μEmacs/PK 4.0

Full screen editor based on MicroEMACS 3.9e

written by Dave G. Conroy
greatly modified by Daniel M. Lawrence
modified by Petri H. Kutvonen

Copyright notices

MicroEMACS 3.9e © Copyright 1987 by Daniel M. Lawrence. Reference Manual Copyright 1987 by Brian Straight and Daniel M. Lawrence. All Rights Reserved. No copyright claimed for modifications made by Petri H. Kutvonen.

Original statement of copying policy

MicroEMACS 3.9e can be copied and distributed freely for any non-commercial purposes. MicroEMACS 3.9e can only be corporated into commercial software with the permission of the current author [Daniel M. Lawrence].

INTRODUCTION

uEmacs/PK 4.0 is a screen editor for programming and word processing. It is available for the IBM-PC and its clones, UNIX® System V and 4.[23]BSD (including SunOS, DEC Ultrix and IBM AIX), and VAX/VMS. Some of its capabilities include:

- ❑ Multiple windows on screen at one time
- ❑ Multiple files in the editor at once
- ❑ Limited on-screen formatting of text
- ❑ User changeable command set
- ❑ User written editing macros
- ❑ Compatibility across all supported environments

This manual is designed as a reference manual. All the commands in uEmacs are listed, in functional groups, along with detailed descriptions of what each command does.

HOW TO START

uEmacs is invoked from the operating system command level with a command of the form:

emacs {*options*} *filelist*

where options may be:

-v	All the following files are in *View* mode (read only).
-e	All the following files can be edited.
-g*n*	Go directly to line *n* of the first file.
+*n*	Go directly to line *n* of the first file.
-s*string*	Go to the end of the first occurrence of *string* in the first file.
-r	Restricted mode. Prevents uEmacs from executing many of its commands which would allow you to break out of it, or edit files other then the ones named on the command line.
-n	Allow reading of files which contain NULL characters.
@*sfile*	Execute macro file *sfile* instead of the standard startup file.

and *filelist* is a list of files to be edited. For example:

```
emacs @start1.cmd -g56 test.c -v head.h def.h
```

means to first execute macro file start1.cmd instead of the standard startup file, emacs.rc (or .emacsrc on UNIX, see appendix for details) and then read in test.c, position the cursor to line 56, and be ready to read in files head.h and def.h in *View* (read-only) mode. In the simple case, uEmacs is usually run by typing:

emacs *file*

where *file* is the name of the file to be edited.

HOW TO GET HELP AND HOW TO EXIT

Esc ? (or actually Meta ? as you soon will learn) will bring up a short summary of all uEmacs commands. On a VT200 or equivalent you can use the **Help** key. On an IBM-PC try **F1**.

Don't panic if you get stuck within uEmacs. **Ctrl-G** will abort almost any operation and **Ctrl-X Ctrl-C** will get you out of uEmacs.

HOW TO TYPE IN COMMANDS

Most commands in uEmacs are a single keystroke, or a keystroke preceded by a command prefix. Control commands appear in the documentation like **^A** which means to depress the <Ctrl> key and while holding it down, type the A character. Meta commands appear as **Meta A** which means to strike the <Meta> key (<Esc> on most computers) and then after releasing it, type the A character. Control-X commands usually appear as **^X A** which means to hold down the control key and type the X character then type the A character. Both meta commands and control-X commands can be control characters as well, for example, ^X ^O (the *delete-blank-lines* command) means to hold down <Ctrl>, type X, keep holding down <Ctrl> and type the O character.

Many commands in uEmacs can be executed a number of times. In order to make one command repeat many times, type <Meta> (<Esc>) followed by a number, and then the command. for example:

Meta 12 ^K

will delete 12 lines starting at the cursor and going down. Sometimes, the repeat count is used as an argument to the command as in the set-tab command where the repeat count is used to set the spacing of the tab stops.

THE COMMAND LIST

The following is a list of all the commands in uEmacs. Listed is the command name, the default (normal) keystrokes used to invoke it, and alternative keys for the IBM-PC and VT200-series terminals, and a description of what the command does.

