-operation [unbound]

Illegal-operation is bound to those keys that do not have a defined interpretation. Executing illegal-operation is an error.

indent-C-procedure ESC-J

Take the current C procedure and reformat it using the *indent* program, a fairly sophisticated pretty printer. *Indent-C-procedure* is God's gift to those who don't like to fiddle about getting their formatting right. *Indent-C-procedure* is usually bound to ESC-J. When switching from mode to mode, ESC-J will be bound to procedures appropriate to that mode. For example, in text mode ESC-J is bound to *justify-paragraph.*

insert-character [unbound]

Inserts its numeric argument into the buffer as a single character. (insert-character '0') inserts the character '0' into the buffer.

insert-file ↑X↑I

Prompt for the name of a file and insert its contents at dot in the current buffer.

insert-string [unbound]

(insert-string stringexpression) is an MLisp function that inserts the string that results from evaluating the given *stringexpression* and inserts it into the current buffer just before dot.

int-process [unbound]

(int-process "process-name") -- Send an interrupt signal to the process.

interactive [unbound]

An MLisp function which is true iff the invoking MLisp function was invoked interactively (ie. bound to a key or by ESC-X).

is-bound [unbound]

an MLisp predicate that is true iff all of its variable name arguments are bound.

justify-paragraph [unbound]

Take the current paragraph (bounded by blank lines or Scribe control lines) and pipe it through the "fmt" command which does paragraph justification. *justify-paragraph* is usually bound to ESC-J when in text mode.

kill-process [unbound]

(kill-process "process-name") -- Send a kill signal to the process.

kill-to-end-of-line ↑K

Deletes characters forward from dot to the immediatly following end-of-line (or end of buffer if there isn't an end of line). If dot is positioned at the end of a line then the end-of-line character is deleted. Text deleted by the ↑K command is placed into the *Kill buffer* (which really is a buffer that you can look at). A ↑K command normally erases the contents of the kill buffer first; subsequent ↑K's in an unbroken sequence append to the kill buffer.

last-key-struck [unbound]

The last command character struck. If you have a function bound to many keys the function may use last-key-struck to tell which key was used to invoke it. (insert-character (last-key-struck)) does the obvious thing.

length [unbound]

Returns the length of its string parameter. (length "time") => 4.

line-to-top-of-window ESC-!

What more can I say? This one is handy if you've just searched for the declaration of a procedure, and want to see the whole body (or as much of it as possible).

list-buffers ↑X↑B

Produces a listing of all existing buffers giving their names, the name of the associated file (if there is one), the number of characters in the buffer and an indication of whether or not the buffer has been modified since it was read or written from the associated file.

list-databases [unbound]

(list-databases) lists all data base search lists.

list-processes [unbound]

(list-processes) -- Analagous to "list-buffers". Processes which have died only appear once in this list before completely disappearing.

load [unbound]

Read the named file as a series of MLisp expressions and execute them. Typically a loaded file consists primarily of *defun*'s and buffer-specific variable assignments and key bindings. *Load* is usually used to load macro libraries and is used to load ".emacs_pro" from your home directory when EMACS starts up.

For example, loading this file:

```
(setq right-margin 75)
(defun (my-linefeed
            (end-of-line)
            (newline-and-indent)
       )
)
(bind-to-key "my-linefeed" 10)
```

sets the *right-margin* to 75 and defines a function called *my-linefeed* and binds it to the linefeed key (which is the ascii character 10 (decimal))

The file name given to *load* is interpreted relative to the EPATH environment variable, which is interpreted in the same manner as the shell's PATH variable. That is, it provides a list of colon-separated names that are taken to be the names of directories that are searched for the named files. The default value of EPATH searches your current directory and then a central system directory.

Temporary hack: in previous versions of EMACS *load*ed files were treated as a sequence of keystrokes. This behaviour has been decreed bogus and unreasonable, hence it has been changed. However, to avoid loud

cries of anguish the *load* command still exhibits the old behaviour if the first character of the loaded file is an ESC.

local-bind-to-key **[unbound]**

Prompt for the name of a command and a key and bind that command to the given key but unlike *bind-to-key* the binding only has effect in the current buffer. This is generally used for mode specific bindings that will generally differ from buffer to buffer.

looking-at **[unbound]**

(looking-at "SearchString") is true iff the given regular expression search string matches the text immediatly following dot. This is for use in packages that want to do a limited sort of parsing. For example, if dot is at the beginning of a line then **(looking-at "[\t]*else])** will be true if the line starts with an "else". See section 13, page 14 for more information on regular expressions.

mark **[unbound]**

An MLisp function that returns the position of the marker in the current buffer. An error is signaled if the marker isn't set. The value of the function is an object of type "marker" -- if it is assigned to a variable then as changes are made to the buffer the variable's value continues to indicate the same position in the buffer.

message **[unbound]**

(message stringexpression) is an MLisp function that places the string that results from the evaluation of the given *stringexpression* into the message region on the display (the line at the bottom).

modify-syntax-entry **[unbound]**

Modify-syntax-entry is used to modify a set of entries in the syntax table associated with the current buffer. Syntax tables are associated with buffers by using the *use-syntax-table* command. Syntax tables are used by commands like *forward-paren* to do a limited form of parsing for language dependent routines. They define such things as which characters are parts of words, which quote strings and which delimit comments (currently, nothing uses the comment specification). To see the contents of a syntax table, use the *dump-syntax-table* command.

The parameter to *modify-syntax-entry* is a string whose first five characters specify the interpretation of the sixth and following characters.

The first character specifies the type. It may be one of the following:

'w' A word character, as used by such commands as *forward-word* and *case-word-capitalize.*

space A character with no special interpretation.

'(' A left parenthesis. Typical candidates for this type are the characters '(', '[' and '{'. Characters of this type also have a matching right parenthesis specified (')', ']' and '}' for example) which appears as the second character of the parameter to *modify-syntax-entry.*

')' A right parenthesis. Typical candidates for this type are the characters ')', ']' and '}'. Characters of this type also have a matching left parenthesis specified ('(', '[' and '{' for

example) which appears as the second character of the parameter to *modify-syntax-entry.*

'"' A quote character. The C string delimiters " and ' are usually given this class, as is the Lisp |.

'\' A prefix character, like \ in C or / in MacLisp.

The second character of the parameter is the matching parenthesis if the character is of the left or right parenthesis type. If you specify that '(' is a right parenthesis matched by ')', then you should also specify that ')' is a left parenthesis matched by '('.

The third character, if equal to '{', says that the character described by this syntax entry can begin a comment; the forth character, if equal to '}' says that the character described by this syntax entry can end a comment. If either the beginning or ending comment sequence is two characters long, then the fifth character provides the second character of the comment sequence.

