GNU Emacs Lisp Reference Manual

Volume 2
For Emacs Version 24.1
Revision 3.1, May 2012

**by Bil Lewis, Dan LaLiberte, Richard Stallman,
the GNU Manual Group, et al.**

This is edition 3.1 of the GNU Emacs Lisp Reference Manual,
corresponding to Emacs version 24.1.

Published by the Free Software Foundation
51 Franklin St, Fifth Floor
Boston, MA 02110-1301
USA
ISBN 1-882114-74-4

Cover art by Etienne Suvasa.

Short Contents

Volume 1

Volume 2

Table of Contents

Volume 1

Volume 2

27 Buffers

A *buffer* is a Lisp object containing text to be edited. Buffers are used to hold the contents of files that are being visited; there may also be buffers that are not visiting files. While several buffers may exist at one time, only one buffer is designated the *current buffer* at any time. Most editing commands act on the contents of the current buffer. Each buffer, including the current buffer, may or may not be displayed in any windows.

27.1 Buffer Basics

Buffers in Emacs editing are objects that have distinct names and hold text that can be edited. Buffers appear to Lisp programs as a special data type. You can think of the contents of a buffer as a string that you can extend; insertions and deletions may occur in any part of the buffer. See Chapter 32 [Text], page 122.

A Lisp buffer object contains numerous pieces of information. Some of this information is directly accessible to the programmer through variables, while other information is accessible only through special-purpose functions. For example, the visited file name is directly accessible through a variable, while the value of point is accessible only through a primitive function.

Buffer-specific information that is directly accessible is stored in *buffer-local* variable bindings, which are variable values that are effective only in a particular buffer. This feature allows each buffer to override the values of certain variables. Most major modes override variables such as `fill-column` or `comment-column` in this way. For more information about buffer-local variables and functions related to them, see Section 11.10 [Buffer-Local Variables], page 150, vol. 1.

For functions and variables related to visiting files in buffers, see Section 25.1 [Visiting Files], page 461, vol. 1 and Section 25.2 [Saving Buffers], page 465, vol. 1. For functions and variables related to the display of buffers in windows, see Section 28.9 [Buffers and Windows], page 36.

`bufferp` *object* [Function]

This function returns `t` if *object* is a buffer, `nil` otherwise.

27.2 The Current Buffer

There are, in general, many buffers in an Emacs session. At any time, one of them is designated the *current buffer*—the buffer in which most editing takes place. Most of the primitives for examining or changing text operate implicitly on the current buffer (see Chapter 32 [Text], page 122).

Normally, the buffer displayed in the selected window is the current buffer, but this is not always so: a Lisp program can temporarily designate any buffer as current in order to operate on its contents, without changing what is displayed on the screen. The most basic function for designating a current buffer is `set-buffer`.

`current-buffer` [Function]

This function returns the current buffer.

```
(current-buffer)
     ⇒ #<buffer buffers.texi>
```

`set-buffer` *buffer-or-name* [Function]

This function makes *buffer-or-name* the current buffer. *buffer-or-name* must be an existing buffer or the name of an existing buffer. The return value is the buffer made current.

This function does not display the buffer in any window, so the user cannot necessarily see the buffer. But Lisp programs will now operate on it.

When an editing command returns to the editor command loop, Emacs automatically calls `set-buffer` on the buffer shown in the selected window. This is to prevent confusion: it ensures that the buffer that the cursor is in, when Emacs reads a command, is the buffer to which that command applies (see Chapter 21 [Command Loop], page 315, vol. 1). Thus, you should not use `set-buffer` to switch visibly to a different buffer; for that, use the functions described in Section 28.10 [Switching Buffers], page 37.

When writing a Lisp function, do *not* rely on this behavior of the command loop to restore the current buffer after an operation. Editing commands can also be called as Lisp functions by other programs, not just from the command loop; it is convenient for the caller if the subroutine does not change which buffer is current (unless, of course, that is the subroutine's purpose).

To operate temporarily on another buffer, put the `set-buffer` within a `save-current-buffer` form. Here, as an example, is a simplified version of the command `append-to-buffer`:

```
(defun append-to-buffer (buffer start end)
  "Append the text of the region to BUFFER."
  (interactive "BAppend to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (save-current-buffer
      (set-buffer (get-buffer-create buffer))
      (insert-buffer-substring oldbuf start end))))
```

Here, we bind a local variable to record the current buffer, and then `save-current-buffer` arranges to make it current again later. Next, `set-buffer` makes the specified buffer current, and `insert-buffer-substring` copies the string from the original buffer to the specified (and now current) buffer.

Alternatively, we can use the `with-current-buffer` macro:

```
(defun append-to-buffer (buffer start end)
  "Append the text of the region to BUFFER."
  (interactive "BAppend to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (with-current-buffer (get-buffer-create buffer)
      (insert-buffer-substring oldbuf start end))))
```

In either case, if the buffer appended to happens to be displayed in some window, the next redisplay will show how its text has changed. If it is not displayed in any window, you will not see the change immediately on the screen. The command causes the buffer to become current temporarily, but does not cause it to be displayed.

If you make local bindings (with `let` or function arguments) for a variable that may also have buffer-local bindings, make sure that the same buffer is current at the beginning

and at the end of the local binding's scope. Otherwise you might bind it in one buffer and unbind it in another!

Do not rely on using `set-buffer` to change the current buffer back, because that won't do the job if a quit happens while the wrong buffer is current. For instance, in the previous example, it would have been wrong to do this:

```
(let ((oldbuf (current-buffer)))
  (set-buffer (get-buffer-create buffer))
  (insert-buffer-substring oldbuf start end)
  (set-buffer oldbuf))
```

Using `save-current-buffer` or `with-current-buffer`, as we did, correctly handles quitting, errors, and `throw`, as well as ordinary evaluation.

`save-current-buffer` *body...* [Special Form]

The `save-current-buffer` special form saves the identity of the current buffer, evaluates the *body* forms, and finally restores that buffer as current. The return value is the value of the last form in *body*. The current buffer is restored even in case of an abnormal exit via `throw` or error (see Section 10.5 [Nonlocal Exits], page 126, vol. 1).

If the buffer that used to be current has been killed by the time of exit from `save-current-buffer`, then it is not made current again, of course. Instead, whichever buffer was current just before exit remains current.

`with-current-buffer` *buffer-or-name body...* [Macro]

The `with-current-buffer` macro saves the identity of the current buffer, makes *buffer-or-name* current, evaluates the *body* forms, and finally restores the current buffer. *buffer-or-name* must specify an existing buffer or the name of an existing buffer.

The return value is the value of the last form in *body*. The current buffer is restored even in case of an abnormal exit via `throw` or error (see Section 10.5 [Nonlocal Exits], page 126, vol. 1).

`with-temp-buffer` *body...* [Macro]

The `with-temp-buffer` macro evaluates the *body* forms with a temporary buffer as the current buffer. It saves the identity of the current buffer, creates a temporary buffer and makes it current, evaluates the *body* forms, and finally restores the previous current buffer while killing the temporary buffer. By default, undo information (see Section 32.9 [Undo], page 137) is not recorded in the buffer created by this macro (but *body* can enable that, if needed).

The return value is the value of the last form in *body*. You can return the contents of the temporary buffer by using `(buffer-string)` as the last form.

The current buffer is restored even in case of an abnormal exit via `throw` or error (see Section 10.5 [Nonlocal Exits], page 126, vol. 1).

See also `with-temp-file` in [Writing to Files], page 469, vol. 1.

27.3 Buffer Names

Each buffer has a unique name, which is a string. Many of the functions that work on buffers accept either a buffer or a buffer name as an argument. Any argument called *buffer-or-name* is of this sort, and an error is signaled if it is neither a string nor a buffer. Any argument called *buffer* must be an actual buffer object, not a name.

Buffers that are ephemeral and generally uninteresting to the user have names starting with a space, so that the `list-buffers` and `buffer-menu` commands don't mention them (but if such a buffer visits a file, it **is** mentioned). A name starting with space also initially disables recording undo information; see Section 32.9 [Undo], page 137.

`buffer-name` **&optional** *buffer* [Function]

This function returns the name of *buffer* as a string. *buffer* defaults to the current buffer.

If `buffer-name` returns `nil`, it means that *buffer* has been killed. See Section 27.10 [Killing Buffers], page 13.

```
(buffer-name)
     ⇒ "buffers.texi"

(setq foo (get-buffer "temp"))
     ⇒ #<buffer temp>
(kill-buffer foo)
     ⇒ nil
(buffer-name foo)
     ⇒ nil
foo
     ⇒ #<killed buffer>
```

`rename-buffer` *newname* **&optional** *unique* [Command]

This function renames the current buffer to *newname*. An error is signaled if *newname* is not a string.

Ordinarily, `rename-buffer` signals an error if *newname* is already in use. However, if *unique* is non-`nil`, it modifies *newname* to make a name that is not in use. Interactively, you can make *unique* non-`nil` with a numeric prefix argument. (This is how the command `rename-uniquely` is implemented.)

This function returns the name actually given to the buffer.

`get-buffer` *buffer-or-name* [Function]

This function returns the buffer specified by *buffer-or-name*. If *buffer-or-name* is a string and there is no buffer with that name, the value is `nil`. If *buffer-or-name* is a buffer, it is returned as given; that is not very useful, so the argument is usually a name. For example:

```
(setq b (get-buffer "lewis"))
     ⇒ #<buffer lewis>
(get-buffer b)
     ⇒ #<buffer lewis>
(get-buffer "Frazzle-nots")
     ⇒ nil
```

See also the function `get-buffer-create` in Section 27.9 [Creating Buffers], page 13.

`generate-new-buffer-name` *starting-name* **&optional** *ignore* [Function]
This function returns a name that would be unique for a new buffer—but does not create the buffer. It starts with *starting-name*, and produces a name not currently in use for any buffer by appending a number inside of '`<...>`'. It starts at 2 and keeps incrementing the number until it is not the name of an existing buffer.

If the optional second argument *ignore* is non-`nil`, it should be a string, a potential buffer name. It means to consider that potential buffer acceptable, if it is tried, even it is the name of an existing buffer (which would normally be rejected). Thus, if buffers named '`foo`', '`foo<2>`', '`foo<3>`' and '`foo<4>`' exist,

```
(generate-new-buffer-name "foo")
     ⇒ "foo<5>"
(generate-new-buffer-name "foo" "foo<3>")
     ⇒ "foo<3>"
(generate-new-buffer-name "foo" "foo<6>")
     ⇒ "foo<5>"
```

See the related function `generate-new-buffer` in Section 27.9 [Creating Buffers], page 13.

27.4 Buffer File Name

The *buffer file name* is the name of the file that is visited in that buffer. When a buffer is not visiting a file, its buffer file name is `nil`. Most of the time, the buffer name is the same as the nondirectory part of the buffer file name, but the buffer file name and the buffer name are distinct and can be set independently. See Section 25.1 [Visiting Files], page 461, vol. 1.

`buffer-file-name` **&optional** *buffer* [Function]
This function returns the absolute file name of the file that *buffer* is visiting. If *buffer* is not visiting any file, `buffer-file-name` returns `nil`. If *buffer* is not supplied, it defaults to the current buffer.

```
(buffer-file-name (other-buffer))
     ⇒ "/usr/user/lewis/manual/files.texi"
```

`buffer-file-name` [Variable]
This buffer-local variable contains the name of the file being visited in the current buffer, or `nil` if it is not visiting a file. It is a permanent local variable, unaffected by `kill-all-local-variables`.

```
buffer-file-name
     ⇒ "/usr/user/lewis/manual/buffers.texi"
```

It is risky to change this variable's value without doing various other things. Normally it is better to use `set-visited-file-name` (see below); some of the things done there, such as changing the buffer name, are not strictly necessary, but others are essential to avoid confusing Emacs.

`buffer-file-truename` [Variable]
This buffer-local variable holds the abbreviated truename of the file visited in the current buffer, or `nil` if no file is visited. It is a permanent local, unaffected by `kill-all-local-variables`. See Section 25.6.3 [Truenames], page 474, vol. 1, and [abbreviate-file-name], page 485, vol. 1.