Moving the cursor

previous-page ^Z Pg Up Prev Scrn

Move one screen towards the beginning of the file.

next-page ^V Pg Dn Next Scrn

Move one screen towards the end of the file.

beginning-of-file Meta < ^Home

Place the cursor at the beginning of the file.

end-of-file Meta > ^End

Place the cursor at the end of the file.

forward-character ^F ⇒ ⇒

Move the cursor one character to the right. Go down to the beginning of the next line if the cursor was already at the end of the current line.

backward-character ^B ⇐ ⇐

Move the cursor one character to the left. Go to the end of the previous line if the cursor was at the beginning of the current line.

next-word Meta F ^⇒

Place the cursor at the beginning of the next word.

previous-word Meta B ^⇐

Place the cursor at the beginning of the previous word.

beginning-of-line ^A Home

Move cursor to the beginning of the current line.

end-of-line ^E End

Move the cursor to the end of the current line.

next-line ^N ⇓ ⇓

Move the cursor down one line.

previous-line ^P ⇑ ⇑

Move the cursor up one line.

goto-line Meta G

Goto a specific line in the file. I.e. Meta 65 Meta G would put the cursor on the 65th line of the current buffer.

next-paragraph Meta N ^⇓

Put the cursor at the first end of paragraph after the cursor.

previous-paragraph Meta P ^⇑

Put the cursor at the first beginning of paragraph before the cursor.

Deleting and inserting

delete-previous-character ^H ←

Delete the character immediately to the left of the cursor. If the cursor is at the beginning of a line, this will join the current line on the end of the previous one.

delete-next-character ^D Del

Delete the character the cursor is on. If the cursor is at the end of a line, the next line is put at the end of the current one.

delete-previous word Meta ^H Meta ←

Delete the word before the cursor.

delete-next-word Meta D

Delete the word starting at the cursor.

kill-to-end-of-line ^K

When used with no argument, this command deletes all text from the cursor to the end of a line. When used on a blank line, it deletes the blank line. When used with an argument, it deletes the specified number of lines.

insert-space ^C Ins

Insert a space before the character the cursor is on.

newline Return Enter

Insert a newline into the text, move the cursor down to the beginning of the next physical line, carrying any text that was after it with it.

newline-and-indent ^J

Insert a newline into the text, and indent the new line the same as the previous line.

handle-tab ^I →| Tab

With no argument, move the cursor to the beginning of the next tab stop. With an argument of zero, use real tab characters when tabbing. With a non-zero argument, use spaces to tab every argument positions.

delete-blank-lines ^X ^O

Delete all the blank lines before and after the current cursor position.

trim-line ^X ^T

Delete trailing white space from current line.

detab-line ^X ^A

Change tabulator characters to appropriate number of spaces on current line.

entab-line ^X ^E

Change spaces to tabulator characters where possible on current line.

kill-paragraph Meta ^W

Delete the paragraph that the cursor is currently in.

kill-region ^W ^Del Remove

Delete all the characters from the cursor to the mark set with the set-mark command.

copy-region Meta W

Copy all the characters between the cursor and the mark set with the set-mark command into the kill buffer (so they can later be yanked elsewhere).

open-line ^O

Insert a newline at the cursor, but do not move the cursor.

Searching

search-forward ^S Meta S

Search for a string from the current cursor position to the end of the file. The string is typed on on the bottom line of the screen, and terminated with the <Meta> key. Special characters can be typed in by preceding them with a ^Q or ^V. A single ^Q or ^V indicates a null string. On successive searches, hitting <Meta> alone causes the last search string to be reused. *Note: The command ^S cannot be used if your terminal uses XON-XOFF-flow control. Use the second form or incremental-search instead.*

search-reverse ^R

This command searches backwards in the file. In all other ways it is like search-forward.

incremental-search ^X S Find

This command is similar to forward-search, but it processes the search as each character of the input string is typed in. This allows the user to only use as many keystrokes as are needed to uniquely specify the string being searched. Several control characters are active while i-searching:

^S or ^X	Skip to the next occurrence of the current string
^R	Skip to the last occurrence of the current string
^H or ←	Back up to the last match (possibly deleting the last character on the search string)
^G	Abort the search, return to start
Meta	End the search, stay here

Always remember to terminate the search by hitting <Meta> (or ^G).

reverse-incremental-search ^X R

This command is the same as incremental-search, but it starts in the reverse direction.

hunt-forward unbound (<Alt> S on the IBM PC)

This command repeats the last search with the last search string

hunt-backward unbound (<Alt> R on the IBM PC)

The last search string is looked for starting at the cursor and going backwards.

Replacing

replace-string Meta R

This command allows you to replace all occurrences of one string with another string. The replacement starts at the current location of the cursor and goes to the end of the current buffer. A numeric argument will limit the number of strings replaced.

query-replace-string Meta ^R

Like the replace-string command, this command will replace one string with another. However, it allows you to step through each string and ask you if you wish to make the replacement. When the computer asks if you wish to make the replacement, the following answers are allowed:

Y	Make the replacement and continue on to the next string
N	Don't make the replacement, then continue
!	Replace the rest of the strings without asking
^G	Stop the command
.	Go back to place the command started
?	Get a list of options

Capitalizing and transposing

case-word-upper Meta U

Change the following word into upper case.

