

# Evi - Emulate/Enhanced vi for GNU Emacs

---

January 1993, Evi Version 0.99.7

Jeffrey R. Lewis

---

Copyright © 1992, 1993 Jeffrey R. Lewis

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the same conditions as for modified versions.

Evi is a vi emulator aimed at those who are either just plain accustomed to vi, or who just simply happen to like its style of editing better than emacs' default. Evi's first goal is vi compatibility. Its second goal is to be an extension of vi, taking advantage of features supplied by the emacs environment, without simply becoming emacs with vi'sh key bindings.

Ideally, you shouldn't need any special manual to start using Evi, other than your vi manual. By default evi is set-up to emulate vi as closely as possible, including using your EXINIT environment variable and/or .exrc startup files. Of course, you'll need to know how to get started with Evi, and you'll probably want to take advantage of some of Evi's extensions, and be aware of what's different between vi and Evi. That's what this manual covers.

A note on conventions used in this manual: *M-x* means the escape key followed by the character x (where x is any character), *C-x* means control-x, *RET* means the return key, *ESC* means the escape key.

## 0.1 Setting up Evi

To just test Evi out, type:

```
M-x load-file RET evi.el RET
M-x evi RET
```

(don't type the spaces) You will now be in the emulator.

If you have any trouble at this point due to something in your .exrc or EXINIT, and wish to suppress sourcing of .exrc and EXINIT, place the following in your .emacs:

```
(setq evi-supress-ex-startup t)
```

Then, send me a note with the contents of your .exrc and EXINIT so we can fix the problem.

If you decide to continue using Evi, I would recommend first you byte compile it to speed things up, using the following shell command:

```
emacs -batch -l evi.el -f batch-byte-compile evi.el
```

Next, if you want to use Evi all the time, put the following at the end of your .emacs file:

```
(load "<wherever-evi-is>/evi")
(setq term-setup-hook 'evi)
```

this will make emacs go into Evi every time you fire it up. Of course, you may wish to have Evi properly installed for all to use - consult your local emacs guru.

## 0.2 File and buffer management

Vi's file management commands have long tried to mimic having multiple buffers, and as such, work well enough to use in an emacs setting. They of course have to take on slightly different meanings, since it makes little sense to emulate the limitations of the vi/ex model that presumably you are trying to avoid by using Evi!

**:e**            Edit a file in the current window. With no argument, brings in a new copy of the file (useful if it has been subsequently modified on disk). **:e!** will override any complaints about the current buffer being modified, and discards all

modifications. With a filename argument, edits that file in the current window (using the copy already in the editor if it was previously read in). There is no difference between `:e! filename` and `:e filename`, because in Evi we don't need to worry about the disposition of the current file before editing the next one. Use `:e#` as a shorthand for editing the most recently accessed buffer not visible in a window.

- `:E` Same as `:e`, but edits the file in another window, creating that window if necessary. If used with no filename, this command splits the current buffer into two windows.
- `:n` Switch to the next file in buffer list that's not currently displayed. Rotates the current file to the end of the buffer list, so repeated use of `:n` will cycle thru all buffers.
- `:N` Same as `:n`, but switches to another window, or creates another window and puts the next file into it.
- `:b` This is an extension. This command switches the current window to the specified buffer, e.g. `:b foo` would switch to the buffer named `foo` (if it exists). By default `:b` switches to the next buffer not displayed. `:b!` will create the buffer if it doesn't exist.
- `:B` Analogous to `:E` and `:N`.
- `:k` Another extension. This command kills the named buffer. If given no argument, kills the current buffer. `:k!` will kill the buffer even if it is modified.
- `:wk` Writes the current buffer, and kills it.
- `:W` Another extension. This command writes all modified buffers, querying the user before each write. `:W!` will write all buffers with no questions asked.
- `:Wq` As above, then exits emacs. See `ZZ` below.

The ex commands which accept filenames as arguments can be file completed using space or tab. Similarly, `:b` and `:k` will buffer-name complete. One thing you might find handy is `:b TAB` to see a list of buffers. Use `C-c` or backspace to escape this.