`buffer-file-number` [Variable]
This buffer-local variable holds the file number and directory device number of the file visited in the current buffer, or `nil` if no file or a nonexistent file is visited. It is a permanent local, unaffected by `kill-all-local-variables`.

The value is normally a list of the form `(`*`filenum devnum`*`)`. This pair of numbers uniquely identifies the file among all files accessible on the system. See the function `file-attributes`, in Section 25.6.4 [File Attributes], page 475, vol. 1, for more information about them.

If `buffer-file-name` is the name of a symbolic link, then both numbers refer to the recursive target.

`get-file-buffer` *filename* [Function]
This function returns the buffer visiting file *filename*. If there is no such buffer, it returns `nil`. The argument *filename*, which must be a string, is expanded (see Section 25.8.4 [File Name Expansion], page 486, vol. 1), then compared against the visited file names of all live buffers. Note that the buffer's `buffer-file-name` must match the expansion of *filename* exactly. This function will not recognize other names for the same file.

```
(get-file-buffer "buffers.texi")
    ⇒ #<buffer buffers.texi>
```

In unusual circumstances, there can be more than one buffer visiting the same file name. In such cases, this function returns the first such buffer in the buffer list.

`find-buffer-visiting` *filename* **&optional** *predicate* [Function]
This is like `get-file-buffer`, except that it can return any buffer visiting the file *possibly under a different name*. That is, the buffer's `buffer-file-name` does not need to match the expansion of *filename* exactly, it only needs to refer to the same file. If *predicate* is non-`nil`, it should be a function of one argument, a buffer visiting *filename*. The buffer is only considered a suitable return value if *predicate* returns non-`nil`. If it can not find a suitable buffer to return, `find-buffer-visiting` returns `nil`.

`set-visited-file-name` *filename* **&optional** *no-query along-with-file* [Command]
If *filename* is a non-empty string, this function changes the name of the file visited in the current buffer to *filename*. (If the buffer had no visited file, this gives it one.) The *next time* the buffer is saved it will go in the newly-specified file.

This command marks the buffer as modified, since it does not (as far as Emacs knows) match the contents of *filename*, even if it matched the former visited file. It also renames the buffer to correspond to the new file name, unless the new name is already in use.

If *filename* is `nil` or the empty string, that stands for "no visited file". In this case, `set-visited-file-name` marks the buffer as having no visited file, without changing the buffer's modified flag.

Normally, this function asks the user for confirmation if there already is a buffer visiting *filename*. If *no-query* is non-`nil`, that prevents asking this question. If there already is a buffer visiting *filename*, and the user confirms or *query* is non-`nil`, this function makes the new buffer name unique by appending a number inside of '`<...>`' to *filename*.

If *along-with-file* is non-`nil`, that means to assume that the former visited file has been renamed to *filename*. In this case, the command does not change the buffer's modified flag, nor the buffer's recorded last file modification time as reported by `visited-file-modtime` (see Section 27.6 [Modification Time], page 8). If *along-with-file* is `nil`, this function clears the recorded last file modification time, after which `visited-file-modtime` returns zero.

When the function `set-visited-file-name` is called interactively, it prompts for *filename* in the minibuffer.

`list-buffers-directory` [Variable]

This buffer-local variable specifies a string to display in a buffer listing where the visited file name would go, for buffers that don't have a visited file name. Dired buffers use this variable.

27.5 Buffer Modification

Emacs keeps a flag called the *modified flag* for each buffer, to record whether you have changed the text of the buffer. This flag is set to `t` whenever you alter the contents of the buffer, and cleared to `nil` when you save it. Thus, the flag shows whether there are unsaved changes. The flag value is normally shown in the mode line (see Section 23.4.4 [Mode Line Variables], page 422, vol. 1), and controls saving (see Section 25.2 [Saving Buffers], page 465, vol. 1) and auto-saving (see Section 26.2 [Auto-Saving], page 507, vol. 1).

Some Lisp programs set the flag explicitly. For example, the function `set-visited-file-name` sets the flag to `t`, because the text does not match the newly-visited file, even if it is unchanged from the file formerly visited.

The functions that modify the contents of buffers are described in Chapter 32 [Text], page 122.

`buffer-modified-p` **&optional** *buffer* [Function]

This function returns `t` if the buffer *buffer* has been modified since it was last read in from a file or saved, or `nil` otherwise. If *buffer* is not supplied, the current buffer is tested.

`set-buffer-modified-p` *flag* [Function]

This function marks the current buffer as modified if *flag* is non-`nil`, or as unmodified if the flag is `nil`.

Another effect of calling this function is to cause unconditional redisplay of the mode line for the current buffer. In fact, the function `force-mode-line-update` works by doing this:

```
(set-buffer-modified-p (buffer-modified-p))
```

`restore-buffer-modified-p` *flag* [Function]
Like `set-buffer-modified-p`, but does not force redisplay of mode lines.

`not-modified` **&optional** *arg* [Command]
This command marks the current buffer as unmodified, and not needing to be saved. If *arg* is non-`nil`, it marks the buffer as modified, so that it will be saved at the next suitable occasion. Interactively, *arg* is the prefix argument.

Don't use this function in programs, since it prints a message in the echo area; use `set-buffer-modified-p` (above) instead.

`buffer-modified-tick` **&optional** *buffer* [Function]
This function returns *buffer*'s modification-count. This is a counter that increments every time the buffer is modified. If *buffer* is `nil` (or omitted), the current buffer is used. The counter can wrap around occasionally.

`buffer-chars-modified-tick` **&optional** *buffer* [Function]
This function returns *buffer*'s character-change modification-count. Changes to text properties leave this counter unchanged; however, each time text is inserted or removed from the buffer, the counter is reset to the value that would be returned by `buffer-modified-tick`. By comparing the values returned by two `buffer-chars-modified-tick` calls, you can tell whether a character change occurred in that buffer in between the calls. If *buffer* is `nil` (or omitted), the current buffer is used.

27.6 Buffer Modification Time

Suppose that you visit a file and make changes in its buffer, and meanwhile the file itself is changed on disk. At this point, saving the buffer would overwrite the changes in the file. Occasionally this may be what you want, but usually it would lose valuable information. Emacs therefore checks the file's modification time using the functions described below before saving the file. (See Section 25.6.4 [File Attributes], page 475, vol. 1, for how to examine a file's modification time.)

`verify-visited-file-modtime` **&optional** *buffer* [Function]
This function compares what *buffer* (by default, the current-buffer) has recorded for the modification time of its visited file against the actual modification time of the file as recorded by the operating system. The two should be the same unless some other process has written the file since Emacs visited or saved it.

The function returns `t` if the last actual modification time and Emacs's recorded modification time are the same, `nil` otherwise. It also returns `t` if the buffer has no recorded last modification time, that is if `visited-file-modtime` would return zero.

It always returns `t` for buffers that are not visiting a file, even if `visited-file-modtime` returns a non-zero value. For instance, it always returns `t` for dired buffers. It returns `t` for buffers that are visiting a file that does not exist and never existed, but `nil` for file-visiting buffers whose file has been deleted.

`clear-visited-file-modtime` [Function]
: This function clears out the record of the last modification time of the file being visited by the current buffer. As a result, the next attempt to save this buffer will not complain of a discrepancy in file modification times.

  This function is called in `set-visited-file-name` and other exceptional places where the usual test to avoid overwriting a changed file should not be done.

`visited-file-modtime` [Function]
: This function returns the current buffer's recorded last file modification time, as a list of the form `(high low)`. (This is the same format that `file-attributes` uses to return time values; see Section 25.6.4 [File Attributes], page 475, vol. 1.)

  If the buffer has no recorded last modification time, this function returns zero. This case occurs, for instance, if the buffer is not visiting a file or if the time has been explicitly cleared by `clear-visited-file-modtime`. Note, however, that `visited-file-modtime` returns a list for some non-file buffers too. For instance, in a Dired buffer listing a directory, it returns the last modification time of that directory, as recorded by Dired.

  For a new buffer visiting a not yet existing file, *high* is -1 and *low* is 65535, that is, $2^{16} - 1$.

`set-visited-file-modtime` **&optional** *time* [Function]
: This function updates the buffer's record of the last modification time of the visited file, to the value specified by *time* if *time* is not `nil`, and otherwise to the last modification time of the visited file.

  If *time* is neither `nil` nor zero, it should have the form `(high . low)` or `(high low)`, in either case containing two integers, each of which holds 16 bits of the time.

  This function is useful if the buffer was not read from the file normally, or if the file itself has been changed for some known benign reason.

`ask-user-about-supersession-threat` *filename* [Function]
: This function is used to ask a user how to proceed after an attempt to modify an buffer visiting file *filename* when the file is newer than the buffer text. Emacs detects this because the modification time of the file on disk is newer than the last save-time of the buffer. This means some other program has probably altered the file.

  Depending on the user's answer, the function may return normally, in which case the modification of the buffer proceeds, or it may signal a `file-supersession` error with data `(filename)`, in which case the proposed buffer modification is not allowed.

  This function is called automatically by Emacs on the proper occasions. It exists so you can customize Emacs by redefining it. See the file '`userlock.el`' for the standard definition.

  See also the file locking mechanism in Section 25.5 [File Locks], page 470, vol. 1.

27.7 Read-Only Buffers

If a buffer is *read-only*, then you cannot change its contents, although you may change your view of the contents by scrolling and narrowing.

Read-only buffers are used in two kinds of situations:

- A buffer visiting a write-protected file is normally read-only.

  Here, the purpose is to inform the user that editing the buffer with the aim of saving it in the file may be futile or undesirable. The user who wants to change the buffer text despite this can do so after clearing the read-only flag with `C-x C-q`.
- Modes such as Dired and Rmail make buffers read-only when altering the contents with the usual editing commands would probably be a mistake.

  The special commands of these modes bind `buffer-read-only` to `nil` (with `let`) or bind `inhibit-read-only` to `t` around the places where they themselves change the text.

`buffer-read-only` [Variable]
: This buffer-local variable specifies whether the buffer is read-only. The buffer is read-only if this variable is non-`nil`.

`inhibit-read-only` [Variable]
: If this variable is non-`nil`, then read-only buffers and, depending on the actual value, some or all read-only characters may be modified. Read-only characters in a buffer are those that have non-`nil` `read-only` properties (either text properties or overlay properties). See Section 32.19.4 [Special Properties], page 162, for more information about text properties. See Section 38.9 [Overlays], page 315, for more information about overlays and their properties.

  If `inhibit-read-only` is `t`, all `read-only` character properties have no effect. If `inhibit-read-only` is a list, then `read-only` character properties have no effect if they are members of the list (comparison is done with `eq`).

`toggle-read-only` **&optional** *arg* [Command]
: This command toggles whether the current buffer is read-only. It is intended for interactive use; do not use it in programs (it may have side-effects, such as enabling View mode, and does not affect read-only text properties). To change the read-only state of a buffer in a program, explicitly set `buffer-read-only` to the proper value. To temporarily ignore a read-only state, bind `inhibit-read-only`.

  If *arg* is non-`nil`, it should be a raw prefix argument. `toggle-read-only` sets `buffer-read-only` to `t` if the numeric value of that prefix argument is positive and to `nil` otherwise. See Section 21.12 [Prefix Command Arguments], page 353, vol. 1.

`barf-if-buffer-read-only` [Function]
: This function signals a `buffer-read-only` error if the current buffer is read-only. See Section 21.2.1 [Using Interactive], page 316, vol. 1, for another way to signal an error if the current buffer is read-only.

27.8 The Buffer List

The *buffer list* is a list of all live buffers. The order of the buffers in this list is based primarily on how recently each buffer has been displayed in a window. Several functions, notably `other-buffer`, use this ordering. A buffer list displayed for the user also follows this order.

Creating a buffer adds it to the end of the buffer list, and killing a buffer removes it from that list. A buffer moves to the front of this list whenever it is chosen for display

in a window (see Section 28.10 [Switching Buffers], page 37) or a window displaying it is selected (see Section 28.7 [Selecting Windows], page 33). A buffer moves to the end of the list when it is buried (see `bury-buffer`, below). There are no functions available to the Lisp programmer which directly manipulate the buffer list.