case-word-capitalize Meta C

Capitalize the following word.

case-word-lower Meta L

Change the following word to lower case.

case-region-upper ^X ^U

Change all the alphabetic characters in a marked region to upper case.

case-region-lower ^X ^L

Change all the alphabetic characters in a marked region to lower case.

transpose-characters ^T

Swap the last and second last characters behind the cursor.

quote-character ^Q ^X Q

Insert the next typed character, ignoring the fact that it may be a command character. *Note: This command may cause problems if XON-XOFF-flow control is used.*

Regions and the kill buffer

set-mark Meta Space Select

This command is used to delimit the beginning of a marked region. Many commands are effective for a region of text. A region is defined as the text between the mark and the current cursor position. To delete a section of text, for example, one moves the cursor to the beginning of the text to be deleted, issues the set-mark command by typing <Meta> <Space>, moves the cursor to the end of the text to be deleted, and then deletes it by using the kill-region (^W) command. Only one mark can be set in one window or one buffer at a time, and uEmacs will try to remember a mark set in an offscreen buffer when it is called back on screen.

exchange-point-and-mark ^X ^X

This command moves the cursor to the current marked position in the current window and moves the mark to where the cursor was. This is very useful in finding where a mark was, or in returning to a position previously marked.

Copying and moving

kill-region ^W Remove

This command is used to copy the current region (as defined by the current mark and the cursor) into the kill buffer.

yank ^Y ^Ins Insert Here

This copies the contents of the kill buffer into the text at the current cursor position. This does not clear the kill buffer, and thus may be used to make multiple copies of a section of text.

copy-region Meta W

This command copies the contents of the current region into the kill buffer without deleting it from the current buffer.

Modes of operation

add-mode ^X M

Add a mode to the current buffer

delete-mode ^X ^M

Delete a mode from the current buffer

add-global-mode Meta M

Add a mode to the global modes which get inherited by any new buffers that are created while editing.

delete-global-mode Meta ^M

Delete a mode from the global mode list. This mode list is displayed as the first line in the output produced by the list-buffers command.

Modes are assigned to all buffers that exist during an editing session. These modes effect the way text is inserted, and the operation of some commands. Legal modes are:

Over - Overwrite mode

In this mode, typed characters replace existing characters rather than being inserted into existing lines. Newlines still insert themselves, but all other characters will write over existing characters on the current line being edited. This mode is very useful for editing charts, figures, and tables.

Wrap - Word wrap mode

In this mode, when the cursor crosses the current fill column (which defaults to 72) it will, at the next word break, automatically insert a newline, dragging the last word down with it. This makes typing prose much easier since the newline (<Return>) only needs to be used between paragraphs.

View - File viewing (read-only) mode

In this mode, no commands which can change the text are allowed.

Cmode - C program editing mode

This mode is for editing programs written in the 'C' programming language. When the newline is used, the editor will attempt to place the cursor at the proper indentation level on the next line. Close braces are automatically un-idented for the user, and also pre-processor commands are automatically set flush with the left margin. When a close parenthesis or brace is typed, if the matching open is on screen, the cursor briefly moves to it, and then back. (Typing any key will abort this fence matching, executing the next command immediately)

Exact - Exact case matching on searching

Normally case is insignificant during the various search commands. This forces all matching to take character case into account.

Magic - Regular expression pattern matching

This feature causes search commands to accept various pattern characters to allow regular expression search and replaces. See chapter "The Magic Mode" for details.

Asave - Automatic save

This causes uEmacs to write your current file on disk when a certain number (default 256) of new characters have been entered.

On-screen formatting

set-fill-column ^X F

Sets the column used by *Wrap* mode and *fill-paragraph* and *justify-paragraph* commands.

handle-tab ^I →| Tab

Given a numeric argument, the tab key resets the normal behavior of the tab key. An argument of zero causes the tab key to generate hardware tabs (at each 8 or 4 columns, see $tab variable). A non-zero argument will cause the tab key to generate enough spaces to reach a column of a multiple of the argument given. This also resets the spacing used while in *Cmode*.

fill-paragraph Meta Q

This takes all the text in the current paragraph (as defined by surrounding blank lines, or a leading indent) and attempt to fill it from the left margin to the current fill column.

justify-paragraph Meta J

This is a modified version of *fill-paragraph.* The left margin is taken to be the current column. No extra white space is inserted after punctuation. The cursor is moved to the beginning of next paragraph.

buffer-position ^X =

This command reports on the current and total lines and characters of the current buffer. It also gives the hexadecimal code of the character currently under the cursor.