The sixth and following characters specify which characters are described by this entry; a range of characters can be specified by putting a '-' between them, a '-' can be described if it appears as the sixth character.

A few examples, to help clear up my muddy exposition:

```
(modify-syntax-entry "w   -")  ; makes '-' behave as a normal word
                               ; character (ESC-F will consider
                               ; one as part of a word)
(modify-syntax-entry "(]  [")  ; makes '[' behave as a left parenthesis
                               ; which is matched by ']'
(modify-syntax-entry ")[  ]")  ; makes ']' behave as a right parenthesis
                               ; which is matched by '['
```

move-to-comment-column **[unbound]**

If the cursor is *not* at the beginning of a line, ESC-C moves the cursor to the column specified by the comment-column variable by inserting tabs and spaces as needed. In any case, it the sets the right margin to the column finally reached. This is usually used in macros for language-specific comments.

nargs **[unbound]**

An MLisp function which returns the number of arguments passed to the invoking MLisp function. For example, within the execution of *foo* invoked by (foo x y) the value of *nargs* will be 2.

narrow-region **[unbound]**

The *narrow-region* command sets the restriction to encompass the region between dot and mark. Text outside this region will henceforth be totally invisible. It won't appear on the screen and it won't be manipulable by any editing commands. This can be useful, for instance, when you want to perform a replacement within a few paragraphs: just narrow down to a region enclosing the paragraphs and execute *replace-string.*

newline **[unbound]**

Just inserts a newline character into the buffer -- this is what the RETURN (↑M) key is generally bound to.

newline-and-backup **↑O**

Insert an end-of-line immediatly *after* dot, effectivly opening up space. If dot is positioned at the beginning of a line, then ↑O will create a blank line preceding the current line and position dot on that new line.

newline-and-indent **LINEFEED**

Insert a newline, just as typing RETURN does, but then insert enough tabs and spaces so that the newly created line has the same indentation as the old one had. This is quite useful when you're typing in a block of program text, all at the same indentation level.

next-error **↑X↑N**

Take the next error message (as returned from the ↑X↑E (compile) command), do a *visit* (↑X↑V) on the file in which the error occurred and set dot to the line on which the error occurred. The error message will be displayed at the top of the window associated with the *Error log* buffer.

next-line **↑N**

Move dot to the next line. ↑N and ↑P attempt to keep dot at the same horizontal position as you move from line to line.

next-page **↑V**

Reposition the current window on the current buffer so that the next page of the buffer is visible in the window (where a *page* is a group of lines slightly smaller than a window). In other words, it flips you forward a page in the buffer. Its inverse is ESC-V. If possible, dot is kept where it is, otherwise it is moved to the middle of the new page.

next-window **↑XN**

Switches to the window (and associated buffer) that is below the current window.

novalue **[unbound]**

Does nothing. (novalue) is a complete no-op, it performs no action and returns no value. Generally the value of a function is the value of the last expression evaluated in it's body, but this value may not be desired, so (novalue) is provided so that you can throw it away.

page-next-window **ESC-↑V**

Repositions the window below the current one (or the top one if the current window is the lowest one on the screen) on the displayed buffer so that the next page of the buffer is visible in the window (where a *page* is a group of lines slightly smaller than a window). In other words, it flips you forward a page in the buffer of the *other* window.

If ESC-↑V is given an argument it will flip the buffer backwards a page, rather than forwards. So ESC-↑V is roughly equivalent to ↑V and ↑UESC-↑V is roughly equivalent to ESC-V except that they deal with the

other window. Yes, yes, yes. I realize that this is a bogus command structure, but I didn't invent it. Besides, you can learn to love it.

parse-error-messages-in-region **[unbound]**

Parses the region between dot and mark for error messages (as in the *compile-it* (↑X↑E) command) and sets up for subsequent invocations of *next-error* (↑X↑N). See the description of the *compile-it* command, and section 9 (page 6).

pause-emacs **[unbound]**

Pause, giving control back to the superior shell using the job control facility of Berkeley Unix. The screen is cleaned up before the shell regains control, and when the shell gives control back to EMACS the screen will be fixed up again. Users of the sea-shell (csh) will probably rather use this command than "return-to-monitor", which is similar, except that it recursivly invokes a new shell.

pop-to-buffer **[unbound]**

Switches to a buffer whose name is provided and ties that buffer to a popped-up window. Pop-to-buffer is exactly the same as switch-to-buffer except that switch-to-buffer ties the buffer to the current window, pop-to-buffer finds a new window to tie it to.

preceding-char **[unbound]**

(preceding-char) is an MLisp function that returns the character immediatly preceding dot. The null character (0) is returned if dot is at the beginning of the buffer. Remember that dot is not 'at' some character, it is between two characters.

prefix-argument-loop **[unbound]**

(prefix-argument-loop <statements>) executes <statements> prefix-argument times. Every function invocation is always prefixed by some argument, usually by the user typing ↑U*n*. If no prefix argument has been provided, 1 is assumed. See also the command provide-prefix-argument and the variable prefix-argument.

previous-command **[unbound]**

(previous-command) *usually* returns the character value of the keystroke that invoked the previous command. In is something like *last-key-struck*, which returns the keystroke that invoked the current command. However, a function may set the variable *this-command* to some value, which will be the value of *previous-command* after the next command invocation. This rather bizarre command/variable pair is intended to be used in the implementation of MLisp functions which behave differently when chained together (ie. executed one after the other). A good example is ↑K, *kill-to-end-of-line* which appends the text from chained kills to the killbuffer.

To use this technique for a set of commands which are to exhibit a chaining behaviour, first pick a magic number. -84, say. Then each command in this set which is chainable should **(setq this-command -84)**. Then to tell if a command is being chained, it suffices to check to see if **(previous-command)** returns -84.