In addition to the fundamental buffer list just described, Emacs maintains a local buffer list for each frame, in which the buffers that have been displayed (or had their windows selected) in that frame come first. (This order is recorded in the frame's `buffer-list` frame parameter; see Section 29.3.3.5 [Buffer Parameters], page 75.) Buffers never displayed in that frame come afterward, ordered according to the fundamental buffer list.

`buffer-list` **&optional** *frame* [Function]

This function returns the buffer list, including all buffers, even those whose names begin with a space. The elements are actual buffers, not their names.

If *frame* is a frame, this returns *frame*'s local buffer list. If *frame* is `nil` or omitted, the fundamental buffer list is used: the buffers appear in order of most recent display or selection, regardless of which frames they were displayed on.

```
(buffer-list)
     ⇒ (#<buffer buffers.texi>
         #<buffer  *Minibuf-1*> #<buffer buffer.c>
         #<buffer *Help*> #<buffer TAGS>)

;; Note that the name of the minibuffer
;;    begins with a space!
(mapcar (function buffer-name) (buffer-list))
    ⇒ ("buffers.texi" " *Minibuf-1*"
        "buffer.c" "*Help*" "TAGS")
```

The list returned by `buffer-list` is constructed specifically; it is not an internal Emacs data structure, and modifying it has no effect on the order of buffers. If you want to change the order of buffers in the fundamental buffer list, here is an easy way:

```
(defun reorder-buffer-list (new-list)
  (while new-list
    (bury-buffer (car new-list))
    (setq new-list (cdr new-list))))
```

With this method, you can specify any order for the list, but there is no danger of losing a buffer or adding something that is not a valid live buffer.

To change the order or value of a specific frame's buffer list, set that frame's `buffer-list` parameter with `modify-frame-parameters` (see Section 29.3.1 [Parameter Access], page 70).

`other-buffer` **&optional** *buffer visible-ok frame* [Function]

This function returns the first buffer in the buffer list other than *buffer*. Usually, this is the buffer appearing in the most recently selected window (in frame *frame* or else the selected frame, see Section 29.9 [Input Focus], page 83), aside from *buffer*. Buffers whose names start with a space are not considered at all.

If *buffer* is not supplied (or if it is not a live buffer), then `other-buffer` returns the first buffer in the selected frame's local buffer list. (If *frame* is non-`nil`, it returns the first buffer in *frame*'s local buffer list instead.)

If *frame* has a non-`nil` `buffer-predicate` parameter, then `other-buffer` uses that predicate to decide which buffers to consider. It calls the predicate once for each buffer, and if the value is `nil`, that buffer is ignored. See Section 29.3.3.5 [Buffer Parameters], page 75.

If *visible-ok* is `nil`, `other-buffer` avoids returning a buffer visible in any window on any visible frame, except as a last resort. If *visible-ok* is non-`nil`, then it does not matter whether a buffer is displayed somewhere or not.

If no suitable buffer exists, the buffer '`*scratch*`' is returned (and created, if necessary).

`last-buffer` **&optional** *buffer visible-ok frame* [Function]

This function returns the last buffer in *frame*'s buffer list other than *BUFFER*. If *frame* is omitted or `nil`, it uses the selected frame's buffer list.

The argument *visible-ok* is handled as with `other-buffer`, see above. If no suitable buffer can be found, the buffer '`*scratch*`' is returned.

`bury-buffer` **&optional** *buffer-or-name* [Command]

This command puts *buffer-or-name* at the end of the buffer list, without changing the order of any of the other buffers on the list. This buffer therefore becomes the least desirable candidate for `other-buffer` to return. The argument can be either a buffer itself or the name of one.

This functions operates on each frame's `buffer-list` parameter as well as the fundamental buffer list; therefore, the buffer that you bury will come last in the value of `(buffer-list` *`frame`*`)` and in the value of `(buffer-list)`. In addition, it also puts the buffer at the end of the list of buffer of the selected window (see Section 28.14 [Window History], page 45) provided it is shown in that window.

If *buffer-or-name* is `nil` or omitted, this means to bury the current buffer. In addition, if the current buffer is displayed in the selected window, this makes sure that the window is either deleted or another buffer is shown in it. More precisely, if the window is dedicated (see Section 28.15 [Dedicated Windows], page 46) and there are other windows on its frame, the window is deleted. If the window is both dedicated and the only window on its frame's terminal, the function specified by `frame-auto-hide-function` (see Section 28.16 [Quitting Windows], page 47) will deal with the window. If the window is not dedicated to its buffer, it calls `switch-to-prev-buffer` (see Section 28.14 [Window History], page 45) to show another buffer in that window. If *buffer-or-name* is displayed in some other window, it remains displayed there.

To replace a buffer in all the windows that display it, use `replace-buffer-in-windows`, See Section 28.9 [Buffers and Windows], page 36.

`unbury-buffer` [Command]

This command switches to the last buffer in the local buffer list of the selected frame. More precisely, it calls the function `switch-to-buffer` (see Section 28.10 [Switching Buffers], page 37), to display the buffer returned by `last-buffer` (see above), in the selected window.

27.9 Creating Buffers

This section describes the two primitives for creating buffers. `get-buffer-create` creates a buffer if it finds no existing buffer with the specified name; `generate-new-buffer` always creates a new buffer and gives it a unique name.

Other functions you can use to create buffers include `with-output-to-temp-buffer` (see Section 38.8 [Temporary Displays], page 313) and `create-file-buffer` (see Section 25.1 [Visiting Files], page 461, vol. 1). Starting a subprocess can also create a buffer (see Chapter 37 [Processes], page 257).

`get-buffer-create` *buffer-or-name* [Function]

This function returns a buffer named *buffer-or-name*. The buffer returned does not become the current buffer—this function does not change which buffer is current.

buffer-or-name must be either a string or an existing buffer. If it is a string and a live buffer with that name already exists, `get-buffer-create` returns that buffer. If no such buffer exists, it creates a new buffer. If *buffer-or-name* is a buffer instead of a string, it is returned as given, even if it is dead.

```
(get-buffer-create "foo")
     ⇒ #<buffer foo>
```

The major mode for a newly created buffer is set to Fundamental mode. (The default value of the variable `major-mode` is handled at a higher level; see Section 23.2.2 [Auto Major Mode], page 403, vol. 1.) If the name begins with a space, the buffer initially disables undo information recording (see Section 32.9 [Undo], page 137).

`generate-new-buffer` *name* [Function]

This function returns a newly created, empty buffer, but does not make it current. The name of the buffer is generated by passing *name* to the function `generate-new-buffer-name` (see Section 27.3 [Buffer Names], page 4). Thus, if there is no buffer named *name*, then that is the name of the new buffer; if that name is in use, a suffix of the form '`<n>`', where *n* is an integer, is appended to *name*.

An error is signaled if *name* is not a string.

```
(generate-new-buffer "bar")
     ⇒ #<buffer bar>
(generate-new-buffer "bar")
     ⇒ #<buffer bar<2>>
(generate-new-buffer "bar")
     ⇒ #<buffer bar<3>>
```

The major mode for the new buffer is set to Fundamental mode. The default value of the variable `major-mode` is handled at a higher level. See Section 23.2.2 [Auto Major Mode], page 403, vol. 1.

27.10 Killing Buffers

Killing a buffer makes its name unknown to Emacs and makes the memory space it occupied available for other use.

The buffer object for the buffer that has been killed remains in existence as long as anything refers to it, but it is specially marked so that you cannot make it current or

display it. Killed buffers retain their identity, however; if you kill two distinct buffers, they remain distinct according to `eq` although both are dead.

If you kill a buffer that is current or displayed in a window, Emacs automatically selects or displays some other buffer instead. This means that killing a buffer can change the current buffer. Therefore, when you kill a buffer, you should also take the precautions associated with changing the current buffer (unless you happen to know that the buffer being killed isn't current). See Section 27.2 [Current Buffer], page 1.

If you kill a buffer that is the base buffer of one or more indirect buffers, the indirect buffers are automatically killed as well.

The `buffer-name` of a buffer is `nil` if, and only if, the buffer is killed. A buffer that has not been killed is called a *live* buffer. To test whether a buffer is live or killed, use the function `buffer-live-p` (see below).

`kill-buffer` **&optional** *buffer-or-name* [Command]

This function kills the buffer *buffer-or-name*, freeing all its memory for other uses or to be returned to the operating system. If *buffer-or-name* is `nil` or omitted, it kills the current buffer.

Any processes that have this buffer as the `process-buffer` are sent the `SIGHUP` ("hangup") signal, which normally causes them to terminate. See Section 37.8 [Signals to Processes], page 270.

If the buffer is visiting a file and contains unsaved changes, `kill-buffer` asks the user to confirm before the buffer is killed. It does this even if not called interactively. To prevent the request for confirmation, clear the modified flag before calling `kill-buffer`. See Section 27.5 [Buffer Modification], page 7.

This function calls `replace-buffer-in-windows` for cleaning up all windows currently displaying the buffer to be killed.

Killing a buffer that is already dead has no effect.

This function returns `t` if it actually killed the buffer. It returns `nil` if the user refuses to confirm or if *buffer-or-name* was already dead.

```
(kill-buffer "foo.unchanged")
     ⇒ t
(kill-buffer "foo.changed")

---------- Buffer: Minibuffer ----------
Buffer foo.changed modified; kill anyway? (yes or no) yes
---------- Buffer: Minibuffer ----------

     ⇒ t
```

`kill-buffer-query-functions` [Variable]

After confirming unsaved changes, `kill-buffer` calls the functions in the list `kill-buffer-query-functions`, in order of appearance, with no arguments. The buffer being killed is the current buffer when they are called. The idea of this feature is that these functions will ask for confirmation from the user. If any of them returns `nil`, `kill-buffer` spares the buffer's life.

`kill-buffer-hook` [Variable]

This is a normal hook run by `kill-buffer` after asking all the questions it is going to ask, just before actually killing the buffer. The buffer to be killed is current when

the hook functions run. See Section 23.1 [Hooks], page 396, vol. 1. This variable is a permanent local, so its local binding is not cleared by changing major modes.

`buffer-offer-save` [User Option]
This variable, if non-`nil` in a particular buffer, tells `save-buffers-kill-emacs` and `save-some-buffers` (if the second optional argument to that function is `t`) to offer to save that buffer, just as they offer to save file-visiting buffers. See [Definition of save-some-buffers], page 465, vol. 1. The variable `buffer-offer-save` automatically becomes buffer-local when set for any reason. See Section 11.10 [Buffer-Local Variables], page 150, vol. 1.

`buffer-save-without-query` [Variable]
This variable, if non-`nil` in a particular buffer, tells `save-buffers-kill-emacs` and `save-some-buffers` to save this buffer (if it's modified) without asking the user. The variable automatically becomes buffer-local when set for any reason.

`buffer-live-p` *object* [Function]
This function returns `t` if *object* is a live buffer (a buffer which has not been killed), `nil` otherwise.

27.11 Indirect Buffers

An *indirect buffer* shares the text of some other buffer, which is called the *base buffer* of the indirect buffer. In some ways it is the analogue, for buffers, of a symbolic link among files. The base buffer may not itself be an indirect buffer.

The text of the indirect buffer is always identical to the text of its base buffer; changes made by editing either one are visible immediately in the other. This includes the text properties as well as the characters themselves.

In all other respects, the indirect buffer and its base buffer are completely separate. They have different names, independent values of point, independent narrowing, independent markers and overlays (though inserting or deleting text in either buffer relocates the markers and overlays for both), independent major modes, and independent buffer-local variable bindings.

An indirect buffer cannot visit a file, but its base buffer can. If you try to save the indirect buffer, that actually saves the base buffer.

Killing an indirect buffer has no effect on its base buffer. Killing the base buffer effectively kills the indirect buffer in that it cannot ever again be the current buffer.