## 0.3 Window management

Vi had commands for multiple file management, but it never tried to pretend it handled multiple windows. However, it did have the `z` command for simple window management, like making the window bigger or smaller. In Evi, the `z` command is extended to handle most Emacs window management needs. Additional suggestions are welcome.

- `z0=` Delete the current window from the screen.
- `z1=` Make the current window fill the screen.
- `z2=` Split the current window in two vertically.
- `z0|` Same as `z0=`.
- `z1|` Same as `z1=`.

**z2|** Split the current window in two horizontally.

**z<num>+**

**z<num>-** These two let you adjust the size of the current window by <num>. Use **z<num>.** to adjust the window size absolutely.

**zf** Go to the next window (forward).

**zn** Same as **zf**.

**zb** Go to the previous window (backward).

**zp** Same as **zb**.

**zH**

**zM**

**zL** These are aliases for **zRET**, **z.**, and **z-** and correspond to the arguments to the mark command.

## 0.4 Accessing emacs commands

By default, no emacs commands are accessible from Evi. However, the three main classes of emacs commands can easily be made accessible by adding selected elisp commands suggested below to your `.evirc` file.

The **C-x** prefix commands integrate most easily with Evi since it would only override `kill-line` in input modes. To access **C-x** prefix commands in all modes, use:

```
(evl-define-key evl-all-keymaps "\C-x" 'evl-emacs-command)
```

The Meta (or **ESC** prefix) commands are no problem if you just want to access them from the top-level, where in vi **ESC** only serves the purpose of ringing the bell, which in vi is a handy way of verifying that you're in command mode, but is not vital considering the emacs mode line. Use the following to access Meta (or **ESC** prefix) commands only from the top-level:

```
(evl-define-key '(top-level) "\e" 'evl-emacs-command)
```

Accessing Meta (or **ESC** prefix) commands from other modes is problematic because **ESC** is the command to exit that mode. However, if you wish, you can redefine the meta prefix to something else - I recommend **C-a**. Thus to use Meta commands in all modes use:

```
(setq evl-meta-prefix-char ?\C-a)
(evl-define-key evl-all-keymaps "\C-a" (cons (current-global-map) ?\e))
```

The **C-c** prefix commands are also problematic because **C-c** is vi's interrupt character. At the top-level, this is again not much of a problem, and you could use:

```
(evl-define-key '(top-level) "\C-c" 'evl-emacs-command)
```

**C-c** will still function as an interrupt character at all other places (including in the middle of a command, where it's most useful). In the various input modes you could forego **C-c** as your interrupt character and access the **C-c** prefix commands by using the following (presumably in addition to the above):

```
(evl-define-key '(input replace ex) "\C-c" 'evl-emacs-command)
```

Note that setting `evl-insert-mode-local-bindings` (described in the next section) will have the same effect because all **C-c** commands are local bindings.

## 0.5 Taking advantage of emacs editing modes

A number of emacs editing modes have handy local key bindings other than Meta, *C-x* and *C-c* prefix bindings. For example, in C mode, RET does intelligent indenting, and } is bound to a command which automatically extends. By default, these aren't accessible of course, but you can have Evi enable these local bindings in insert mode by setting:

```
(setq evi-insert-mode-local-bindings t)
```

As current policy, however, Evi will not allow local mode bindings to override TAB, BS, or DEL, as well as, for obvious reasons, ESC. ESC prefix commands, however, can be accessed as described in the previous section.

## 0.6 Customizing Evi

Like vi, Evi will source your `.exrc` or `~/.exrc` file, and/or your EXINIT environment variable. If your startup runs into problems, let me know - you shouldn't have to change your vi initialization stuff to make Evi happy.

If you wish to use some Evi extensions in your startup, but still need to use vi, place these in `.exrc.evi`, `~/.exrc.evi` or EVIINIT so that vi won't gag on them.

Emacs lisp startup code for evi, such as that suggested in the previous sections, can be placed in either `.evirc` or `~/.evirc`.