Did I hear you scream "hack"??

previous-line ↑P

Move dot to the previous line. ↑N and ↑P attempt to keep dot at the same horizontal position as you move from line to line.

previous-page ESC-V

Repositions the current window on the current buffer so that the previous page of the buffer is visible in the window (where a *page* is a group of lines slightly smaller than a window). In other words, it flips you backward a page in the buffer. Its inverse is ↑V. If possible, dot is kept where it is, otherwise it is moved to the middle of the new page.

previous-window ↑XP

Switches to the window (and associated buffer) that is above the current window.

print **[unbound]**

Print the value of the named variable. This is the command you use when you want to inquire about the setting of some switch or parameter.

process-output **[unbound]**

(process-output) -- Can only be called by the *on-output-procedure* to procure the output generated by the process whose name is given by *MPX-process*. Returns the output as a string.

process-status **[unbound]**

(process-status "process-name") -- Returns -1 if "process-name" isn't a process, 0 if the process is stopped, and 1 if the process is running.

progn **[unbound]**

(progn expressions...) is an MLisp function that evaluates the expressions and returns the value of the last expression evaluated. *Progn* is roughly equivalent to a compound statement (begin-end block) in more conventional languages and is used where you want to execute several expressions when there is space for only one (eg. the *then* or *else* parts of an *if* expression).

provide-prefix-argument **[unbound]**

(provide-prefix-argument <value> <statement>) provides the prefix argument <value> to the <statement>. For example, the most efficient way to skip forward 5 words is:

```
(provide-prefix-argument 5 (forward-word))
```

See also the command prefix-argument-loop and the variable prefix-argument.

push-back-character **[unbound]**

Takes the character provided as its argument and causes it to be used as the next character read from the keyboard. It is generally only useful in MLisp functions which read characters from the keyboard, and upon finding one that they don't understand, terminate and behave as though the key had been struck to the EMACS keyboard command interpreter. For example, ITS style incremental search.

put-database-entry **[unbound]**

(put-database-entry dbname key) takes the current buffer and stores it into the named database under the given key.

query-replace-string ESC-Q

Replace all occurrences of one string with another, starting at dot and ending at the end of the buffer. EMACS prompts for an old and a new string in the minibuffer (the line at the bottom of the screen). See the section on searching, section 13 page 14 for more information on search strings. For each occurrence of the old string, EMACS requests that the user type in a character to tell it what to do (dot will be positioned just after the found string). The possible replies are:

<space>	Change this occurrence and continue to the next.
n	Don't change this occurrence, but continue to the next
r	Enter a recursive-edit. This allows you to make some local changes, then continue the query-replace-string by typing ↑C.
!	Change this occurrence and all the rest of the occurrences without bothering to ask.
.	Change this one and stop: don't do any more replaces.
↑G	Don't change this occurrence and stop: don't do any more replaces.
?	(or anything else) Print a short list of the query/replace options.

quietly-read-abbrev-file **[unbound]**

Read in and define abbrevs appearing in a named file. This file should have been written using *write-abbrev-file*. Unlike *read-abbrev-file*, an error message is not printed if the file cannot be found.

quit-process **[unbound]**

(quit-process "process-name") -- Send a quit signal to the process.

quote-character ↑Q

Insert into the buffer the next character typed without interpreting it as a command. This is how you insert funny characters. For example, to insert a ↑L (form feed or page break character) type ↑Q↑L. This is the only situation where ↑G isn't interpreted as an abort character.

re-query-replace-string **[unbound]**

re-query-replace-string is identical to *query-replace-string* except that the search string is a regular expression rather than an uninterpreted sequence of characters. See the section on searching, section 13 page 14 for more information.

re-replace-string **[unbound]**

re-replace-string is identical to *replace-string* except that the search string is a regular expression rather than an uninterpreted sequence of characters. See the section on searching, section 13 page 14 for more information.

re-search-forward **[unbound]**

re-search-forward is identical to *search-forward* except that the search string is a regular expression rather than an uninterpreted sequence of characters. See the section on searching, section 13 page 14 for more information.

re-search-reverse **[unbound]**

re-search-reverse is identical to *search-reverse* except that the search string is a regular expression rather than an uninterpreted sequence of characters. See the section on searching, section 13 page 14 for more information.

read-abbrev-file **[unbound]**

Read in and define abbrevs appearing in a named file. This file should have been written using *write-abbrev-file*. An error message is printed if the file cannot be found.

read-file **↑X↑R**

Prompt for the name of a file; erase the contents of the current buffer; read the file into the buffer and associate the name with the buffer. Dot is set to the beginning of the buffer.

recursion-depth **[unbound]**

Returns the depth of nesting within *recursive-edit*'s. It returns 0 at the outermost level.

recursive-edit **[unbound]**

The *recursive-edit* function is a call on the keyboard read/interpret/execute routine. After *recursive-edit* is called the user can enter commands from the keyboard as usual, except that when he exits EMACS by calling *exit-emacs* (typing ↑C) it actually returns from the call to *recursive-edit*. This function is handy for packages that want to pop into some state, let the user do some editing, then when they're done perform some cleanup and let the user resume. For example, a mail system could use this for message composition.

redraw-display ↑L

Clear the screen and rewrite it. This is useful if some transmission glitch, or a message from a friend, has messed up the screen.

region-around-match **[unbound]**

Region-around-match sets dot and mark around the region matched by the last search. An argument of *n* puts dot and mark around the *n*'th subpattern matched by '\(' and '\)'. This can then be used in conjuction with *region-to-string* to extract fields matched by a patter. For example, consider the following fragment that extracts user names and host names from mail addresses:

```
(re-search-forward "\\([a-z][a-z]*\\) *@ *\\([a-z][a-z]*\\)")
(region-around-match 1)
(setq username (region-to-string))
(region-around-match 2)
(setq host (region-to-string))
```

Applying this MLisp code to the text "send it to jag@vlsi" would set the variable 'username' to "jag" and 'host' to "vlsi".

region-to-process **[unbound]**

(region-to-process "process-name") -- The region is wrapped up and sent to the process.

region-to-string **[unbound]**

Returns the region between dot and mark as a string. Please be kind to the storage allocator, don't use huge strings.

remove-all-local-bindings **[unbound]**

Perform a remove-local-binding for all possible keys; effectively undoes all local bindings. Mode packages should execute this to initialize the local binding table to a clean state.

remove-binding **[unbound]**

Removes the global binding of the given key. Actually, it just rebinds the key to *illegal-operation.*

remove-local-binding **[unbound]**

Removes the local binding of the given key. The global binding will subsequently be used when interpreting the key. ***Bug:*** there really should be some way of saving the current binding of a key, then restoring it later.

replace-string **ESC-R**

Replace all occurrences of one string for another, starting at dot and ending and the end of the buffer. EMACS prompts for an old and a new string in the minibuffer (the line at the bottom of the screen). Unlike **query-replace-string** EMACS doesn't ask any questions about particular occurrences, it just changes them. Dot will be left after the last changed string. See the section on searching, section 13 page 14 for more information on search strings.

return-prefix-argument [unbound]

(return-prefix-argument *n*) sets the numeric prefix argument to be used by the next function invocation to *n*. The next function may be either the next function in the normal flow of MLisp execution or the next function invoked from a keystroke. *Return-prefix-argument* is to be used by functions that are to be bound to keys and which are to provide a prefix argument for the next keyboard command.