`make-indirect-buffer` *base-buffer name* **&optional** *clone* [Command]
This creates and returns an indirect buffer named *name* whose base buffer is *base-buffer*. The argument *base-buffer* may be a live buffer or the name (a string) of an existing buffer. If *name* is the name of an existing buffer, an error is signaled.

If *clone* is non-`nil`, then the indirect buffer originally shares the "state" of *base-buffer* such as major mode, minor modes, buffer local variables and so on. If *clone* is omitted or `nil` the indirect buffer's state is set to the default state for new buffers.

If *base-buffer* is an indirect buffer, its base buffer is used as the base for the new buffer. If, in addition, *clone* is non-`nil`, the initial state is copied from the actual base buffer, not from *base-buffer*.

`clone-indirect-buffer` *newname display-flag* **&optional** *norecord* [Command]
This function creates and returns a new indirect buffer that shares the current buffer's base buffer and copies the rest of the current buffer's attributes. (If the current buffer is not indirect, it is used as the base buffer.)

If *display-flag* is non-`nil`, that means to display the new buffer by calling `pop-to-buffer`. If *norecord* is non-`nil`, that means not to put the new buffer to the front of the buffer list.

`buffer-base-buffer` **&optional** *buffer* [Function]
This function returns the base buffer of *buffer*, which defaults to the current buffer. If *buffer* is not indirect, the value is `nil`. Otherwise, the value is another buffer, which is never an indirect buffer.

27.12 Swapping Text Between Two Buffers

Specialized modes sometimes need to let the user access from the same buffer several vastly different types of text. For example, you may need to display a summary of the buffer text, in addition to letting the user access the text itself.

This could be implemented with multiple buffers (kept in sync when the user edits the text), or with narrowing (see Section 30.4 [Narrowing], page 109). But these alternatives might sometimes become tedious or prohibitively expensive, especially if each type of text requires expensive buffer-global operations in order to provide correct display and editing commands.

Emacs provides another facility for such modes: you can quickly swap buffer text between two buffers with `buffer-swap-text`. This function is very fast because it doesn't move any text, it only changes the internal data structures of the buffer object to point to a different chunk of text. Using it, you can pretend that a group of two or more buffers are actually a single virtual buffer that holds the contents of all the individual buffers together.

`buffer-swap-text` *buffer* [Function]
This function swaps the text of the current buffer and that of its argument *buffer*. It signals an error if one of the two buffers is an indirect buffer (see Section 27.11 [Indirect Buffers], page 15) or is a base buffer of an indirect buffer.

All the buffer properties that are related to the buffer text are swapped as well: the positions of point and mark, all the markers, the overlays, the text properties, the undo list, the value of the `enable-multibyte-characters` flag (see Section 33.1 [Text Representations], page 182), etc.

If you use `buffer-swap-text` on a file-visiting buffer, you should set up a hook to save the buffer's original text rather than what it was swapped with. `write-region-annotate-functions` works for this purpose. You should probably set `buffer-saved-size` to -2 in the buffer, so that changes in the text it is swapped with will not interfere with auto-saving.

27.13 The Buffer Gap

Emacs buffers are implemented using an invisible *gap* to make insertion and deletion faster. Insertion works by filling in part of the gap, and deletion adds to the gap. Of course, this means that the gap must first be moved to the locus of the insertion or deletion. Emacs

moves the gap only when you try to insert or delete. This is why your first editing command in one part of a large buffer, after previously editing in another far-away part, sometimes involves a noticeable delay.

This mechanism works invisibly, and Lisp code should never be affected by the gap's current location, but these functions are available for getting information about the gap status.

`gap-position` [Function]
: This function returns the current gap position in the current buffer.

`gap-size` [Function]
: This function returns the current gap size of the current buffer.

28 Windows

This chapter describes the functions and variables related to Emacs windows. See Chapter 29 [Frames], page 66, for how windows are assigned an area of screen available for Emacs to use. See Chapter 38 [Display], page 299, for information on how text is displayed in windows.

28.1 Basic Concepts of Emacs Windows

A *window* is a area of the screen that is used to display a buffer (see Chapter 27 [Buffers], page 1). In Emacs Lisp, windows are represented by a special Lisp object type.

Windows are grouped into frames (see Chapter 29 [Frames], page 66). Each frame contains at least one window; the user can subdivide it into multiple, non-overlapping windows to view several buffers at once. Lisp programs can use multiple windows for a variety of purposes. In Rmail, for example, you can view a summary of message titles in one window, and the contents of the selected message in another window.

Emacs uses the word "window" with a different meaning than in graphical desktop environments and window systems, such as the X Window System. When Emacs is run on X, each of its graphical X windows is an Emacs frame (containing one or more Emacs windows). When Emacs is run on a text terminal, the frame fills the entire terminal screen.

Unlike X windows, Emacs windows are *tiled*; they never overlap within the area of the frame. When a window is created, resized, or deleted, the change in window space is taken from or given to the adjacent windows, so that the total area of the frame is unchanged.

A *live window* is one that is actually displaying a buffer in a frame. Such a window can be *deleted*, i.e. removed from the frame (see Section 28.6 [Deleting Windows], page 31); then it is no longer live, but the Lisp object representing it might be still referenced from other Lisp objects. A deleted window may be brought back to life by restoring a saved window configuration (see Section 28.23 [Window Configurations], page 60).

`windowp` *object* [Function]
: This function returns `t` if *object* is a window (whether or not it is live). Otherwise, it returns `nil`.

`window-live-p` *object* [Function]
: This function returns `t` if *object* is a live window and `nil` otherwise. A live window is one that displays a buffer.

The windows in each frame are organized into a *window tree*. See Section 28.2 [Windows and Frames], page 19. The leaf nodes of each window tree are live windows—the ones actually displaying buffers. The internal nodes of the window tree are internal windows, which are not live. You can distinguish internal windows from deleted windows with `window-valid-p`.

`window-valid-p` *object* [Function]
: This function returns `t` if *object* is a live window, or an internal window in a window tree. Otherwise, it returns `nil`, including for the case where *object* is a deleted window.

In each frame, at any time, exactly one Emacs window is designated as *selected within the frame*. For the selected frame, that window is called the *selected window*—the one in which most editing takes place, and in which the cursor for selected windows appears (see Section 29.3.3.7 [Cursor Parameters], page 76). The selected window's buffer is usually also the current buffer, except when `set-buffer` has been used (see Section 27.2 [Current Buffer], page 1). As for non-selected frames, the window selected within the frame becomes the selected window if the frame is ever selected. See Section 28.7 [Selecting Windows], page 33.

`selected-window` [Function]
: This function returns the selected window (which is always a live window).

28.2 Windows and Frames

Each window belongs to exactly one frame (see Chapter 29 [Frames], page 66).

`window-frame` *window* [Function]
: This function returns the frame that the window *window* belongs to. If *window* is `nil`, it defaults to the selected window.

`window-list` **&optional** *frame minibuffer window* [Function]
: This function returns a list of live windows belonging to the frame *frame*. If *frame* is omitted or `nil`, it defaults to the selected frame.

  The optional argument *minibuffer* specifies whether to include the minibuffer window in the returned list. If *minibuffer* is `t`, the minibuffer window is included. If *minibuffer* is `nil` or omitted, the minibuffer window is included only if it is active. If *minibuffer* is neither `nil` nor `t`, the minibuffer window is never included.

  The optional argument *window*, if non-`nil`, should be a live window on the specified frame; then *window* will be the first element in the returned list. If *window* is omitted or `nil`, the window selected within the frame is the first element.

Windows in the same frame are organized into a *window tree*, whose leaf nodes are the live windows. The internal nodes of a window tree are not live; they exist for the purpose of organizing the relationships between live windows. The root node of a window tree is called the *root window*. It can be either a live window (if the frame has just one window), or an internal window.

A minibuffer window (see Section 20.11 [Minibuffer Windows], page 311, vol. 1) is not part of its frame's window tree unless the frame is a minibuffer-only frame. Nonetheless, most of the functions in this section accept the minibuffer window as an argument. Also, the function `window-tree` described at the end of this section lists the minibuffer window alongside the actual window tree.

`frame-root-window` **&optional** *frame-or-window* [Function]
: This function returns the root window for *frame-or-window*. The argument *frame-or-window* should be either a window or a frame; if omitted or `nil`, it defaults to the selected frame. If *frame-or-window* is a window, the return value is the root window of that window's frame.

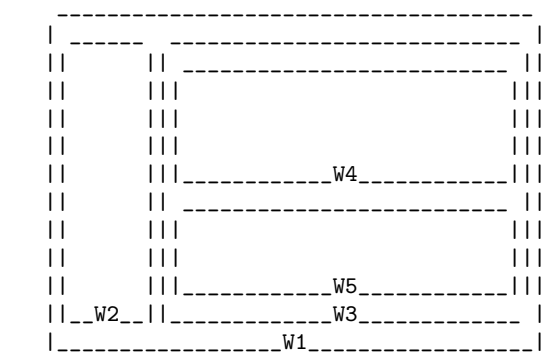
When a window is split, there are two live windows where previously there was one. One of these is represented by the same Lisp window object as the original window, and the other is represented by a newly-created Lisp window object. Both of these live windows become leaf nodes of the window tree, as *child windows* of a single internal window. If necessary, Emacs automatically creates this internal window, which is also called the *parent window*, and assigns it to the appropriate position in the window tree. A set of windows that share the same parent are called *siblings*.

`window-parent` **&optional** *window* [Function]
: This function returns the parent window of *window*. If *window* is omitted or `nil`, it defaults to the selected window. The return value is `nil` if *window* has no parent (i.e. it is a minibuffer window or the root window of its frame).

Each internal window always has at least two child windows. If this number falls to one as a result of window deletion, Emacs automatically deletes the internal window, and its sole remaining child window takes its place in the window tree.

Each child window can be either a live window, or an internal window (which in turn would have its own child windows). Therefore, each internal window can be thought of as occupying a certain rectangular *screen area*—the union of the areas occupied by the live windows that are ultimately descended from it.

For each internal window, the screen areas of the immediate children are arranged either vertically or horizontally (never both). If the child windows are arranged one above the other, they are said to form a *vertical combination*; if they are arranged side by side, they are said to form a *horizontal combination*. Consider the following example:



The root window of this frame is an internal window, `W1`. Its child windows form a horizontal combination, consisting of the live window `W2` and the internal window `W3`. The child windows of `W3` form a vertical combination, consisting of the live windows `W4` and `W5`. Hence, the live windows in this window tree are `W2` `W4`, and `W5`.

The following functions can be used to retrieve a child window of an internal window, and the siblings of a child window.

`window-top-child` *window* [Function]
: This function returns the topmost child window of *window*, if *window* is an internal window whose children form a vertical combination. For any other type of window, the return value is `nil`.

`window-left-child` *window* [Function]
: This function returns the leftmost child window of *window*, if *window* is an internal window whose children form a horizontal combination. For any other type of window, the return value is `nil`.

`window-child` *window* [Function]
: This function returns the first child window of the internal window *window*—the topmost child window for a vertical combination, or the leftmost child window for a horizontal combination. If *window* is a live window, the return value is `nil`.

`window-combined-p` **&optional** *window horizontal* [Function]
: This function returns a non-`nil` value if and only if *window* is part of a vertical combination. If *window* is omitted or `nil`, it defaults to the selected one.

  If the optional argument *horizontal* is non-`nil`, this means to return non-`nil` if and only if *window* is part of a horizontal combination.

`window-next-sibling` **&optional** *window* [Function]
: This function returns the next sibling of the window *window*. If omitted or `nil`, *window* defaults to the selected window. The return value is `nil` if *window* is the last child of its parent.

`window-prev-sibling` **&optional** *window* [Function]
: This function returns the previous sibling of the window *window*. If omitted or `nil`, *window* defaults to the selected window. The return value is `nil` if *window* is the first child of its parent.

The functions `window-next-sibling` and `window-prev-sibling` should not be confused with the functions `next-window` and `previous-window`, which return the next and previous window, respectively, in the cyclic ordering of windows (see Section 28.8 [Cyclic Window Ordering], page 34).