And you can, of course, hack away at the the Evi source code if you want something not easily addressed by the above methods. If you feel what you've done would be generally useful, please email it to me, or post it.

One particular customization, not covered elsewhere, is how Evi handles the current directory. By default it behaves like vi - you have one global current directory, which you change using `:cd` (also see `:pushd`, and friends described below). Alternately, you may like the emacs behaviour better, which is that each buffer has its own idea of the current directory, and by default that directory is the directory that the file for that buffer resides in. In this mode, you can also change the current directory using `:cd`, but that will only affect the current buffer. To get this behaviour, place the following in your `.evirc`:

```
(setq evi-global-directory nil)
```

Another customization you might like to make is to alter the behaviour of ZZ. By default it is bound to `:Wq!`, which quietly writes all modified files and exits. If, however, you would like to be asked about each modified buffer before it is saved in order to avoid accidentally saving a file you didn't want saved, map ZZ to `:Wq`:

```
map ZZ :Wq\n
```

## 0.7 Arrow keys

If you like your arrow keys, there's one incompatibility that's not easily fixed under version 18 of emacs. Vi recognizes arrow keys that send ESC prefixed sequences by using a very short timeout to determine the difference between an ESC typed by a person, and an ESC sequence sent by the terminal. A compromise, if you happen to have such a terminal is to `:set notimeout`, which makes ESC ESC behave like a single ESC and maps your arrow keys to `h`, `j`, `k`, and `l`. As a side effect in Evi, the normal emacs ESC prefix commands will be in effect (except for ESC ESC of course). This could be undesirable if you're in the habit of

hitting the escape key a lot to verify you're in command mode, and follow that immediately with a command. This should be fixable in version 19 of emacs.

## 0.8 Enhancements

### 0.8.1 Edit repeat command

`_` is a new version of the repeat command `.` that prompts you with the keystrokes to repeat, allowing you to edit them before executing. This is particularly useful for the abovementioned complex operators. If you don't wish to re-execute the command, just hit `C-c`.

### 0.8.2 Ex input escapes

If you put

```
(setq ex-input-escapes t)
```

in your `.evirc`, then `Ex (:)` commands will accept the following escapes: `\e` for `ESC`, `\n` for newline, `\r` for `RET`, and `\C-x` for control-`x` (for all valid control characters). `\` otherwise works like `C-v`. Thus:

```
map \ \C-f
```

would make the space character be page forward, and

```
map g \|
```

would make `g` be goto-column. Note that `|` is normally a command separator and thus must be escaped.

### 0.8.3 Extended undo

The command `[u` continues the previous undo, by undoing one more change previous to the last change undone. Thus, a long enough sequence of `[us` will take you back to the unmodified state. If you went back too far, a `u` will reverse this process, and subsequent `[us` will move forward through the changes. Note that `vim` does this using repeat command `(.)`; however, that conflicts with the meaning of `u.` in `vi`, which is: 'undo, then do again'. This is quite handy for reapplying a change that you initially did in the wrong place, so `evi` leaves that meaning alone, and defines a new command for 'extended undo'. Personally, I use 'extended undo' more than 'undo line', so I swap the two definitions in my `.evirc`:

```
(evi-define-key '(vi) "[u" 'evi-undo-line)
(evi-define-key '(vi) "U" 'evi-undo-more)
```

### 0.8.4 Ex command completion

In `Ex (:)` commands, you can use the `TAB` to perform completion on the following: commands, variable names (`:set`), filenames, buffernames, maps, or abbreviations. Which completion to perform is determined solely by where you are in the partial command you are typing. For example:

```
:set erTAB
```

would complete to:

```
:set errorbells
```

(leaving the cursor after the `s` in `errorbells`.)