return-to-monitor ↑_

Recursivly invokes a new shell, allowing the user to enter normal shell commands and run other programs. Return to EMACS by exiting the shell; ie. by typing ↑D.

save-excursion [unbound]

(save-excursion expressions...) is an MLisp function that evaluates the given expressions and returns the value of the last expression evaluated. It is much like *progn* except that before any expressions are executed dot and the current buffer are "marked" (via the marker mechanism) then after the last expression is executed dot and the current buffer are reset to the marked values. This properly takes into account all movements of dot and insertions and deletions that occur. *Save-excursion* is useful in MLisp functions where you want to go do something somewhere else in this or some other buffer but want to return to the same place when you're done; for example, inserting a tab at the beginning of the current line.

save-restriction [unbound]

Save-restriction is only useful to people writing MLisp programs. It is used to save the region restriction for the current buffer (and ***only*** the region restriction) during the execution of some subexpression that presumably uses region restrictions. The value of **(save-excursion expressions...)** is the value of the last expression evaluated.

save-window-excursion [unbound]

save-window-excursion is identical to *save-excursion* except that it also saves (in a rough sort of way) the state of the windows. That is, **(save-window-excursion expressions...)** saves the current dot, mark, buffer and window state, executes the expressions, restores the saved information and returns the value of the last expression evaluated.

When the window state is saved EMACS remembers which buffers were visible. When it is restored, EMACS makes sure that exactly those buffers are visible. EMACS does *not* save and restore the exact layout of the windows: this is a feature, not a bug.

scroll-one-line-down ESC-Z

Repositions the current window on the current buffer so that the line which is currently the second to the last line in the window becomes the last -- effectivly it moves the buffer down one line in the window. ↑Z is its inverse.

scroll-one-line-up ↑Z

Repositions the current window on the current buffer so that the line which is currently the second line in the window becomes the first -- effectivly it moves the buffer up one line in the window. ESC-Z is its inverse.

search-forward ↑S

Prompt for a string and search for a match in the current buffer, moving forwards from dot, stopping at the end of the buffer. Dot is left at the end of the matched string if a match is found, or is unmoved if not. See the section on searching, section 13 page 14 for more information.

search-reverse ↑R

Prompt for a string and search for a match in the current buffer, moving backwards from dot, stopping at the beginning of the buffer. Dot is left at the beginning of the matched string if a match is found, or is unmoved if not. See the section on searching, section 13 page 14 for more information.

self-insert [unbound]

This is tied to those keys which are supposed to self-insert. It is roughly the same as (insert-character (last-key-struck)) with the exception that it doesn't work unless it is bound to a key.

send-string-to-terminal [unbound]

(send-string-to-terminal "string") sends the string argumetn out to the terminal with *no* conversion or interpretation. This should only be used for such applications as loading function keys when EMACS starts up. If you screw up the screen, EMACS won't know about it and won't fix it up automatically for you -- you'll have to type ↑L.

set [unbound]

Set the value of some variable internal to EMACS. EMACS will ask for the name of a variable and a value to set it to. The variables control such things as margins, display layout options, the behavior of search commands, and much more. The available variables and switches are described elsewhere. Note that if *set* is used from MLisp the variable name must be a string: (set "left-margin" 77).

set-mark ↑@

Puts the marker for this buffer at the place where dot is now, and leaves it there. As text is inserted or deleted around the mark, the mark will remain in place. Use ↑X↑X to move to the currently marked position.

setq [unbound]

Assigns a new value to a variable. Variables may have either string or integer values. (setq i 5) sets i to 5; (setq s (concat "a" "b")) sets s to "ab".

shrink-window ↑X↑Z

Makes the current window one line shorter, and the window below (or the one above if there is no window below) one line taller. Can't be used if there is only one window on the screen.

sit-for [unbound]

Updates the display and pauses for n/10 seconds. (sit-for 10) waits for one second. This is useful in such things as a Lisp auto-paren balencer.

split-current-window ↑X2

Enter two-window mode. Actually, it takes the current window and splits it into two windows, dividing the space on the screen equally between the two windows. An arbitrary number of windows can be created -- the only limit is on the amount of space available on the screen, which, *sigh*, is only 24 lines on most terminals available these days (with the notable exception of the Ann Arbor Ambassador which has 60).

start-filtered-process [unbound]

(start-filtered-process "command" "buffer-name" "on-output-procedure") -- Does the same thing as start-process except that things are set up so that "on-output-procedure" is automatically called whenever output has been received from this process. This procedure can access the name of the process producing the output by refering to the variable *MPX-process*, and can retrieve the output itself by calling the procedure *process-output*.

The filter procedure must be careful to avoid generating side-effects (eg. *search-forward*). Moreover, if it attempts to go to the terminal for information, output from other processes may be lost.

start-process [unbound]

(start-process "command" "buffer-name") -- The home shell is used to start a process executing the command. This process is tied to the buffer "buffer-name" unless it is null in which case the "Command execution" buffer is used. Output from the process is automatically attached to the end of the buffer. Each time this is done, the mark is left at the end of the output (which is the end of the buffer).

start-remembering ↑X(

All following keystrokes will be remembered by EMACS.

stop-process [unbound]

(stop-process "process-name") -- Tell the process to stop by sending it a stop signal. Use *continue-process* to carry on.

stop-remembering **↑X)**

Stops remembering keystrokes, as initiated by ↑X(. The remembered keystrokes are not forgotten and may be re-executed with ↑XE.

string-to-char **[unbound]**

Returns the integer value of the first character of its string argument. (string-to-char "0") = '0'.

string-to-process **[unbound]**

(string-to-process "process-name" "string") -- The string is sent to the process.

substr **[unbound]**

(substr str pos n) returns the substring of string *str* starting at position *pos* (numbering from 1) and running for *n* characters. If *pos* is less than 0, then length of the string is added to it; the same is done for *n*. (substr "kzin" 2 2) = "zi"; (substr "blotto.c" -2 2) = ".c".

switch-to-buffer **↑XB**

Prompt for the name of the buffer and associate it with the current window. The old buffer associated with this window merely loses that association: it is not erased or changed in any way. If the new buffer does not exist, it will be created, in contrast with ↑X↑O.

system-name **[unbound]**

Is an MLisp function that returns the name of the system on which EMACS is being run. This should be the ArpaNet or EtherNet (or whatever) host name of the machine.