You can use the following functions to find the first live window on a frame, and to retrieve the entire window tree of a frame:

`frame-first-window` **&optional** *frame-or-window* [Function]
: This function returns the live window at the upper left corner of the frame specified by *frame-or-window*. The argument *frame-or-window* must denote a window or a live frame and defaults to the selected frame. If *frame-or-window* specifies a window, this function returns the first window on that window's frame. Under the assumption that the frame from our canonical example is selected `(frame-first-window)` returns `W2`.

`window-tree` **&optional** *frame* [Function]
: This function returns a list representing the window tree for frame *frame*. If *frame* is omitted or `nil`, it defaults to the selected frame.

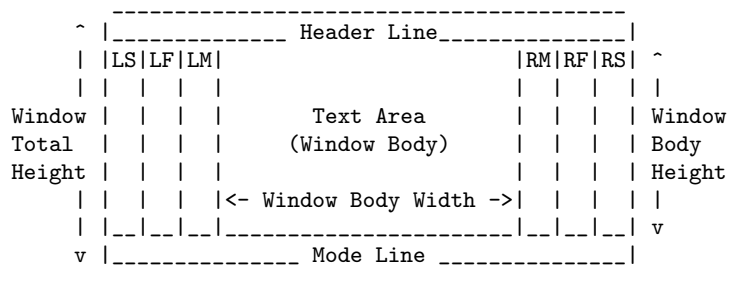
  The return value is a list of the form `(root mini)`, where *root* represents the window tree of the frame's root window, and *mini* is the frame's minibuffer window.

  If the root window is live, *root* is that window itself. Otherwise, *root* is a list `(dir edges w1 w2 ...)` where *dir* is `nil` for a horizontal combination and `t` for a vertical combination, *edges* gives the size and position of the combination, and the remaining

elements are the child windows. Each child window may again be a window object (for a live window) or a list with the same format as above (for an internal window). The *edges* element is a list `(left top right bottom)`, similar to the value returned by `window-edges` (see Section 28.22 [Coordinates and Windows], page 58).

28.3 Window Sizes

The following schematic shows the structure of a live window:



At the center of the window is the *text area*, or *body*, where the buffer text is displayed. On each side of the text area is a series of vertical areas; from innermost to outermost, these are the left and right margins, denoted by LM and RM in the schematic (see Section 38.15.5 [Display Margins], page 354); the left and right fringes, denoted by LF and RF (see Section 38.13 [Fringes], page 344); and the left or right scroll bar, only one of which is present at any time, denoted by LS and RS (see Section 38.14 [Scroll Bars], page 349). At the top of the window is an optional header line (see Section 23.4.7 [Header Lines], page 426, vol. 1), and at the bottom of the window is the mode line (see Section 23.4 [Mode Line Format], page 419, vol. 1).

Emacs provides several functions for finding the height and width of a window. Except where noted, Emacs reports window heights and widths as integer numbers of lines and columns, respectively. On a graphical display, each "line" and "column" actually corresponds to the height and width of a "default" character specified by the frame's default font. Thus, if a window is displaying text with a different font or size, the reported height and width for that window may differ from the actual number of text lines or columns displayed within it.

The *total height* of a window is the distance between the top and bottom of the window, including the header line (if one exists) and the mode line. The *total width* of a window is the distance between the left and right edges of the mode line. Note that the height of a frame is not the same as the height of its windows, since a frame may also contain an echo area, menu bar, and tool bar (see Section 29.3.4 [Size and Position], page 79).

`window-total-height` **&optional** *window* [Function]

This function returns the total height, in lines, of the window *window*. If *window* is omitted or `nil`, it defaults to the selected window. If *window* is an internal window, the return value is the total height occupied by its descendant windows.

`window-total-width` **&optional** *window* [Function]
: This function returns the total width, in columns, of the window *window*. If *window* is omitted or `nil`, it defaults to the selected window. If *window* is internal, the return value is the total width occupied by its descendant windows.

`window-total-size` **&optional** *window horizontal* [Function]
: This function returns either the total height or width of the window *window*. If *horizontal* is omitted or `nil`, this is equivalent to calling `window-total-height` for *window*; otherwise it is equivalent to calling `window-total-width` for *window*.

The following functions can be used to determine whether a given window has any adjacent windows.

`window-full-height-p` **&optional** *window* [Function]
: This function returns non-`nil` if *window* has no other window above or below it in its frame, i.e. its total height equals the total height of the root window on that frame. If *window* is omitted or `nil`, it defaults to the selected window.

`window-full-width-p` **&optional** *window* [Function]
: This function returns non-`nil` if *window* has no other window to the left or right in its frame, i.e. its total width equals that of the root window on that frame. If *window* is omitted or `nil`, it defaults to the selected window.

The *body height* of a window is the height of its text area, which does not include the mode or header line. Similarly, the *body width* is the width of the text area, which does not include the scroll bar, fringes, or margins.

`window-body-height` **&optional** *window* [Function]
: This function returns the body height, in lines, of the window *window*. If *window* is omitted or `nil`, it defaults to the selected window; otherwise it must be a live window.

  If there is a partially-visible line at the bottom of the text area, that counts as a whole line; to exclude such a partially-visible line, use `window-text-height`, below.

`window-body-width` **&optional** *window* [Function]
: This function returns the body width, in columns, of the window *window*. If *window* is omitted or `nil`, it defaults to the selected window; otherwise it must be a live window.

`window-body-size` **&optional** *window horizontal* [Function]
: This function returns the body height or body width of *window*. If *horizontal* is omitted or `nil`, it is equivalent to calling `window-body-height` for *window*; otherwise it is equivalent to calling `window-body-width`.

`window-text-height` **&optional** *window* [Function]
: This function is like `window-body-height`, except that any partially-visible line at the bottom of the text area is not counted.

For compatibility with previous versions of Emacs, `window-height` is an alias for `window-total-height`, and `window-width` is an alias for `window-body-width`. These aliases are considered obsolete and will be removed in the future.

Commands that change the size of windows (see Section 28.4 [Resizing Windows], page 24), or split them (see Section 28.5 [Splitting Windows], page 26), obey the variables `window-min-height` and `window-min-width`, which specify the smallest allowable window height and width. See Section "Deleting and Rearranging Windows" in *The GNU Emacs Manual*. They also obey the variable `window-size-fixed`, with which a window can be *fixed* in size:

`window-size-fixed` [Variable]
: If this buffer-local variable is non-`nil`, the size of any window displaying the buffer cannot normally be changed. Deleting a window or changing the frame's size may still change its size, if there is no choice.

  If the value is `height`, then only the window's height is fixed; if the value is `width`, then only the window's width is fixed. Any other non-`nil` value fixes both the width and the height.

`window-size-fixed-p` **&optional** *window horizontal* [Function]
: This function returns a non-`nil` value if *window*'s height is fixed. If *window* is omitted or `nil`, it defaults to the selected window. If the optional argument *horizontal* is non-`nil`, the return value is non-`nil` if *window*'s width is fixed.

  A `nil` return value does not necessarily mean that *window* can be resized in the desired direction. To determine that, use the function `window-resizable`. See Section 28.4 [Resizing Windows], page 24.

See Section 28.22 [Coordinates and Windows], page 58, for more functions that report the positions of various parts of a window relative to the frame, from which you can calculate its size. In particular, you can use the functions `window-pixel-edges` and `window-inside-pixel-edges` to find the size in pixels, for graphical displays.

28.4 Resizing Windows

This section describes functions for resizing a window without changing the size of its frame. Because live windows do not overlap, these functions are meaningful only on frames that contain two or more windows: resizing a window also changes the size of a neighboring window. If there is just one window on a frame, its size cannot be changed except by resizing the frame (see Section 29.3.4 [Size and Position], page 79).

Except where noted, these functions also accept internal windows as arguments. Resizing an internal window causes its child windows to be resized to fit the same space.

`window-resizable` *window delta* **&optional** *horizontal ignore* [Function]
: This function returns *delta* if the size of *window* can be changed vertically by *delta* lines. If the optional argument *horizontal* is non-`nil`, it instead returns *delta* if *window* can be resized horizontally by *delta* columns. It does not actually change the window size.

  If *window* is `nil`, it defaults to the selected window.

  A positive value of *delta* means to check whether the window can be enlarged by that number of lines or columns; a negative value of *delta* means to check whether the window can be shrunk by that many lines or columns. If *delta* is non-zero, a return value of 0 means that the window cannot be resized.

Normally, the variables `window-min-height` and `window-min-width` specify the smallest allowable window size. See Section "Deleting and Rearranging Windows" in *The GNU Emacs Manual*. However, if the optional argument *ignore* is non-`nil`, this function ignores `window-min-height` and `window-min-width`, as well as `window-size-fixed`. Instead, it considers the minimum-height window to be one consisting of a header (if any), a mode line, plus a text area one line tall; and a minimum-width window as one consisting of fringes, margins, and scroll bar (if any), plus a text area two columns wide.

`window-resize` *window delta* **&optional** *horizontal ignore* [Function]

This function resizes *window* by *delta* increments. If *horizontal* is `nil`, it changes the height by *delta* lines; otherwise, it changes the width by *delta* columns. A positive *delta* means to enlarge the window, and a negative *delta* means to shrink it.

If *window* is `nil`, it defaults to the selected window. If the window cannot be resized as demanded, an error is signaled.

The optional argument *ignore* has the same meaning as for the function `window-resizable` above.

The choice of which window edges this function alters depends on the values of the option `window-combination-resize` and the combination limits of the involved windows; in some cases, it may alter both edges. See Section 28.5 [Splitting Windows], page 26. To resize by moving only the bottom or right edge of a window, use the function `adjust-window-trailing-edge`, below.

`adjust-window-trailing-edge` *window delta* **&optional** *horizontal* [Function]

This function moves *window*'s bottom edge by *delta* lines. If optional argument *horizontal* is non-`nil`, it instead moves the right edge by *delta* columns. If *window* is `nil`, it defaults to the selected window.

A positive *delta* moves the edge downwards or to the right; a negative *delta* moves it upwards or to the left. If the edge cannot be moved as far as specified by *delta*, this function moves it as far as possible but does not signal a error.

This function tries to resize windows adjacent to the edge that is moved. If this is not possible for some reason (e.g. if that adjacent window is fixed-size), it may resize other windows.

The following commands resize windows in more specific ways. When called interactively, they act on the selected window.

`fit-window-to-buffer` **&optional** *window max-height min-height override* [Command]

This command adjusts the height of *window* to fit the text in it. It returns non-`nil` if it was able to resize *window*, and `nil` otherwise. If *window* is omitted or `nil`, it defaults to the selected window. Otherwise, it should be a live window.

The optional argument *max-height*, if non-`nil`, specifies the maximum total height that this function can give *window*. The optional argument *min-height*, if non-`nil`, specifies the minimum total height that it can give, which overrides the variable `window-min-height`.

If the optional argument *override* is non-`nil`, this function ignores any size restrictions imposed by `window-min-height` and `window-min-width`.

`shrink-window-if-larger-than-buffer` **&optional** *window* [Command]
This command attempts to reduce *window*'s height as much as possible while still showing its full buffer, but no less than `window-min-height` lines. The return value is non-`nil` if the window was resized, and `nil` otherwise. If *window* is omitted or `nil`, it defaults to the selected window. Otherwise, it should be a live window.

This command does nothing if the window is already too short to display all of its buffer, or if any of the buffer is scrolled off-screen, or if the window is the only live window in its frame.

`balance-windows` **&optional** *window-or-frame* [Command]
This function balances windows in a way that gives more space to full-width and/or full-height windows. If *window-or-frame* specifies a frame, it balances all windows on that frame. If *window-or-frame* specifies a window, it balances only that window and its siblings (see Section 28.2 [Windows and Frames], page 19).

`balance-windows-area` [Command]
This function attempts to give all windows on the selected frame approximately the same share of the screen area. Full-width or full-height windows are not given more space than other windows.

`maximize-window` **&optional** *window* [Command]
This function attempts to make *window* as large as possible, in both dimensions, without resizing its frame or deleting other windows. If *window* is omitted or `nil`, it defaults to the selected window.