Multiple windows

split-current-window ^X 2

If possible, this command splits the current window into two near equal windows, each displaying the buffer displayed by the original window. A numeric argument of 1 forces the upper window to be the new current window, and an argument of 2 forces the lower window to be the new current window.

delete-window ^X 0

This command attempts to delete the current window, retrieving the lines for use in the window above or below it.

delete-other-windows ^X 1

All other windows are deleted by this command. The current window becomes the only window, using the entire available screen.

next-window ^X O

Make the next window down the current window. With an argument, this makes the *n*th window from the top current.

previous-window ^X P

Make the next window up the current window. With an argument, this makes the *n*th window from the bottom the current window.

scroll-next-down Meta ^V

Scroll the next window down a page.

scroll-next-up Meta ^Z

Scroll the next window up a page.

Controlling windows

grow-window ^X ^

Enlarge the current window by the argument number of lines (1 by default).

shrink-window ^X ^Z

Shrink the current window by the argument number of lines (1 by default).

resize-window ^X W

Change the size of the current window to the number of line specified by the argument, if possible.

move-window-down ^X ^N

Move the window into the current buffer down by one line.

move-window-up ^X ^P

Move the window into the current buffer up by one line.

redraw-display Meta ^L

Redraw the current window with the current line in the middle of the window, or with an argument, with the current line on the *n*th line of the current window.

clear-and-redraw ^L

Clear the screen and redraw the entire display. Useful on timesharing systems where messages and other things can garbage the display.

Multiple buffers

select-buffer ^X B

Switch to using another buffer in the current window. uEmacs will prompt you for the name of the buffer to use.

next-buffer ^X X

Switch to using the next buffer in the buffer list in the current window.

name-buffer Meta ^N

Change the name of the current buffer.

delete-buffer ^X K

Dispose of an undisplayed buffer in the editor and reclaim the space. This does not delete the file the buffer was read from.

list-buffers ^X ^B

Split the current window and in one half bring up a list of all the buffers currently existing in the editor. The active modes, change flag, and active flag for each buffer is also displayed. (The change flag is an * if the buffer has been changed and not written out. The active flag is not an @ if the file had been specified on the command line, but has not been read in yet since nothing has switched to that buffer.)

Reading from disk

find-file ^X ^F

Find the named file. If it is already in a buffer, make that buffer active in the current window, otherwise attempt to create a new buffer and read the file into it.

read-file ^X ^R

Read the named file into the current buffer (overwriting the previous contents of the current buffer. If the change flag is set, a confirmation will be asked).

insert-file ^X ^I

Insert the named file into the current position of the current buffer.

view-file ^X ^V

Like find-file, this command either finds the file in a buffer, or creates a new buffer and reads the file in. In addition, this leaves that buffer in *View* mode.

Automatic file name completion

File name completion can be used with all file oriented commands (*find-file, view-file, ...*) but it works only under UNIX and MS-DOS. It is invoked by a <Space> or <Tab>. If there exist more than one possible completions they are displayed one by one. If the file name contains wild card characters, the name is expanded instead of simple completion. Special characters can be entered verbatim by prefixing them with ^V (or ^Q).

Saving to disk

save-file ^X ^S ^X ^D

If the contents of the current buffer have been changed, write it back to the file it was read from. *Use ^X ^D if your terminal uses XON-XOFF-flow control.*

write-file ^X ^W

Write the contents of the current file to the named file, this also changes the file name associated with the current buffer to the new file name.

change-file-name ^X N

Change the name associated with the current buffer to the file name given.

quick-exit Meta Z

Write out all changed buffers to the files they were read from and exit the editor. This is the normal way to exit uEmacs.

Accessing the operating system

shell-command ^X !

Send one command to execute to the operating system command processor, or shell. Upon completion, uEmacs will wait for a keystroke to redraw the screen.

pipe-command ^X @

Execute one operating system command and pipe the resulting output into a buffer by the name of "command".

filter-buffer ^X #

Execute one operating system command, using the contents of the current buffer as input, and sending the results back to the same buffer, replacing the original text.

i-shell ^X C

Push up to a new command processor or shell. Upon exiting the shell, uEmacs will redraw its screen and continue editing.

suspend-emacs ^X D (only under 4.[23]BSD)

This command suspends the editing processor and puts it into the background. The "fg" command will restart uEmacs.

exit-emacs ^X ^C

Exit uEmacs back to the operating system. If there are any unwritten, changed buffers, the editor will promt to discard changes.

Key bindings and commands

bind-to-key Meta K

This command takes one of the named commands and binds it to a key. From then on, whenever that key is struck, the bound command is executed.

unbind-key Meta ^K

This unbinds a command from a key.

describe-key ^X ?