temp-use-buffer **[unbound]**

Switch to a named buffer *without* changing window associations. The commands pop-to-buffer and switch-to-buffer both cause a window to be tied to the selected buffer, temp-use-buffer does not. There are a couple of problems that you must beware when using this command: The keyboard command driver insists that the buffer tied to the current window be the current buffer, if it sees a difference then it changes the current buffer to be the one tied to the current window. This means that temp-use-buffer will be ineffective from the keyboard, switch-to-buffer should be used instead. The other problem is that "dot" is really a rather funny concept. There is a value of "dot" associated with each *window*, not with each *buffer*. This is done so that there is a valid interpretation to having the same buffer visible in several windows. There is also a value of "dot" associated with the current buffer. When you switch to a buffer with temp-use-buffer, this "transient dot" is what gets used. So, if you switch to another buffer, then use temp-use-buffer to get back, "dot" will have been set to 1. You can use save-excursion to remember your position.

to-col [unbound]

(to-col *n*) is an MLisp function that insert tabs and spaces to move the following character to printing column *n*.

transpose-characters ↑T

Take the two characters preceding dot and exchange them. One of the most common errors for typists to make is transposing two letters, typing "hte" when "the" is meant. ↑T makes correcting these errors easy, especially if you can develop a "↑T reflex".

unlink-file [unbound]

(unlink-file *fn***)** attempts to unlink (remove) the file named *fn*. It returns true if the unlink failed.

use-abbrev-table [unbound]

Sets the current local abbrev table to the one with the given name. Local abbrev tables are buffer specific and are usually set depending on the major mode. Several buffers may have the same local abbrev table. If either the selected abbrev table or the global abbrev table have had some abbrevs defined in them, *abbrev-mode* is turned on for the current buffer.

use-global-map [unbound]

(use-global-map "mapname") uses the named map to be used for the global interpretation of all key strokes. *use-local-map* is used to change the local interpretation of key strokes. See the section on keymaps, 14 page 16, for more information.

use-local-map [unbound]

(use-local-map "mapname") uses the named map to be used for the local interpretation of all key strokes. *use-global-map* is used to change the global interpretation of key strokes. See the section on keymaps, 14 page 16, for more information.

use-old-buffer ↑X↑O

Prompt for the name of the buffer and associate it with the current window. The old buffer associated with this window merely loses that association: it is not erased or changed in any way. The buffer must already exist, in contrast with ↑XB.

use-syntax-table [unbound]

Associates the named syntax table with the current buffer. See the description of the modify-syntax-entry command for more information on syntax tables.

users-full-name [unbound]

MLisp function that returns the users full name as a string. [Really, it returns the contents of the gecos field of the passwd entry for the current user, which is used on many systems for the users full name.]

users-login-name [unbound]

MLisp function that returns the users login name as a string.

visit-file ↑X↑V

Visit-file asks for the name of a file and switches to a buffer that contains it. The file name is expanded to it's full absolute form (that is, it will start with a '/'). If no buffer contains the file already then EMACS will switch to a new buffer and read the file into it. The name of this new buffer will be just the last component of the file name (everything after the last '/' in the name). If there is already a buffer by that name, and it contains some other file, then EMACS will ask "Enter a new buffer name or <CR> to overwrite the old buffer". For example, if my current directory is "/usr/jag/emacs" and I do a ↑X↑V and give EMACS the file name "../.emacs_pro"then the name of the new buffer will be ".emacs_pro" and the file name will be "/usr/jag/.emacs_pro". ↑X↑V is the approved way of switching from one file to another within an invocation of EMACS.

while [unbound]

(while test expressions...) is an MLisp function that executes the given expressions while the test is true.

widen-region [unbound]

The *widen-region* command sets the restriction to encompass the entire buffer. It is usualy used after a *narrow-region* to restore EMACS's attention to the whole buffer.

working-directory [unbound]

Returns the pathname of the current working directory.

write-abbrev-file [unbound]

Write all defined abbrevs to a named file. This file is suitable for reading back with *read-abbrev-file.*

write-current-file ↑X↑S

Write the contents of the current buffer to the file whose name is associated with the buffer.

write-file-exit ↑X↑F

Write all modified buffers to their associated files and if all goes well, EMACS will exit.

write-modified-files ↑X↑M

Write each modified buffer (as indicated by ↑X↑B) onto the file whose name is associated with the buffer. EMACS will complain if a modified buffer does not have an associated file.

write-named-file ↑X↑W

Prompt for a name; write the contents of the current buffer to the named file.

yank-buffer ESC-↑Y

Take the contents of the buffer whose name is prompted for and insert it at dot in the current buffer. Dot is left after the inserted text.

yank-from-killbuffer ↑Y

Take the contents of the kill buffer and inserts it at dot in the current buffer. Dot is left after the inserted text.

/ [unbound]

(| e_1 e_2) MLisp function that returns e_1 | e_2.

21. Options

This chapter describes (in alpahbetical order) all of the variables which the user may set to configure EMACS to taste.

ask-about-buffer-names

The *ask-about-buffer-names* variable controls what the *visit-file* command does if it detects a collision when constructing a buffer name. If *ask-about-buffer-names* is true (the default) then Emacs will ask for a new buffer name to be given, or for <CR> to be typed which will overwrite the old buffer. If it is false then a buffer name will be synthesized by appending "<*n*>" to the buffer name, for a unique value of *n*. For example, if I *visit-file* "makefile" then the buffer name will be "makefile"; then if I *visit-file* "man/makefile" the buffer name will be "makefile<2>".

backup-by-copying

If true, then when a backup of a file is made (see the section on the backup-before-writing variable) then rather than doing the fancy link/unlink footwork, EMACS copies the original file onto the backup. This preserves all link and owner information & ensures that the files I-number doesn't change (you're crazy if you worry about a files I-number). Backup-by-copying incurs a fairly heafty performance penalty. See the section on the backup-by-copying-when-linked variable for a description of a compromise. (default OFF)

backup-by-copying-when-linked

If true, then when a backup of a file is made (see the section on the backup-before-writing variable) then if the link count of the file is greater than 1, rather than doing the fancy link/unlink footwork, EMACS copies the original file onto the backup. If the link count is 1, then the link/unlink trick is pulled. This preserves link information when it is important, but still manages reasonable performance the rest of the time. See the section on the backup-by-copying variable for a description of a how to have owner & I-number information preserved. (default OFF)

backup-when-writing

If ON EMACS will make a backup of a file just before the first time that it is overwritten. The backup will have the same name as the original, except that the string ".BAK" will be appended; unless the last name in the path has more than 10 characters, in which case it will be truncated to 10 characters. "foo.c" gets backed up on "foo.c.BAK"; "/usr/jag/foo.c" on "/usr/jag/foo.c.BAK"; and "EtherService.c" on "EtherServi.BAK". The backup will only be made the first time that the file is rewritten from within the same invocation of EMACS, so if you write out the file several times the .BAK file will contain the file as it was before EMACS was invoked. The backup is normally made by fancy footwork with links and unlinks, to achieve acceptable performance: when "foo.c" is to be rewritten, EMACS effectivly executes a "mv foo.c foo.c.BAK" and then creates foo.c a write the new copy. The file protection of foo.c is copied from the old foo.c, but old links to the file now point to the .BAK file, and the owner of the new file is the person running EMACS. If you don't like this behaviour, see the switches backup-by-copying and backup-by-copying-when-linked. (default OFF)

buffer-is-modified

Buffer-is-modified is true iff the current buffer has been modified since it was last written out. You may set if OFF (ie. to 0) if you want EMACS to ignore the mods that have been made to this buffer -- it doesn't get you back to the unmodified version, it just tells EMACS not to write it out with the other modified files. EMACS sets buffer-is-modified true any time the buffer is modified.