`minimize-window` **&optional** *window* [Command]
This function attempts to make *window* as small as possible, in both dimensions, without deleting it or resizing its frame. If *window* is omitted or `nil`, it defaults to the selected window.

28.5 Splitting Windows

This section describes functions for creating a new window by *splitting* an existing one.

`split-window` **&optional** *window size side* [Command]
This function creates a new live window next to the window *window*. If *window* is omitted or `nil`, it defaults to the selected window. That window is "split", and reduced in size. The space is taken up by the new window, which is returned.

The optional second argument *size* determines the sizes of *window* and/or the new window. If it is omitted or `nil`, both windows are given equal sizes; if there is an odd line, it is allocated to the new window. If *size* is a positive number, *window* is given *size* lines (or columns, depending on the value of *side*). If *size* is a negative number, the new window is given $-$*size* lines (or columns).

If *size* is `nil`, this function obeys the variables `window-min-height` and `window-min-width`. See Section "Deleting and Rearranging Windows" in *The GNU Emacs Manual*. Thus, it signals an error if splitting would result in making a window smaller than those variables specify. However, a non-`nil` value for *size* causes those variables to be ignored; in that case, the smallest allowable window is considered to be one that has space for a text area one line tall and/or two columns wide.

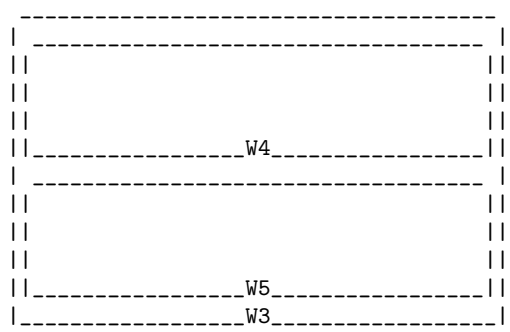
The optional third argument *side* determines the position of the new window relative to *window*. If it is `nil` or `below`, the new window is placed below *window*. If it is `above`, the new window is placed above *window*. In both these cases, *size* specifies a total window height, in lines.

If *side* is `t` or `right`, the new window is placed on the right of *window*. If *side* is `left`, the new window is placed on the left of *window*. In both these cases, *size* specifies a total window width, in columns.

If *window* is a live window, the new window inherits various properties from it, including margins and scroll bars. If *window* is an internal window, the new window inherits the properties of the window selected within *window*'s frame.

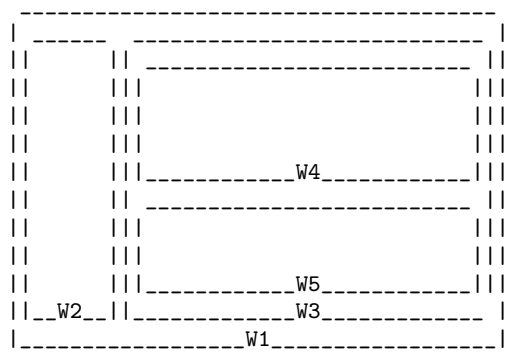
The behavior of this function may be altered by the window parameters of *window*, so long as the variable `ignore-window-parameters` is `nil`. If the value of the `split-window` window parameter is `t`, this function ignores all other window parameters. Otherwise, if the value of the `split-window` window parameter is a function, that function is called with the arguments *window*, *size*, and *side*, in lieu of the usual action of `split-window`. Otherwise, this function obeys the `window-atom` or `window-side` window parameter, if any. See Section 28.24 [Window Parameters], page 62.

As an example, here is a sequence of `split-window` calls that yields the window configuration discussed in Section 28.2 [Windows and Frames], page 19. This example demonstrates splitting a live window as well as splitting an internal window. We begin with a frame containing a single window (a live root window), which we denote by *W4*. Calling `(split-window W4)` yields this window configuration:



The `split-window` call has created a new live window, denoted by *W5*. It has also created a new internal window, denoted by *W3*, which becomes the root window and the parent of both *W4* and *W5*.

Next, we call `(split-window W3 nil 'left)`, passing the internal window *W3* as the argument. The result:



A new live window *W2* is created, to the left of the internal window *W3*. A new internal window *W1* is created, becoming the new root window.

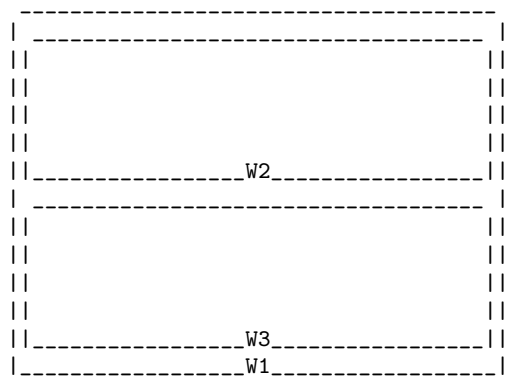
`window-combination-resize` [User Option]

If this variable is `nil`, `split-window` can only split a window (denoted by *window*) if *window*'s screen area is large enough to accommodate both itself and the new window.

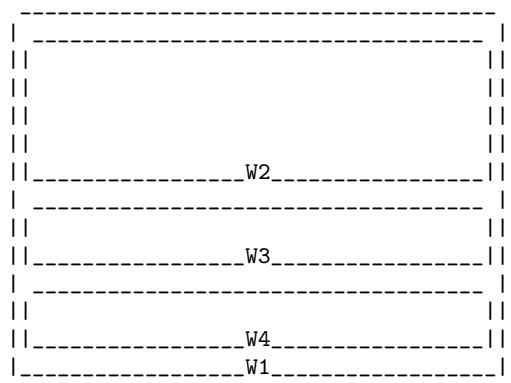
If this variable is `t`, `split-window` tries to resize all windows that are part of the same combination as *window*, in order to accommodate the new window. In particular, this may allow `split-window` to succeed even if *window* is a fixed-size window or too small to ordinarily split. Furthermore, subsequently resizing or deleting *window* may resize all other windows in its combination.

The default is `nil`. Other values are reserved for future use. The value of this variable is ignored when `window-combination-limit` is non-`nil` (see below).

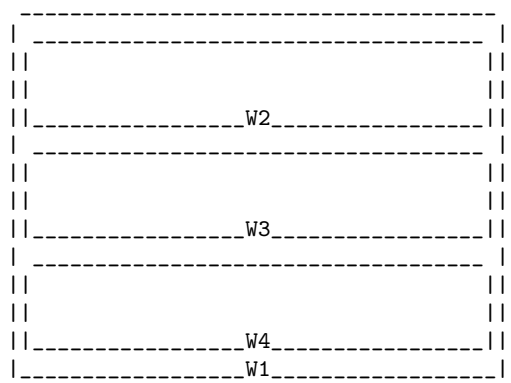
To illustrate the effect of `window-combination-resize`, consider the following window configuration:



If `window-combination-resize` is `nil`, splitting window `W3` leaves the size of `W2` unchanged:



If `window-combination-resize` is `t`, splitting `W3` instead leaves all three live windows with approximately the same height:



`window-combination-limit` [User Option]

If the value of this variable is `t`, the `split-window` function always creates a new internal window. If the value is `nil`, the new live window is allowed to share the existing parent window, if one exists, provided the split occurs in the same direction as the existing window combination (otherwise, a new internal window is created anyway). The default is `nil`. Other values are reserved for future use.

Thus, if the value of this variable is at all times `t`, then at all times every window tree is a binary tree (a tree where each window except the root window has exactly one sibling).

Furthermore, `split-window` calls `set-window-combination-limit` on the newly-created internal window, recording the current value of this variable. This affects how the window tree is rearranged when the child windows are deleted (see below).

`set-window-combination-limit` *window limit* [Function]

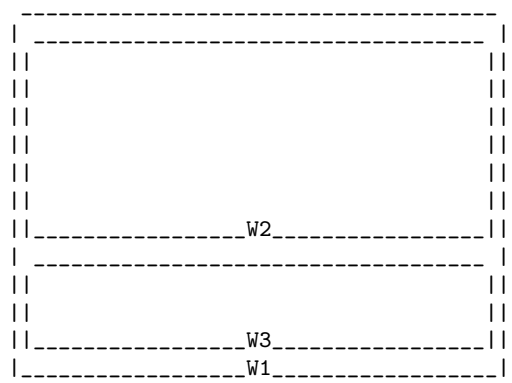
This functions sets the *combination limit* of the window *window* to *limit*. This value can be retrieved via the function `window-combination-limit`. See below for its effects; note that it is only meaningful for internal windows. The `split-window` function automatically calls this function, passing the value of the variable `window-combination-limit` as *limit*.

`window-combination-limit` *window* [Function]

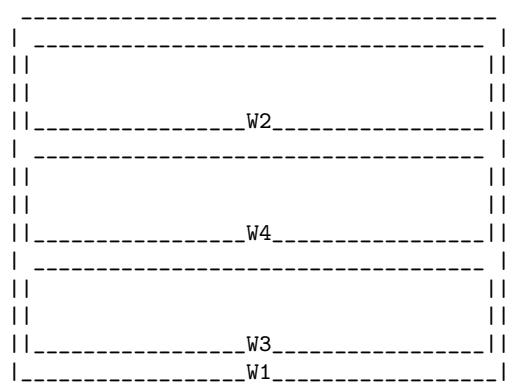
This function returns the combination limit for *window*.

The combination limit is meaningful only for an internal window. If it is `nil`, then Emacs is allowed to automatically delete *window*, in response to a window deletion, in order to group the child windows of *window* with its sibling windows to form a new window combination. If the combination limit is `t`, the child windows of *window* are never automatically re-combined with its siblings.

To illustrate the effect of `window-combination-limit`, consider the following configuration (throughout this example, we will assume that `window-combination-resize` is `nil`):

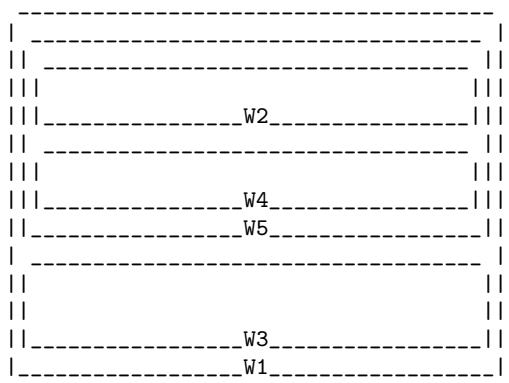


If `window-combination-limit` is `nil`, splitting `W2` into two windows, one above the other, yields



The newly-created window, `W4`, shares the same internal window `W1`. If `W4` is resized, it is allowed to resize the other live window, `W3`.

If `window-combination-limit` is `t`, splitting `W2` in the initial configuration would instead have produced this:



A new internal window `W5` has been created; its children are `W2` and the new live window `W4`. Now, `W2` is the only sibling of `W4`, so resizing `W4` will resize `W2`, leaving `W3` unaffected.

For interactive use, Emacs provides two commands which always split the selected window. These call `split-window` internally.

`split-window-right` **&optional** *size* [Command]
: This function splits the selected window into two side-by-side windows, putting the selected window on the left. If *size* is positive, the left window gets *size* columns; if *size* is negative, the right window gets $-$*size* columns.

`split-window-below` **&optional** *size* [Command]
: This function splits the selected window into two windows, one above the other, leaving the upper window selected. If *size* is positive, the upper window gets *size* lines; if *size* is negative, the lower window gets $-$*size* lines.

`split-window-keep-point` [User Option]
: If the value of this variable is non-`nil` (the default), `split-window-below` behaves as described above.

  If it is `nil`, `split-window-below` adjusts point in each of the two windows to minimize redisplay. (This is useful on slow terminals.) It selects whichever window contains the screen line that point was previously on. Note that this only affects `split-window-below`, not the lower-level `split-window` function.

28.6 Deleting Windows

Deleting a window removes it from the frame's window tree. If the window is a live window, it disappears from the screen. If the window is an internal window, its child windows are deleted too.