This command will allow you to type a key and it will then report the name of the command bound to that key.

execute-named-command Meta X

This command will prompt you for the name of a command to execute. Typing <Space> part way through will tell the editor to attempt to complete the name on its own. If it then beeps, there is no such command to complete.

describe-bindings unbound

This command splits the current window, and in one of the windows makes a list of all the named commands, and the keys currently bound to them.

apropos Meta A

This command is a modification of the command *describe-bindings.* It lists all named commands that match a given substring.

Command execution

Commands can also be executed as command scripts. This allows commands and their arguments to be stored in files and executed. The general form of a command script line is:

{*optional repeat count*} *command-name* {*optional arguments*}

execute-command-line unbound

Execute a typed in script line.

execute-buffer unbound

Executes script lines in the named buffer. If the buffer is off screen and an error occurs during execution, the cursor will be left on the line causing the error.

execute-file unbound

Executes script lines from a file. This is the normal way to execute a special script.

clear-message-line unbound

Clears the message line during script execution. This is useful so as not to leave a confusing message from the last commands in a script.

write-message unbound

Write a message on the command line.

unmark-buffer unbound

Remove the change flag from the current buffer. This is very useful in scripts where you are creating help windows, and don't want uEmacs to complain about not saving them to a file.

insert-string unbound

Insert a string into the current buffer. This allows you to build up text within a buffer without reading it in from a file. Some special characters are allowed, as follows:

~n	newline
~t	tab
~b	backspace
~f	formfeed

overwrite-string unbound

Insert or overwrite a string into the current buffer. Works like *insert-string* in *Over* mode.

Screen size

On an IBM-PC the screen size is controlled by the monitor type. Although 25x80 is the default, 43x80 can be used on an EGA screen and 52x80 on a VGA screen. The screen size is set automatically when setting the screen resolution variable $sres to "EGA" or "VGA" (see Environmental variables).

Under UNIX and VMS the default screen size depends on terminal settings used by the operating system. If you use a window system under UNIX, uEmacs can even detect dynamic changes of virtual screen size and resize itself accordingly.

However, the screen size can be controlled manually by the following two commands:

change-screen-size Meta ^D

Change the number of lines on screen.

change-screen-width Meta ^T

Change the number of columns on screen.

Keyboard macro execution

Also available is one keyboard macro, which allows you to record a number of commands as they are executed and play them back.

begin-macro ^X (

Start recording keyboard macro.

end-macro ^X)

Stop recording keyboard macro.

execute-macro ^X E

Execute keyboard macro.

THE MAGIC MODE

In the Magic mode of uEmacs certain characters gain special meanings when used in a search pattern. Collectively they are know as regular expressions, and a limited number of them are supported in uEmacs. They grant greater flexibility when using the search command. However, they do not affect the incremental search command.

The symbols that have special meaning in *Magic* mode are ^, $, ., *, [(and], used with it), and \ . The characters ^ and $ fix the search pattern to the beginning and end of line, respectively. The ^ character must appear at the beginning of the search string, and the $ must appear at the end, otherwise they loose their meaning and are treated just like any other character.

For example, in *Magic* mode, searching for the pattern "t$" would put the cursor at the end of any line that ended with the letter 't'. Note that this is different than searching for "t<NL>", that is, 't' followed by a newline character. The character $ (and ^, for that matter) matches a position, not a character, so the cursor remains at the end of the line. But a newline is a character that must be matched, just like any other character, which means that the cursor is placed just after it - on the beginning of the next line.

The character . has a very simple meaning - it matches any single character, except the newline. Thus a search for "bad.er" could match "badger", "badder" (slang), or up to the 'r' of "bad error".

The character * is known as closure, and means that zero or more of the preceding character will match. If there is no character preceding, * has no special meaning, and since it will not match with a newline, * will have no special meaning if preceded by the beginning of line symbol ^ or the literal newline character <NL>.

The notion of zero or more characters is important. If, for example, your cursor was on the line

```
This line is missing two vowels.
```

and a search was made for "a*", the cursor would not move, because it is guaranteed to match no letter 'a' , which satisfies the search conditions. If you wanted to search for one or more of the letter 'a', you would search for "aa*", which would match the letter a, then zero or more of them.

The character [indicates the beginning of a character class. It is similar to the 'any' character ., but you get to choose which characters you want to match. The character class is ended with the character]. So, while a search for "ba.e" will match "bane", "bade", "bale", "bate", etc, you can limit it to matching "babe" and "bake" by searching for "ba[bk]e". Only one of the characters inside the [and] will match a character. If in fact you want to match any character except those in the character class, you can put a ^ as the first character. It must be the first character of the class, or else it has no special meaning. So, a search for [^aeiou] will match any character except a vowel, but a search for [aeiou^] will match any vowel or a ^.