case-fold-search

If set ON all searches will ignore the case of alphabetics when doing comparisons. (default OFF)

checkpoint-frequency

The number of keystrokes between checkpoints. Every "checkpoint-frequency" keystrokes all buffers which have been modified since they were last checkpointed are written to a file named "*file*.CKP". *File* is the file name associated with the buffer, or if that is null, the name of the buffer. Proper account is taken of the restriction on file names to 14 characters. (default 300)

comment-column

The column at which comments are to start. Used by the language-dependent commenting features through the *move-to-comment-column* command. (default 33)

ctlchar-with-↑

If set ON control characters are printed as ↑C (an '↑' character followed by the upper case alphabetic that corresponds to the control character), otherwise they are printed according to the usual Unix convention ('\' followed by a three digit octal number). (default OFF)

default-case-fold-search

Default-case-fold-search provides the default value for *case-fold-search*, which is used whenever a new buffer is created. (default OFF)

default-comment-column

Default-comment-column provides the default value for *comment-column*, which is used whenever a new buffer is created. Its initial value is 33.

default-left-margin

Default-left-margin provides the default value for *left-margin*, which is used whenever a new buffer is created. (default 1)

default-mode-line-format

This is the value given to *mode-line-format* when a buffer is created.

default-right-margin

Default-right-margin provides the default value for *right-margin*, which is used whenever a new buffer is created. Its initial value is some very large number.

default-tab-size

This is the value given to *tab-size* when a buffer is created. (default 8).

files-should-end-with-newline

Indicates that when a buffer is written to a file, and the buffer doesn't end in a newline, then the user should be asked if they want to have a newline appended. It used to be that this was the default action, but some people objected to the question being asked. (default ON)

global-mode-string

Global-mode-string is a global variable used in the construction of mode lines see section 16, page 18 for more information.

help-on-command-completion

If ON EMACS will print a list of possibilities when an ambiguous command is given, otherwise it just rings the bell and waits for you to type more. (default ON)

left-margin

The left margin for automatic text justification. After an automatically generated *newline* the new line will be indented to the left margin.

mode-line-format

mode-line-format is a buffer specific variable used to specify the format of a mode line. See section 16, page 18 for more information.

mode-string

Mode-string is a buffer specific variable used in the construction of mode lines see section 16, page 18 for more information.

needs-checkpointing

A buffer-specific variable which if ON indicates that the buffer should be checkpointed periodically. If it is OFF, then no checkpoints will be done. (default ON)

pop-up-windows

If ON EMACS will try to use some window other than the current one when it spontaneously generates a buffer that it wants you to see or when you visit a file (it may split the current window). If OFF the current window is always used. (default ON)

prefix-argument

Every function invocation is always prefixed by a numeric argument, either explicitly with ↑U*n* or provide-prefix-argument. The value of the variable prefix-argument is the argument prefixed to the invocation of the current MLisp function. For example, if the following function:

```
(defun
    (show-it
        (message (concat "The prefix argument is " prefix-argument))
    )
)
```

were bound to the key ↑A then typing ↑U↑A would cause the message "The prefix argument is 4" to be printed, and ↑U13↑A would print "The prefix argument is 13". See also the commands prefix-argument-loop and provide-prefix-argument.

prefix-argument-provided

True iff the execution of the current function was prefixed by a numeric argument. Use *prefix-argument* to get it's value.

prefix-string

The string that is inserted after an automatic *newline* has been generated in response to going past the right margin. This is generally used by the language-dependent commenting features. (default "")

quick-redisplay

If ON EMACS won't worry so much about the case where you have the same buffer on view in several windows -- it may let the other windows be inaccurate for a short while (but they will eventually be fixed up). Turning this ON speeds up EMACS substantially when the same buffer is on view in several windows. When it is OFF, all windows are always accurate. (default OFF)

replace-case

If ON EMACS will alter the case of strings substituted with *replace-string* or *query-replace-string* to match the case of the original string. For example, replacing "which" by "that" in the string "Which is silly" results in "That is silly"; in the string "the car which is red" results in "the car that is red"; and in the string "WHICH THING?" results in "THAT THING?".

right-margin

The right margin for automatic text justification. If a character is inserted at the end of a line and to the right of the right margin EMACS will automatically insert at the beginning of the preceding word a newline, tabs and spaces to indent to the left margin, and the prefix string. With the right margin set to something like (for eg.) 72 you can type in a document without worrying about when to hit the *return* key, EMACS will automatically do it for you at exactly the right place.

scroll-step

The number of lines by which windows are scrolled if dot moves outside the window. If dot has moved more than *scroll-step* lines outside of the window or *scroll-step* is zero then dot is centered in the window. Otherwise the window is moved up or down *scroll-step* lines. Setting *scroll-step* to 1 will cause the window to scroll by 1 line if you're typing at the end of the window and hit RETURN.

silently-kill-processes

If ON EMACS will kill processes when it exits *without* asking any questions. Normally, if you have processes running when EMACS exits, the question "You have processes on the prowl, should I hunt them down for you" is asked. (default OFF)

stack-trace-on-error

If ON EMACS will write a MLisp stack trace to the "Stack trace" buffer whenever an error is encountered from within an MLisp function (even inside an *error-occured*). This is all there is in the way of a debugging facility. (default OFF)

tab-size

A buffer-specific variable which specifies the number of characters between tab stops. It's not clear that user specifiable tabs are a good idea, since the rest of Unix and most other DEC styled operating systems have the magic number 8 so deeply wired into them. (default 8)

this-command

The meaning of the variable *this-command* is tightly intertwined with the meaning of the function *previous-command*. Look at its documentation for a description of *this-command*.