### 0.8.5 Word definition

You can define exactly what Evi treats as words for the `w`, `b`, `e`, `W`, `B` and `E` commands. They are defined by setting either or both of the new options `word` (for `w`, `b` and `e`) or `Word` (for `W`, `B` and `E`) to a regular expression describing what words look like. For example, here's a definition of words that only considers alphanumeric words:

```
set word=[a-zA-Z0-9]+
```

Contrast this with the default definition:

```
[a-zA-Z0-9_]+\|[\^a-zA-Z0-9_\ \t\n]+\|[\^[\ \t]]*\n
```

See the emacs documentation on regular expressions for details.

### 0.8.6 Sentence and larger motions

My interpretation of sentence, paragraph, and section motion differs somewhat from `vi`'s in that they behave more analogously to how word motion behaves - e.g. a forward paragraph takes you to the beginning of the next paragraph - not the blank line after the previous paragraph. However, when doing a delete using one of these motions, unless you are at the beginning of the sentence, paragraph or section, the delete will only happen to the end of the sentence, paragraph or section, not to the beginning of the next. I find this *\*much\** more useful than the `vi` behaviour - if you disagree, please let me know.

### 0.8.7 Directory commands

`:pushd`, `:popd` and `:dirs` commands exist, similar to those found in `csh` and `bash`. Note these only make sense in conjunction with `evi-global-directory = t` (which is the default).

### 0.8.8 Background shell commands

The `:!` command now takes an optional `&` (as in `:!&`), which causes the shell command to be executed asynchronously, with the output going to the window `*Shell Command Output*`.

### 0.8.9 Unnamed register

The unnamed register (where deleted text goes) is preserved across excursions into insert mode, etc. This means you can delete something, insert something, then 'put' the deleted text. In `vi`, for no apparent reason, you can't do this, even though insert mode doesn't use the unnamed register.

The unnamed register is also preserved between buffers, so you can yank text in one buffer and put it into another.

### 0.8.10 New command counts

Several commands that didn't take counts in `vi` take counts in Evi. `p` and `P` take a prefix count and will put the text that many times, regardless of the size of the text - `vi` will apparently only do the prefix count for less than line sized text. `/` takes a prefix count to find the *n*th occurrence of a string. `D` takes a count (I could never figure out why it didn't, since `C` takes a count).

### 0.8.11 Rectangle edits and arbitrary regions

In `vi`, you do most of your editing based on regions defined by `vi`'s motion commands. For example, `dw` deletes the region starting at the current cursor location and extending up to



where the cursor would be if you'd typed `w`. Those edit commands that don't explicitly use a motion command are just shorthand for ones that do: e.g. `x` is shorthand for `dl`.

However, this doesn't give you a convenient way of operating on arbitrary rectangular regions, or regions that are inconvenient to describe by a single motion command. Evi allows you to operate on such regions by a special form of the `m` (mark) command, and several new commands that are only understood when they are operands to edit operators (such as `c`, `d`, `>`, etc).

The basic idea is that you will define a region by two points. For a rectangle region, for example, you will define two opposite corners. You define the first point by using `m..`. Then you move to the second point and execute the operator you want, specifying the motion to be one of: `r` for rectangle, `R` for rows, `C` for columns, and `a` for arbitrary (meaning everything between the marked point and where the cursor currently is).

```
m.3j5wda
```

would delete the text from where the cursor started to 3 lines down and 5 words over. `R` is often handy for operating on large arbitrary sections of text, for example say you needed to shift some text that ran on for several pages and you weren't sure just how long it was at the start:

```
m.C-fC-fjj>R
```

### 0.8.12 Shell window

`:shell` starts up an emacs shell in the current window (instead of suspending emacs, and starting a subshell). The shell to run comes from the vi variable `isshell`, and defaults to the value of the environment variable `SHELL`. `:gdb program` starts up gdb in the current window on the specified program. For both of these, you are automatically placed in insert mode, where you should be able to interact as you would expect to, except that `ESC` will take you into command mode. While in command mode, hitting return will send the current line as input to the shell/gdb, similar to command-line editing vi-style in bash, but will leave you in command mode.