If you have a lot of characters in order that you want to put in the character class, you may use a dash (-) as a range character. So, [a-z] will match any letter (or any lower case letter if *Exact* mode is on), and [0-9a-f] will match any digit or any letter 'a' through 'f', which happen to be the characters for hexadecimal numbers. If the dash is at the beginning or end of a character class, it is taken to be just a dash.

The escape character \ is for those times when you want to be in *Magic* mode, but also want to use a regular expression character to be just a character. It turns off the special meaning of the character. So a search for "it \ ." will search for a line with "it.", and not "it" followed by any other character. The escape character will also let you put ^, -, or] inside a character class with no special side effects.

uEmacs MACROS

Macros are programs that are used to customize the editor and to perform complicated editing tasks. They may be stored in files or buffers and may be executed using an appropriate command, or bound to a particular keystroke. The *execute-macro-<n>* editor commands cause the macros, numbered from 1 to 40, to be executed. Macros are stored by executing files that contain the *store-macro* command. The macro number is given to the *store-macro* command as an argument. All script lines then encountered will be stored rather than being executed.

There are many different aspects to the macro language within uEmacs. Editor commands are the various commands that manipulate text, buffers, windows, etc, within the editor. Directives are commands which control what lines get executed within a macro. Also there are various types of variables. Environmental variables both control and report on different aspects of the editor. User variables hold string values which may be changed and inspected. Buffer variables allow text to be placed into variables. Interactive variables allow the program to prompt the user for information. Functions can be used to manipulate all these variables.

Variables

Variables in uEmacs can be used to return values within expressions, as repeat counts to editing commands, or as text to be inserted into buffers and messages. The value of these variables is set using the *set* (^X A) command. For example, to set the current fill column to 64 characters, the following macro line would be used:

```
set $fillcol 64
```

or to have the contents of %name inserted at the point in the current buffer, the command to use would be:

```
insert-string %name
```

Environmental variables

"What good is a quote if you can't change it?"

These variables are used to change different aspects of the way the editor works. Also they will return the current settings if used as part of an expression. All environmental variable names begin with a dollar sign ($).

$pagelen	Number of screen lines used currently
$curwidth	Number of columns used currently
$curcol	Current column of point in current buffer
$curline	Current line of point in current buffer
$cbufname	Name of the current buffer
$cfname	File name of the current buffer
$wline	Number of lines in current window
$cwline	Current line in window
$fillcol	Current fill column (*Wrap* mode, *fill-paragraph, justify-paragraph*)
$lwidth	Width of current line
$line	Text of current line
$curchar	Current character under the cursor
$flicker	Flicker flag, set to TRUE if IBM CGA or old AT&T/Olivetti, set to FALSE for most others
$sres	Current screen resolution: CGA, MONO, EGA, or VGA on the IBM-PC, NORMAL on all others
$debug	Flag to trigger macro debugging
$discmd	If TRUE, display commands on command line
$status	Return status of the success of the last command (TRUE or FALSE) usually used with !force
$asave	Number of characters between auto-saves in *Asave* mode
$acount	Number of characters until next auto-save
$version	uEmacs version number
$progname	Returns program name, "uEmacs/PK"
$search	Search pattern
$replace	Replacement pattern, can e.g. be set to an empty string
$match	Last matched pattern in *Magic* mode
$kill	Kill buffer (read only)
$tab	"Hard" tabulator stop, 8 or 4, use the command *handle-tab* to set other (soft) tabulator stops
$scroll	Scrolling flag, TRUE if your screen can scroll, FALSE otherwise, can be set to FALSE if you don't like uEmacs's way of scrolling
$jump	Number of lines to scroll when top or end of screen has been reached, default 1, the value 0 has a special meaning: scroll 1/2 page (which is the default if scrolling is not enabled)
$overlap	Number of overlapping lines when browsing files with commands *next-page* and *previous-page,* 2 is a typical value, default is 0 which has a special meaning: 1/3 page

User variables

User variables allow you, the user, to store strings and manipulate them. These strings can be pieces of text, numbers (in text form), or the logical values TRUE and FALSE. These variables can be combined, tested, inserted into buffers, and otherwise used to control the way your macros execute. Up to 100 user variables may be in use in one editing session. All users variable names must begin with a percent sign (%) and may contain any printing characters. Only the first 10 characters are significant (i.e. differences beyond the tenth character are ignored). Most operators will truncate strings to a length of 128 characters.