Even after a window is deleted, it continues to exist as a Lisp object, until there are no more references to it. Window deletion can be reversed, by restoring a saved window configuration (see Section 28.23 [Window Configurations], page 60).

`delete-window` **&optional** *window* [Command]
: This function removes *window* from display and returns `nil`. If *window* is omitted or `nil`, it defaults to the selected window. If deleting the window would leave no more

windows in the window tree (e.g. if it is the only live window in the frame), an error is signaled.

By default, the space taken up by *window* is given to one of its adjacent sibling windows, if any. However, if the variable `window-combination-resize` is non-`nil`, the space is proportionally distributed among any remaining windows in the window combination. See Section 28.5 [Splitting Windows], page 26.

The behavior of this function may be altered by the window parameters of *window*, so long as the variable `ignore-window-parameters` is `nil`. If the value of the `delete-window` window parameter is `t`, this function ignores all other window parameters. Otherwise, if the value of the `delete-window` window parameter is a function, that function is called with the argument *window*, in lieu of the usual action of `delete-window`. Otherwise, this function obeys the `window-atom` or `window-side` window parameter, if any. See Section 28.24 [Window Parameters], page 62.

`delete-other-windows` **&optional** *window* [Command]

This function makes *window* fill its frame, by deleting other windows as necessary. If *window* is omitted or `nil`, it defaults to the selected window. The return value is `nil`.

The behavior of this function may be altered by the window parameters of *window*, so long as the variable `ignore-window-parameters` is `nil`. If the value of the `delete-other-windows` window parameter is `t`, this function ignores all other window parameters. Otherwise, if the value of the `delete-other-windows` window parameter is a function, that function is called with the argument *window*, in lieu of the usual action of `delete-other-windows`. Otherwise, this function obeys the `window-atom` or `window-side` window parameter, if any. See Section 28.24 [Window Parameters], page 62.

`delete-windows-on` **&optional** *buffer-or-name frame* [Command]

This function deletes all windows showing *buffer-or-name*, by calling `delete-window` on those windows. *buffer-or-name* should be a buffer, or the name of a buffer; if omitted or `nil`, it defaults to the current buffer. If there are no windows showing the specified buffer, this function does nothing. If the specified buffer is a minibuffer, an error is signaled.

If there is a dedicated window showing the buffer, and that window is the only one on its frame, this function also deletes that frame if it is not the only frame on the terminal.

The optional argument *frame* specifies which frames to operate on:

- `nil` means operate on all frames.
- `t` means operate on the selected frame.
- `visible` means operate on all visible frames.
- `0` means operate on all visible or iconified frames.
- A frame means operate on that frame.

Note that this argument does not have the same meaning as in other functions which scan all live windows (see Section 28.8 [Cyclic Window Ordering], page 34). Specifically, the meanings of `t` and `nil` here are the opposite of what they are in those other functions.

28.7 Selecting Windows

`select-window` *window* **&optional** *norecord* [Function]
: This function makes *window* the selected window, as well as the window selected within its frame (see Section 28.1 [Basic Windows], page 18). *window* must be a live window. Unless *window* already is the selected window, its buffer becomes the current buffer (see Section 28.9 [Buffers and Windows], page 36). The return value is *window*.

  By default, this function also moves *window*'s selected buffer to the front of the buffer list (see Section 27.8 [The Buffer List], page 10), and makes *window* the most recently selected window. However, if the optional argument *norecord* is non-`nil`, these additional actions are omitted.

The sequence of calls to `select-window` with a non-`nil` *norecord* argument determines an ordering of windows by their selection time. The function `get-lru-window` can be used to retrieve the least recently selected live window (see Section 28.8 [Cyclic Window Ordering], page 34).

`save-selected-window` *forms...* [Macro]
: This macro records the selected frame, as well as the selected window of each frame, executes *forms* in sequence, then restores the earlier selected frame and windows. It also saves and restores the current buffer. It returns the value of the last form in *forms*.

  This macro does not save or restore anything about the sizes, arrangement or contents of windows; therefore, if *forms* change them, the change persists. If the previously selected window of some frame is no longer live at the time of exit from *forms*, that frame's selected window is left alone. If the previously selected window is no longer live, then whatever window is selected at the end of *forms* remains selected. The current buffer is restored if and only if it is still live when exiting *forms*.

  This macro changes neither the ordering of recently selected windows nor the buffer list.

`with-selected-window` *window forms...* [Macro]
: This macro selects *window*, executes *forms* in sequence, then restores the previously selected window and current buffer. The ordering of recently selected windows and the buffer list remain unchanged unless you deliberately change them within *forms*; for example, by calling `select-window` with argument *norecord* `nil`.

  This macro does not change the order of recently selected windows or the buffer list.

`frame-selected-window` **&optional** *frame* [Function]
: This function returns the window on *frame* that is selected within that frame. *frame* should be a live frame; if omitted or `nil`, it defaults to the selected frame.

`set-frame-selected-window` *frame window* **&optional** *norecord* [Function]
: This function makes *window* the window selected within the frame *frame*. *frame* should be a live frame; if omitted or `nil`, it defaults to the selected frame. *window* should be a live window; if omitted or `nil`, it defaults to the selected window.

  If *frame* is the selected frame, this makes *window* the selected window.

If the optional argument *norecord* is non-`nil`, this function does not alter the list of most recently selected windows, nor the buffer list.

28.8 Cyclic Ordering of Windows

When you use the command `C-x o` (`other-window`) to select some other window, it moves through live windows in a specific order. For any given configuration of windows, this order never varies. It is called the *cyclic ordering of windows*.

The ordering is determined by a depth-first traversal of the frame's window tree, retrieving the live windows which are the leaf nodes of the tree (see Section 28.2 [Windows and Frames], page 19). If the minibuffer is active, the minibuffer window is included too. The ordering is cyclic, so the last window in the sequence is followed by the first one.

`next-window` **&optional** *window minibuf all-frames* [Function]

This function returns a live window, the one following *window* in the cyclic ordering of windows. *window* should be a live window; if omitted or `nil`, it defaults to the selected window.

The optional argument *minibuf* specifies whether minibuffer windows should be included in the cyclic ordering. Normally, when *minibuf* is `nil`, a minibuffer window is included only if it is currently "active"; this matches the behavior of `C-x o`. (Note that a minibuffer window is active as long as its minibuffer is in use; see Chapter 20 [Minibuffers], page 284, vol. 1).

If *minibuf* is `t`, the cyclic ordering includes all minibuffer windows. If *minibuf* is neither `t` nor `nil`, minibuffer windows are not included even if they are active.

The optional argument *all-frames* specifies which frames to consider:

- `nil` means to consider windows on *window*'s frame. If the minibuffer window is considered (as specified by the *minibuf* argument), then frames that share the minibuffer window are considered too.
- `t` means to consider windows on all existing frames.
- `visible` means to consider windows on all visible frames.
- 0 means to consider windows on all visible or iconified frames.
- A frame means to consider windows on that specific frame.
- Anything else means to consider windows on *window*'s frame, and no others.

If more than one frame is considered, the cyclic ordering is obtained by appending the orderings for those frames, in the same order as the list of all live frames (see Section 29.7 [Finding All Frames], page 82).

`previous-window` **&optional** *window minibuf all-frames* [Function]

This function returns a live window, the one preceding *window* in the cyclic ordering of windows. The other arguments are handled like in `next-window`.

`other-window` *count* **&optional** *all-frames* [Command]

This function selects a live window, one *count* places from the selected window in the cyclic ordering of windows. If *count* is a positive number, it skips *count* windows forwards; if *count* is negative, it skips −*count* windows backwards; if *count* is zero,

that simply re-selects the selected window. When called interactively, *count* is the numeric prefix argument.

The optional argument *all-frames* has the same meaning as in `next-window`, like a `nil` *minibuf* argument to `next-window`.

This function does not select a window that has a non-`nil` `no-other-window` window parameter (see Section 28.24 [Window Parameters], page 62).

`walk-windows` *fun* **&optional** *minibuf all-frames* [Function]

This function calls the function *fun* once for each live window, with the window as the argument.

It follows the cyclic ordering of windows. The optional arguments *minibuf* and *all-frames* specify the set of windows included; these have the same arguments as in `next-window`. If *all-frames* specifies a frame, the first window walked is the first window on that frame (the one returned by `frame-first-window`), not necessarily the selected window.

If *fun* changes the window configuration by splitting or deleting windows, that does not alter the set of windows walked, which is determined prior to calling *fun* for the first time.

`one-window-p` **&optional** *no-mini all-frames* [Function]

This function returns `t` if the selected window is the only live window, and `nil` otherwise.

If the minibuffer window is active, it is normally considered (so that this function returns `nil`). However, if the optional argument *no-mini* is non-`nil`, the minibuffer window is ignored even if active. The optional argument *all-frames* has the same meaning as for `next-window`.

The following functions return a window which satisfies some criterion, without selecting it:

`get-lru-window` **&optional** *all-frames dedicated* [Function]

This function returns a live window which is heuristically the "least recently used" window. The optional argument *all-frames* has the same meaning as in `next-window`.

If any full-width windows are present, only those windows are considered. The selected window is never returned, unless it is the only candidate. A minibuffer window is never a candidate. A dedicated window (see Section 28.15 [Dedicated Windows], page 46) is never a candidate unless the optional argument *dedicated* is non-`nil`.

`get-largest-window` **&optional** *all-frames dedicated* [Function]

This function returns the window with the largest area (height times width). A minibuffer window is never a candidate. A dedicated window (see Section 28.15 [Dedicated Windows], page 46) is never a candidate unless the optional argument *dedicated* is non-`nil`.

If there are two candidate windows of the same size, this function prefers the one that comes first in the cyclic ordering of windows, starting from the selected window.

The optional argument *all-frames* specifies the windows to search, and has the same meaning as in `next-window`.

`get-window-with-predicate` *predicate* **&optional** *minibuf all-frames default* [Function]

This function calls the function *predicate* for each of the windows in the cyclic order of windows in turn, passing it the window as an argument. If the predicate returns non-`nil` for any window, this function stops and returns that window. If no such window is found, the return value is *default* (which defaults to `nil`).

The optional arguments *minibuf* and *all-frames* specify the windows to search, and have the same meanings as in `next-window`.

28.9 Buffers and Windows

This section describes low-level functions for examining and setting the contents of windows. See Section 28.10 [Switching Buffers], page 37, for higher-level functions for displaying a specific buffer in a window.

`window-buffer` **&optional** *window* [Function]

This function returns the buffer that *window* is displaying. If *window* is omitted or `nil` it defaults to the selected window. If *window* is an internal window, this function returns `nil`.

`set-window-buffer` *window buffer-or-name* **&optional** *keep-margins* [Function]

This function makes *window* display *buffer-or-name*. *window* should be a live window; if `nil`, it defaults to the selected window. *buffer-or-name* should be a buffer, or the name of an existing buffer. This function does not change which window is selected, nor does it directly change which buffer is current (see Section 27.2 [Current Buffer], page 1). Its return value is `nil`.

If *window* is *strongly dedicated* to a buffer and *buffer-or-name* does not specify that buffer, this function signals an error. See Section 28.15 [Dedicated Windows], page 46.

By default, this function resets *window*'s position, display margins, fringe widths, and scroll bar settings, based on the local variables in the specified buffer. However, if the optional argument *keep-margins* is non-`nil`, it leaves the display margins and fringe widths unchanged.

When writing an application, you should normally use the higher-level functions described in Section 28.10 [Switching Buffers], page 37, instead of calling `set-window-buffer` directly.

This runs `window-scroll-functions`, followed by `window-configuration-change-hook`. See Section 28.25 [Window Hooks], page 64.

`buffer-display-count` [Variable]

This buffer-local variable records the number of times a buffer has been displayed in a window. It is incremented each time `set-window-buffer` is called for the buffer.

`buffer-display-time` [Variable]

This buffer-local variable records the time at which a buffer was last displayed in a window. The value is `nil` if the buffer has never been displayed. It is updated each time `set-window-buffer` is called for the buffer, with the value returned by `current-time` (see Section 39.5 [Time of Day], page 399).