track-eol-on-↑N-↑P

If ON then ↑N and ↑P will "stick" to the end of a line if they are started there. If OFF ↑N and ↑P will try to stay in the same column as you move up and down even if you started at the end of a line. (default ON)

unlink-checkpoint-files

If ON EMACS will unlink the corresponding checkpoint file after the master copy is written -- this avoids having a lot of .CKP files lying around but it does compromise safety a little. For example, as you're editing a file called "foo.c" EMACS will be periodically be writing a checkpoint file called "foo.c.CKP" that contains all of your recent changes. When you rewrite the file (with ↑X↑F or ↑X↑S for example) if unlink-checkpoint-files is ON then the .CKP file will be unlinked, otherwise it will be left. (default OFF)

visible-bell

If ON EMACS will attempt to use a visible bell, usually a horrendous flashing of the screen, instead of the audible bell, when it is notifying you of some error. This is a more "socially acceptable" technique when people are working in a crowded terminal room. (default OFF)

wrap-long-lines

If ON EMACS will display long lines by "wrapping" their continuation onto the next line (the first line will be terminated with a '\'). If OFF long lines get truncated at the right edge of the screen and a '$' is display to indicate that this has happened. (default OFF)

22. Command summary

Key	Binding
↑@	set-mark
↑A	beginning-of-line
↑B	backward-character
↑C	exit-emacs
↑D	delete-next-character
↑E	end-of-line
↑F	forward-character
↑G	illegal-operation
↑H	delete-previous-character
↑I	self-insert
↑J	newline-and-indent
↑K	kill-to-end-of-line
↑L	redraw-display

↑M	newline
↑N	next-line
↑O	newline-and-backup
↑P	previous-line
↑Q	quote-character
↑R	search-reverse
↑S	search-forward
↑T	transpose-characters
↑U	argument-prefix
↑V	next-page
↑W	delete-to-killbuffer
↑X	↑X-prefix
↑X-↑B	list-buffers
↑X-↑C	exit-emacs
↑X-↑D	describe-word-in-buffer
↑X-↑E	compile-it
↑X-↑F	write-file-exit
↑X-↑I	insert-file
↑X-↑M	write-modified-files
↑X-↑N	next-error
↑X-↑O	use-old-buffer
↑X-↑R	read-file
↑X-↑S	write-current-file
↑X-↑V	visit-file
↑X-↑W	write-named-file
↑X-↑X	exchange-dot-and-mark
↑X-↑Z	shrink-window
↑X-!	execute-monitor-command
↑X-(	start-remembering
↑X-)	stop-remembering
↑X-1	delete-other-windows
↑X-2	split-current-window
↑X-b	switch-to-buffer
↑X-d	delete-window
↑X-e	execute-keyboard-macro
↑X-m	smail
↑X-n	next-window
↑X-p	previous-window
↑X-r	rmail
↑X-z	enlarge-window
↑Y	yank-from-killbuffer
↑Z	scroll-one-line-up
ESC	ESC-prefix
ESC-↑C	exit-emacs
ESC-↑V	page-next-window
ESC-↑W	delete-region-to-buffer
ESC-↑Y	yank-buffer
ESC-ESC	execute-mlisp-line
ESC-↑↑	case-region-invert
ESC-!	line-to-top-of-window
ESC-,	beginning-of-window
ESC--	meta-minus
ESC-.	end-of-window
ESC-0..ESC-9	meta-digit
ESC-<	beginning-of-file
ESC->	end-of-file
ESC-?	apropos
ESC-[	backward-paragraph
ESC-]	forward-paragraph
ESC-↑	case-word-invert
ESC-a	backward-sentence
ESC-b	backward-word
ESC-d	delete-next-word
ESC-e	forward-sentence
ESC-f	forward-word
ESC-h	delete-previous-word
ESC-j	indent-C-procedure
ESC-l	case-word-lower
ESC-q	query-replace-string
ESC-r	replace-string
ESC-u	case-word-upper
ESC-v	previous-page
ESC-x	execute-extended-command
ESC-z	scroll-one-line-down
↑–	return-to-monitor
..+	self-insert
-	minus
../	self-insert
0..9	digit
:..~	self-insert
↑?	delete-previous-character

Index

! 32
!= 32

% 32

& 32
&Default-Transpose-Direction 31
&Default-Transpose-Follow 31
&Default-Transpose-Magic 31
&Occurances-Extra-Lines 25

* 32

+ 32

- 33

/ 33

< 33
<< 33
<= 33

= 33

> 33
>= 33
>> 33

Abbrev-mode 8, 62
Abort-operation 33
Active-process 20, 34
Append-region-to-buffer 34
Append-to-file 34
Appply-look 29
Apropos 5, 34
Arg 34
Argc 34
Argument-prefix 34
Argv 35
Ask-about-buffer-names 64
Auto-execute 35
Autoload 35

Backup-before-writing 64, 65
Backup-by-copying 64, 65
Backup-by-copying-when-linked 64, 65
Backup-when-writing 65
Backward-balanced-paren-line 35
Backward-character 35
Backward-paragraph 36
Backward-paren 36
Backward-sentence 36
Backward-word 36
Baud-rate 36
Begin-C-comment 24
Beginning-of-file 36
Beginning-of-line 36
Beginning-of-window 36
Bind-to-key 9, 16, 36, 44, 50
Bobp 37
Bolp 37
Buff 23
Buffer list 23
Buffer-is-modified 65
Buffer-size 37

C-mode 37
C= 37
Capitalize-word 23
Capword 23
Case-fold-search 37, 65, 66
Case-region-capitalize 37
Case-region-invert 37
Case-region-lower 37
Case-region-upper 37
Case-word-capitalize 37, 50
Case-word-invert 37
Case-word-lower 23, 38
Case-word-upper 23, 38
Cd 26
Change-current-process 20, 38
Change-directory 38
Char-to-string 38
Checkpoint-frequency 65
Clock 31
Command prefix (also known as META) 38
Command-prefix 38
Comment-column 24, 51, 65, 66
Compile-it 38
Concat 39
Continue-process 20, 21, 39, 60
Copy-region-to-buffer 39
Ctlchar-with-↑ 66
Current-buffer-name 39
Current-column 39
Current-file-name 39
Current-indent 39
Current-process 20, 39
Current-time 39

Declare-global 10, 39
Default-case-fold-search 66
Default-comment-column 66
Default-left-margin 66
Default-mode-line-format 19, 66
Default-right-margin 66
Default-tab-size 66
Define-buffer-macro 40, 43
Define-global-abbrev 8, 40
Define-keyboard-macro 9, 40, 44
Define-keymap 16, 40
Define-local-abbrev 8, 40
Define-string-macro 8, 40, 44
Defun 11, 40
Delete-buffer 40
Delete-macro 41