Buffer variables

Buffer variables are special in that they can only be queried and cannot be set. What buffer variables are is a way to take text from a buffer and place it in a variable. For example, if you have a buffer by the name of *rigel2*, and it contains the text:

```
Richmond
Lafayette
<*>Bloomington                    (where <*> is the current point)
Indianapolis
Gary
-* uEmacs/PK 4.0: rigel2 (Wrap) /data/rigel2.txt ---------------- All --
```

and within a command you reference *#rigel2*, like:

```
insert-string #rigel2
```

uEmacs would start at the current point in the *rigel2* buffer and grab all the text up to the end of that line and pass that back. Then it would advance the point to the beginning of the next line. Thus, after our last command executes, the string "Bloomington" gets inserted into the current buffer, and the buffer *rigel2* now looks like this:

```
Richmond
Lafayette
Bloomington
<*>Indianapolis                   (where <*> is the current point)
Gary
-* uEmacs/PK 4.0: rigel2 (Wrap) /data/rigel2.txt ---------------- All --
```

As you have probably noticed, a buffer variable consists of the buffer name, preceded by a pound sign (#).

Interactive variables

Interactive variables are actually a method to prompt the user for a string. This is done by using an at sign (@) followed either with a quoted string, or a variable containing a string. The string is the placed on the bottom line, and the editor waits for the user to type in a string. Then the string typed in by the users is returned as the value of the interactive variable. For example:

```
set %quest "What file? "
find-file @%quest
```

will ask the user for a file name, and then attempt to find it.

Functions

Functions can be used to manipulate variables in various ways. Functions can have one, two, or three arguments. These arguments will always be placed after the function on the current command line. For example, if we wanted to increase the current fill column by two, using uEmacs's set (^X A) command, we would write:

```
set $fillcol &add $fillcol 2
 \       \       \       \       \____second operand
  \       \       \       \_________first operand
   \       \       \_________________function to execute
    \       \_____________________________variable to set
     \_____________________________________set (^X A) command
```

Function names always begin with the ampersand (&) character, and are only significant to the first three characters after the ampersand. Functions will normally expect one of three types of arguments, and will automatically convert types when needed.

num An ascii string of digits which is interpreted as a numeric value. Any string which does not start with a digit or a minus sign (-) will be considered zero.

str An arbitrary string of characters. Strings are limited to 128 characters in length.

log A logical value consisting of the string "TRUE" or "FALSE". Numeric strings will also evaluate to "FALSE" if they are equal to zero, and "TRUE" if they are non-zero. Arbitrary text strings will have the value of "FALSE".

Numeric functions: (return num)

&add	*num num*	Add two numbers.
&sub	*num num*	Subtract the second number from the first.
&times	*num num*	Multiply two numbers.
&divide	*num num*	Divide the first number by the second giving an integer result.
&mod	*num num*	Return the reminder of dividing the first number by the second.
&negate	*num*	Multiply the arg by -1.
&abs	*num*	Absolute value.

String manipulation functions: (return str)

&cat	*str str*	Concatenate the two strings to form one.
&left	*str num*	Return the num leftmost characters from *str*.
&right	*str num*	Return the num rightmost characters from *str*.
&mid	*str num1 num2*	Starting from *num1* position in *str*, return *num2* characters.
&upper	*str*	Uppercase *str.*
&lower	*str*	Lowercase *str.*
&chr	*num*	Return a single character with character code *num.*
&env	*str*	Retrieve a system environment variable.
&binding	*str*	Look up what function name is bound to key *str.*
&gtkey	*str*	Waits and returns next keystroke, argument not used.
&find	*str*	Look for a file *str* on the search path, return the full name.

Logical and testing functions: (return log)

&not	*log*	Return the opposite logical value.
&and	*log1 log2*	If both *log1* and *log2* are TRUE, return TRUE.
&or	*log1 log2*	If either *log1* or *log2* is TRUE, return TRUE.
&equal	*num1 num2*	If *num1* and *num2* are numerically equal, return TRUE.
&less	*num1 num2*	If *num1* is less than *num2*, return TRUE.
&greater	*num1 num2*	If *num1* is greater than *num2*, return TRUE.
&sequal	*str1 str2*	If the two strings are the same, return TRUE.
&sless	*str1 str2*	If *str1* is less alphabetically than *str2*, return TRUE.
&sgreater	*str1 str2*	If *str1* is alphabetically greater than or *str2*, return TRUE.
&exist	*str*	If file *str* exists, return TRUE.

Other string functions (return num) and special functions:

&sindex	*str1 str2*	Returns position of substring *str2* within *str1,* or 0 if not found.
&ascii	*str*	Returns ASCII code of first character of *str.*
&indirect	*str*	Evaluate *str* as a variable.

This last function deserves more explanation. The &IND function evaluates its argument, takes the resulting string, and then uses it as a variable name. For example, given the following code sequence:

```
; set up reference table
set %one      "elephant"
set %two      "giraffe"
set %three    "donkey"

set %index    "two"
insert-string &ind %index
```

the string "giraffe" would have been inserted at the point in the current buffer. This indirection can be safely nested up to about 10 levels.