### 0.8.13 Extended marks

The marks used by the mark command `m` are emacs markers, thus they mark a position in a buffer, not necessarily the current one. This affects the goto mark commands ``` and `'`. For example, if mark `a` is placed in one buffer, then later in another buffer, the command ``a` is typed, evi will first switch to that buffer, then go to the location in that buffer. `'` and ``` also accept `.` and `,` for pop context, and unpop context respectively. Thus, `'.` will take you to the previous context (defined as in vi by a region of relative motion, with an 'absolute' motion pushing a new context. quotes surround 'absolute' because a search is considered an absolute motion for this purpose), and `'.` will take you to the context before that. There is a ring of 10 contexts so after 10 `'.` commands you'll end up at the original previous context. 'Unpop context' means move forward thru the ring. `''` and ```` are defined as exchange current location with the location of the previous context. The context ring is buffer local, so use of it will always keep you in the same buffer.

### 0.8.14 Register enhancements

Two changes involving registers. First, `"'`, and `""`, are new commands which allow you to insert literal text directly into a register. `''` inserts a single character, and `""` inserts a

string. E.g. `""helloESC` inserts the string ‘hello’ into the unnamed register, and `"a"/` inserts a slash into register a. The choice of command names are intuitive (they suggest quotes around a literal char/string), albeit unfortunate because `"` is supposed to be the prefix for a register specification, not a command. Second, the register specification `"^` specifies appending to the unnamed register (the one that gets used when no register is specified). E.g., `""ickESC` appends ‘ick’ to the unnamed register.

### 0.8.15 Language/mode specific editing

`%` exhibits language sensitivity in that it ignores parentheses embedded in quotes. What defines quotes is based on what minor mode emacs is in (such as c-mode or lisp-mode), or you can roll your own (see emacs command `modify-syntax-entry`).

`=` is no longer specific to `:set lisp`. It indents according to the mode. See emacs command `indent-according-to-mode`.

## 0.9 New operators

The `*` operator can be used to send text to emacs processes. `*` prompts for the name of a process buffer, and the region specified is sent to that process. Subsequent invocations of `*` will use the same process buffer as last specified as a default. E.g., to send the current line of text as a command to the emacs shell (see `:shell`), type `***shell*RET`, or if the shell is already the default, just `**RET`. Paragraph motion or parenthesis match is often perfect for sending function definitions to an interpreter, e.g. place the cursor at the beginning of the function, and type `*}RET` or `*/RET`. If the function def is less easily described you can use `m.` and `yR` described above. In specifying the process buffer, you can use buffer completion using space or tab.

I’m experimenting with some new complex operators. I’m particularly interested in your thoughts on these:

`[{` operates over lines in a region. It takes a motion, and a sequence of operations to perform on each line in the region defined by the motion. The sequence of operations is prompted for on the bottom line. Double the `{` to operate on whole lines. The point starts in the first column for each line operated on. For example:

```
[{i> C-vESC
```

would prefix every line in the rest of the current paragraph with `>`. The `C-v ESC` sequence inserts an `ESC` into the string you are entering so that it will terminate input when the loop body is executed, not as you are entering the command. For example:

```
10[{{i/* C-vESCA */C-vESC
```

would place C-style comments around the next 10 lines.

`[( defines a parameterized macro body. A parameterized macro is different from standard macro text in that it is parameterized by prefix count and register specification. In the body of such a macro, there are two special commands: # and &. # is replaced by the prefix count applied to this macro, and & is replaced by the register specification applied to this macro. For example:`

```
"a8[(j#w&dwRET
```

would go down one line, move over 8 words, then delete the next word into register ‘a’. This is rather contrived, but it gives you the idea. Parameterized macro bodies are

obviously not very useful typed out each time, and are intended to be the body of a map macro. For example:

```
:map M [(j#w&dw\eRET
"a8M
```

would be a much more likely scenario for the use of such a macro.

## 0.10 Differences

The following vi commands behave differently in Evi or not implemented:

<b>C-@</b>	(insert mode only) Not implemented.
<b>C-q</b>	Not implemented.
<b>C-[</b>	Doesn't always do the right thing, and it doesn't do the timeout trick necessary to allow it to recognize keypad sequences.
<b>#</b>	Not implemented.
<b>Q</b>	Quits evi mode, returning you to emacs, similiar to the way <i>Q</i> in vi quits visual mode, returning you to ex.