Delete-next-character 41
Delete-next-word 41
Delete-other-windows 41
Delete-previous-character 41
Delete-previous-word 41
Delete-region-to-buffer 41
Delete-to-killbuffer 41
Delete-white-space 41
Delete-window 42
Deleting files 24
Describe-bindings 5, 42
Describe-command 5, 42
Describe-key 5, 42
Describe-variable 42
Describe-word-in-buffer 42
Digit 42
Directory 26
Dired 24
Display-file-percentage 36
Dot 42
Dump-syntax-table 43, 50

Edit-macro 40, 43
Electric-c 8
End-C-comment 24
End-of-file 43
End-of-line 43
End-of-window 43
Enlarge-window 43
Eobp 43
Eolp 43
Eot-process 20, 43
Erase-buffer 43
Erase-region 43
Error-message 44
Error-occured 44, 69
Exchange-dot-and-mark 44
Execute-extended-command 44
Execute-keyboard-macro 8, 44
Execute-mlisp-buffer 44
Execute-mlisp-line 44
Execute-monitor-command 44
Exit-emacs 45
Expand-mlisp-variable 45, 47
Expand-mlisp-word 45, 47
Extend-database-search-list 23, 45

Fetch-database-entry 23, 45
File-exists 45
Files-should-end-with-newline 66
Filter-region 45
Following-char 45
Forward-balanced-paren-line 46
Forward-character 5, 46
Forward-paragraph 46
Forward-paren 46, 50
Forward-sentence 46
Forward-word 46, 50

Get-tty-buffer 46
Get-tty-character 47
Get-tty-command 47
Get-tty-string 47
Get-tty-variable 47
Getenv 47
Global-mode-string 19, 66
Goto-character 47
Goto-line 25
Goto-percent 25
Grab-last-line 26

Help facilities 5, 22, 28, 44, 67
Help-on-command-completion 67

If 47
Illegal-operation 47, 57
Indent-C-procedure 24, 48
Index-entry 29
Info 25
Insert-character 48
Insert-file 48
Insert-string 48
Int-process 20, 48
Interactive 48
Is-bound 48

Justify-paragraph 29, 48

Kill-process 20, 48
Kill-to-end-of-line 48

Last-key-struck 49
Left-margin 66, 67
Length 49
Line-to-top-of-window 49
Lisp 25
Lisp-kill-output 26
List-buffers 49
List-databases 23, 49
List-processes 21, 49
Load 49
Local-bind-to-key 16, 50
Looking-at 14, 50

Mail, sending and receiving 26
Mark 50
Message 50
Mode lines 3, 18, 66, 67
Mode-line-format 19, 67
Mode-string 19, 67
Modify-syntax-entry 50, 62
Move-to-comment-column 51, 65

Nargs 51
Narrow-region 18, 51
Needs-checkpointing 67
Newline 52
Newline-and-backup 52
Newline-and-indent 52
Next-error 52

Next-line 52
Next-page 52
Next-window 52
Novalue 52

Occur 25
Occurances 25
Occurances of a string 25
On-output-procedure 21, 54, 60
One-line-buffer-list 23

Page-next-window 52
Parse-error-messages-in-region 53
Pause-emacs 53
Pop-to-buffer 53, 61
Pop-up-windows 67
Pr-newline 26
Preceding-char 53
Prefix arguments 34, 53, 54, 58, 67, 68
Prefix-argument 53, 54, 67
Prefix-argument-loop 53, 54, 67
Prefix-argument-provided 68
Prefix-string 68
Previous-command 53
Previous-line 54
Previous-page 54
Previous-window 54
Print 54
Process-output 21, 54
Process-status 21, 54
Processes, high level access 25
Progn 10, 54
Provide-prefix-argument 53, 54, 67
Push-back-character 55
Put-database-entry 23, 55
Pwd 26

Query-replace-string 14, 55, 68
Quick-redisplay 68
Quietly-read-abbrev-file 8, 55
Quit-process 21, 55
Quote-character 55

Re-query-replace-string 14, 56
Re-replace-string 14, 56
Re-search-forward 14, 56
Re-search-reverse 14, 56
Read-abbrev-file 8, 55, 56, 63
Read-file 18, 35, 56
Reading mail 27
Receiving mail 27
Recursion-depth 56
Recursive-edit 18, 56
Redraw-display 57
Region restrictions 18, 51, 58, 63
Region-around-match 57
Region-to-process 21, 57
Region-to-string 57
Remove-all-local-bindings 57
Remove-binding 57
Remove-local-binding 57
Replace-case 68
Replace-string 14, 18, 57, 68
Return-prefix-argument 58
Return-to-monitor 53, 58
Right-margin 66, 68

Save-excursion 58, 61
Save-restriction 18, 58
Save-window-excursion 58
Scribe mode 29
Scribe-command 29
Scroll-one-line-down 58
Scroll-one-line-up 59
Scroll-step 68
Search-forward 14, 59
Search-reverse 14, 59
Self-insert 59
Send-eot 26
Send-int-signal 26
Send-quit-signal 26
Send-string-to-terminal 59
Sending mail 27
Set 59
Set-mark 59
Setq 59
Shell 25
Shrink-window 5, 60
Silently-kill-processes 21, 68
Sit-for 60
Spell 30
Split-current-window 60
Stack-trace-on-error 69
Start-filtered-process 21, 60
Start-process 21, 60
Start-remembering 8, 60
Stop-process 20, 21, 39, 60
Stop-remembering 8, 61
String-to-char 61
String-to-process 22, 61
Substr 61
Switch-to-buffer 10, 53, 61
System-name 61

Tab-size 69
Temp-use-buffer 61
Text-mode 31
This-command 69
Time 31
To-col 62
Track-eol-on-↑N-↑P 69
Transp 31
Transpose-characters 62
Transpose-line 31
Transpose-word 31

Unlink-checkpoint-files 69
Unlink-file 62
Use-abbrev-table 8, 40, 62
Use-global-map 16, 62

Use-local-map 16, 62
Use-old-buffer 62
Use-syntax-table 50, 62
Users-full-name 63
Users-login-name 63

Visible-bell 69
Visit-file 9, 35, 63

While 63
Widen-region 18, 63
Word-mode-search 37
Working-directory 63
Wrap-long-lines 69
Write-abbrev-file 8, 55, 56, 63
Write-current-file 18, 63
Write-file-exit 63
Write-modified-files 64
Write-named-file 64
Write-region-to-file 32

Yank-buffer 64
Yank-from-killbuffer 64

↑ 33

| 64