Directives

Directives are commands which only operate within an executing macro, i.e. they do not make sense as a single command. As such, they cannot be called up singly or bound to keystroke. Used within macros, they control what lines are executed and in what order.

Directives always start with the exclamation mark (!) character and must be the first thing placed on a line. Directives executed singly (via the execute-command-line command) interactively will be ignored.

!ENDM directive

This directive is used to terminate a macro being stored. For example, if a file being executed contains the text:


```
;       Read in a file in view mode, and make the window red
26      store-macro
        find-file @"File to view: "
        add-mode "view"
        add-mode "red"
!endm
write-message "[Consult macro has been loaded]"
```

only the lines between the store-macro command and the !ENDM directive are stored in macro 26.

!FORCE directive

When uEmacs executes a macro, if any command fails, the macro is terminated at that point. If a line is preceded by a !FORCE directive, execution continues weather the command succeeds or not.

```
;       Merge the top two windows
save-window           ;remember what window we are at
1 next-window         ;go to the top window
delete-window         ;merge it with the second
!force restore-window ;This will continue
add-mode "red"
```

!IF, !ELSE, and !ENDIF directives

The !IF directive allows statements only to be executed if a condition specified in the directive is met. Every line following the !IF directive, until the first !ELSE or !ENDIF directive, is only executed if the expression following the !IF directive evaluates to a TRUE value. For example, the following macro segment creates the portion of a text file automatically.

```
!if &sequal %curplace "timespace vortex"
        insert-string "First, rematerialize~n"
!endif
!if &sequal %planet "earth"
        !if &sequal %time "late 20th century"
                write-message "Contact U.N.I.T."
        !else
                insert-string "Investigate the situation....~n"
                insert-string "(SAY 'stay here Sara')~n"
        !endif
!else
        set %conditions @"Atmosphere conditions outside? "
        !if &sequal %conditions "safe"
                insert-string &cat "Go outside......" "~n"
                insert-string "lock the door~n"
        !else
                insert-string "Dematerialize... try somewhen else"
                newline
        !endif
!endif
```

!GOTO directive

Flow can be controlled within a uEmacs macro using the !GOTO directive. It takes as an argument a label. A label consists of a line starting with an asterisk (*) and then an alphanumeric label. Only labels in the currently executing macro can be jumped to, and trying to jump to a non-existing label terminates execution of a macro.

```
;       Create a block of DATA statements for a BASIC program
        insert-string "1000 DATA "
        set %linenum 1000
*nxtin
        update-screen ;make sure we see the changes
        set %data @"Next number: "
        !if &equal %data 0
                !goto finish
        !endif
        !if &greater $curcol 60
                2 delete-previous-character
                newline
                set %linenum &add %linenum 10
                insert-string &cat %linenum " DATA "
        !endif
        insert-string &cat %data ", "
        !goto nxtin
*finish
        2 delete-previous-character
        newline
```

!RETURN directive

The !RETURN directive causes the current macro to exit, either returning to the caller (if any) or to interactive mode. For example:

```
;       Check the monitor type and set %mtyp
!if &sequal $sres "MONO"
        set %mtyp 1
        !return
!else
        set %mtyp 2
!endif
insert-string "You are on a MONOCHROME machine!~n"
```

!WHILE and !ENDWHILE directives

The !WHILE directive causes a block of of statements to be executed while a condition specified in the directive is TRUE. Every line following the !WHILE directive until the first !ENDWHILE directive belongs to body of the loop. For example:

```
;       Now we know that emacs.hlp is visible, switch to it
!while &not &sequal $cbufname "emacs.hlp"
        next-window
!endwhile
```


APPENDIX: Search order for the initialization file

Under UNIX uEmacs searches for its initialization file in the following order:

1. File given with the @-option on the command line.
2. *.emacsrc* in the current directory.
3. *.emacsrc* in your home directory.
4. System *.emacsrc* in a standard place (compiled into the program).

If any of the files 1-3 is found the system initialization file will **not** be executed. The system *.emacsrc* tries to execute two additional initialization files **after** the system initialization file, namely:

5. *.emrc* in your home directory.
6. *.emrc* in the current directory.

The preferred method for modifying your uEmacs environment is to create an *.emrc* file in your home directory where you can put any **additional commands** you want to be executed. You can further tailor this by creating *.emrc* files in any subdirectories.

Under MS-DOS the search order is similar but a little bit simpler:

1. File given with the @-option on the command line.
2. *emacs.rc* in the current directory.
3. System *emacs.rc,* along the DOS %path, usually in the same directory as uEmacs.

The system *emacs.rc* tries then to execute the file:

4. *em.rc* in the current directory.