Digit registers don't work entirely correctly - there are circumstances in which separate lines of a change/deletion are supposed to go into separate registers

`:set lisp` has no effect, however, emacs does largely take care of any lisp'ish behaviour you'd want automatically if the file you're editing is suffixed with `.l` or `.el`. One particular loss, however, is that `)` and `(` don't work on s-expressions like they would in vi with lisp set.

## 0.11 Supported ex commands and variable settings

The following ex commands are supported in Evi:

```
abbrev, cd, chdir, copy, delete, edit, file, global, map, move, next,
preserve, print, put, quit, read, recover, set, shell, source, substitute,
tag, unabbrev, unmap, write, wq, yank, !, <, >, &
```

The following ex variable settings are supported in Evi:

```
autoindent, errorbells, ignorecase, magic, notimeout, shell,
shiftwidth, showmatch, tabstop, wrapmargin, wrapscan
```

## 0.12 Note to vip users

Undo does not continue via `..`. This is incompatible with vi - the sequence `u.` in vi means 'undo, then do again', whereas in vip it means 'undo, then undo some more.' For the vip functionality use `evi-undo-more`, described in the section on enhancements.

The vip commands for editing a file (`v` and `V`) and switching buffers (`s`, and `S`) are not supported. Use `:e`, `:E`, `:b`, and `:B` instead. See previous section on file and buffer management, or try these cute macros which are (mostly) functional replacements, including doing file and buffer completion (note the space on the end of the line):

```
:map v :edit |map V :Edit |map K :kill |map s :buffer |map S :Buffer
```

`:q` exits emacs. I believe the default behaviour in `vip` is to simply kill the current buffer (a concept `vi` doesn't really have) - either that or quit `vi` emulation. Any of these choices is reasonable, however, given `:k` for killing buffers (a new command for a new concept), and `Q` for exiting `vi` emulation, I chose to have `:q` do exactly what it does in `vi`.

### 0.13 What to do if things go wrong

There are still some glitches in `evi`, of course, and if you encounter one, please email me what went wrong, in as much detail as you can. My address is `jlewis@cse.ogi.edu`. In the meantime, if `Evi` doesn't seem to be responding, or you're having difficulty getting out of some mode, first try `C-c` (this is `vi`'s interrupt character), then if you're still in a pickle, try `C-g` (this is emacs' interrupt character - there are some situations that you might get into where it works instead of `C-c`).

### 0.14 Mailing list

There is a mailing list for discussion of `evi` - usage, bugs, new features, etc. To join the list, send a note to `evi-list-request@brandx.rain.com` including your e-mail address in the body of the message. To submit to the list, send mail to `evi-list@brandx.rain.com`.

### 0.15 Getting a copy of Evi

You can obtain a copy of `Evi` via FTP from an elisp archive site, or from the mail-server on my machine (which will assure the most up-to-date version). The main elisp-archive is at `archive.cis.ohio-state.edu` (in the directory `gnu/emacs/elisp-archive`) - `Evi` is in `modes/evi.el.Z`. To get it from my mail-server, send a message to `mail-server@brandx.rain.com` containing:

```
send evi.el
- or -
send evi.el.Z
followed by:
end
```

`evi.el.Z` will be uuencoded for delivery. Also, add 'send index' if you want the file sizes.

### 0.16 Credits

Masahiko Sato - for having the audacity to write `vip` in the first place ;-)  
 Eric Bensen - for being the first real advocate and those helpful folks who gave feedback on the first release  
 and the following immensely helpful souls: James Montebello, Steven Dick, Mike Sangrey, Brian Divine, Bill Reynolds, Roger E. Benz, Richard Ryan, Volker Englisch, John Haugen, Calvin Clark, Sam Falkner, Mark Pulver and Cameron Gregory