`get-buffer-window` **&optional** *buffer-or-name all-frames* [Function]
This function returns the first window displaying *buffer-or-name* in the cyclic ordering of windows, starting from the selected window (see Section 28.8 [Cyclic Window Ordering], page 34). If no such window exists, the return value is `nil`.

buffer-or-name should be a buffer or the name of a buffer; if omitted or `nil`, it defaults to the current buffer. The optional argument *all-frames* specifies which windows to consider:

- `t` means consider windows on all existing frames.
- `visible` means consider windows on all visible frames.
- 0 means consider windows on all visible or iconified frames.
- A frame means consider windows on that frame only.
- Any other value means consider windows on the selected frame.

Note that these meanings differ slightly from those of the *all-frames* argument to `next-window` (see Section 28.8 [Cyclic Window Ordering], page 34). This function may be changed in a future version of Emacs to eliminate this discrepancy.

`get-buffer-window-list` **&optional** *buffer-or-name minibuf all-frames* [Function]
This function returns a list of all windows currently displaying *buffer-or-name*. *buffer-or-name* should be a buffer or the name of an existing buffer. If omitted or `nil`, it defaults to the current buffer.

The arguments *minibuf* and *all-frames* have the same meanings as in the function `next-window` (see Section 28.8 [Cyclic Window Ordering], page 34). Note that the *all-frames* argument does *not* behave exactly like in `get-buffer-window`.

`replace-buffer-in-windows` **&optional** *buffer-or-name* [Command]
This command replaces *buffer-or-name* with some other buffer, in all windows displaying it. *buffer-or-name* should be a buffer, or the name of an existing buffer; if omitted or `nil`, it defaults to the current buffer.

The replacement buffer in each window is chosen via `switch-to-prev-buffer` (see Section 28.14 [Window History], page 45). Any dedicated window displaying *buffer-or-name* is deleted (see Section 28.15 [Dedicated Windows], page 46), unless it is the only window on its frame—if it is the only window, and that frame is not the only frame on its terminal, the frame is "dismissed" by calling the function specified by `frame-auto-hide-function` (see Section 28.16 [Quitting Windows], page 47). If the dedicated window is the only window on the only frame on its terminal, the buffer is replaced anyway.

28.10 Switching to a Buffer in a Window

This section describes high-level functions for switching to a specified buffer in some window.

Do *not* use these functions to make a buffer temporarily current just so a Lisp program can access or modify it. They have side-effects, such as changing window histories (see Section 28.14 [Window History], page 45), which will surprise the user if used that way. If you want to make a buffer current to modify it in Lisp, use `with-current-buffer`, `save-current-buffer`, or `set-buffer`. See Section 27.2 [Current Buffer], page 1.

`switch-to-buffer` *buffer-or-name* **&optional** *norecord* *force-same-window* [Command]

This function displays *buffer-or-name* in the selected window, and makes it the current buffer. (In contrast, `set-buffer` makes the buffer current but does not display it; see Section 27.2 [Current Buffer], page 1). It is often used interactively (as the binding of `C-x b`), as well as in Lisp programs. The return value is the buffer switched to.

If *buffer-or-name* is `nil`, it defaults to the buffer returned by `other-buffer` (see Section 27.8 [The Buffer List], page 10). If *buffer-or-name* is a string that is not the name of any existing buffer, this function creates a new buffer with that name; the new buffer's major mode is determined by the variable `major-mode` (see Section 23.2 [Major Modes], page 399, vol. 1).

Normally the specified buffer is put at the front of the buffer list—both the global buffer list and the selected frame's buffer list (see Section 27.8 [The Buffer List], page 10). However, this is not done if the optional argument *norecord* is non-`nil`.

If this function is unable to display the buffer in the selected window—usually because the selected window is a minibuffer window or is strongly dedicated to its buffer (see Section 28.15 [Dedicated Windows], page 46)—then it normally tries to display the buffer in some other window, in the manner of `pop-to-buffer` (see below). However, if the optional argument *force-same-window* is non-`nil`, it signals an error instead.

The next two functions are similar to `switch-to-buffer`, except for the described features.

`switch-to-buffer-other-window` *buffer-or-name* **&optional** *norecord* [Command]

This function makes the buffer specified by *buffer-or-name* current and displays it in some window other than the selected window. It uses the function `pop-to-buffer` internally (see below).

If the selected window already displays the specified buffer, it continues to do so, but another window is nonetheless found to display it as well.

The *buffer-or-name* and *norecord* arguments have the same meanings as in `switch-to-buffer`.

`switch-to-buffer-other-frame` *buffer-or-name* **&optional** *norecord* [Command]

This function makes the buffer specified by *buffer-or-name* current and displays it, usually in a new frame. It uses the function `pop-to-buffer` (see below).

If the specified buffer is already displayed in another window, in any frame on the current terminal, this switches to that window instead of creating a new frame. However, the selected window is never used for this.

The *buffer-or-name* and *norecord* arguments have the same meanings as in `switch-to-buffer`.

The above commands use the function `pop-to-buffer`, which flexibly displays a buffer in some window and selects that window for editing. In turn, `pop-to-buffer` uses `display-buffer` for displaying the buffer. Hence, all the variables affecting `display-buffer` will affect it as well. See Section 28.11 [Choosing Window], page 39, for the documentation of `display-buffer`.

`pop-to-buffer` *buffer-or-name* **&optional** *action norecord* [Command]
: This function makes *buffer-or-name* the current buffer and displays it in some window, preferably not the window previously selected. It then selects the displaying window. If that window is on a different graphical frame, that frame is given input focus if possible (see Section 29.9 [Input Focus], page 83). The return value is the buffer that was switched to.

  If *buffer-or-name* is `nil`, it defaults to the buffer returned by `other-buffer` (see Section 27.8 [The Buffer List], page 10). If *buffer-or-name* is a string that is not the name of any existing buffer, this function creates a new buffer with that name; the new buffer's major mode is determined by the variable `major-mode` (see Section 23.2 [Major Modes], page 399, vol. 1).

  If *action* is non-`nil`, it should be a display action to pass to `display-buffer` (see Section 28.11 [Choosing Window], page 39). Alternatively, a non-`nil`, non-list value means to pop to a window other than the selected one—even if the buffer is already displayed in the selected window.

  Like `switch-to-buffer`, this function updates the buffer list unless *norecord* is non-`nil`.

28.11 Choosing a Window for Display

The command `display-buffer` flexibly chooses a window for display, and displays a specified buffer in that window. It can be called interactively, via the key binding *`C-x 4 C-o`*. It is also used as a subroutine by many functions and commands, including `switch-to-buffer` and `pop-to-buffer` (see Section 28.10 [Switching Buffers], page 37).

This command performs several complex steps to find a window to display in. These steps are described by means of *display actions*, which have the form *`(function . alist)`*. Here, *function* is either a function or a list of functions, which we refer to as *action functions*; *alist* is an association list, which we refer to as *action alists*.

An action function accepts two arguments: the buffer to display and an action alist. It attempts to display the buffer in some window, picking or creating a window according to its own criteria. If successful, it returns the window; otherwise, it returns `nil`. See Section 28.12 [Display Action Functions], page 40, for a list of predefined action functions.

`display-buffer` works by combining display actions from several sources, and calling the action functions in turn, until one of them manages to display the buffer and returns a non-`nil` value.

`display-buffer` *buffer-or-name* **&optional** *action frame* [Command]
: This command makes *buffer-or-name* appear in some window, without selecting the window or making the buffer current. The argument *buffer-or-name* must be a buffer or the name of an existing buffer. The return value is the window chosen to display the buffer.

  The optional argument *action*, if non-`nil`, should normally be a display action (described above). `display-buffer` builds a list of action functions and an action alist, by consolidating display actions from the following sources (in order):

  - The variable `display-buffer-overriding-action`.
  - The user option `display-buffer-alist`.

- A special action for handling `special-display-buffer-names` and `special-display-regexps`, if either of those variables is non-`nil`. See Section 28.13 [Choosing Window Options], page 41.
- The *action* argument.
- The user option `display-buffer-base-action`.
- The constant `display-buffer-fallback-action`.

Each action function is called in turn, passing the buffer as the first argument and the combined action alist as the second argument, until one of the functions returns non-`nil`.

The argument *action* can also have a non-`nil`, non-list value. This has the special meaning that the buffer should be displayed in a window other than the selected one, even if the selected window is already displaying it. If called interactively with a prefix argument, *action* is `t`.

The optional argument *frame*, if non-`nil`, specifies which frames to check when deciding whether the buffer is already displayed. It is equivalent to adding an element `(reusable-frames . frame)` to the action alist of *action*. See Section 28.12 [Display Action Functions], page 40.

`display-buffer-overriding-action` [Variable]
The value of this variable should be a display action, which is treated with the highest priority by `display-buffer`. The default value is empty, i.e. `(nil . nil)`.

`display-buffer-alist` [User Option]
The value of this option is an alist mapping regular expressions to display actions. If the name of the buffer passed to `display-buffer` matches a regular expression in this alist, then `display-buffer` uses the corresponding display action.

`display-buffer-base-action` [User Option]
The value of this option should be a display action. This option can be used to define a "standard" display action for calls to `display-buffer`.

`display-buffer-fallback-action` [Constant]
This display action specifies the fallback behavior for `display-buffer` if no other display actions are given.

28.12 Action Functions for `display-buffer`

The following basic action functions are defined in Emacs. Each of these functions takes two arguments: *buffer*, the buffer to display, and *alist*, an action alist. Each action function returns the window if it succeeds, and `nil` if it fails.

`display-buffer-same-window` *buffer alist* [Function]
This function tries to display *buffer* in the selected window. It fails if the selected window is a minibuffer window or is dedicated to another buffer (see Section 28.15 [Dedicated Windows], page 46). It also fails if *alist* has a non-`nil` `inhibit-same-window` entry.

`display-buffer-reuse-window` *buffer alist* [Function]
This function tries to "display" *buffer* by finding a window that is already displaying it.

If *alist* has a non-`nil` `inhibit-same-window` entry, the selected window is not eligible for reuse. If *alist* contains a `reusable-frames` entry, its value determines which frames to search for a reusable window:

- `nil` means consider windows on the selected frame. (Actually, the last non-minibuffer frame.)
- `t` means consider windows on all frames.
- `visible` means consider windows on all visible frames.
- 0 means consider windows on all visible or iconified frames.
- A frame means consider windows on that frame only.

If *alist* contains no `reusable-frames` entry, this function normally searches just the selected frame; however, if either the variable `display-buffer-reuse-frames` or the variable `pop-up-frames` is non-`nil`, it searches all frames on the current terminal. See Section 28.13 [Choosing Window Options], page 41.

If this function chooses a window on another frame, it makes that frame visible and raises it if necessary.

`display-buffer-pop-up-frame` *buffer alist* [Function]
This function creates a new frame, and displays the buffer in that frame's window. It actually performs the frame creation by calling the function specified in `pop-up-frame-function` (see Section 28.13 [Choosing Window Options], page 41).

`display-buffer-pop-up-window` *buffer alist* [Function]
This function tries to display *buffer* by splitting the largest or least recently-used window (typically one on the selected frame). It actually performs the split by calling the function specified in `split-window-preferred-function` (see Section 28.13 [Choosing Window Options], page 41).

It can fail if no window splitting can be performed for some reason (e.g. if there is just one frame and it has an `unsplittable` frame parameter; see Section 29.3.3.5 [Buffer Parameters], page 75).

`display-buffer-use-some-window` *buffer alist* [Function]
This function tries to display *buffer* by choosing an existing window and displaying the buffer in that window. It can fail if all windows are dedicated to another buffer (see Section 28.15 [Dedicated Windows], page 46).

28.13 Additional Options for Displaying Buffers

The behavior of the standard display actions of `display-buffer` (see Section 28.11 [Choosing Window], page 39) can be modified by a variety of user options.

`display-buffer-reuse-frames` [User Option]
If the value of this variable is non-`nil`, `display-buffer` may search all frames on the current terminal when looking for a window already displaying the specified buffer. The default is `nil`. This variable is consulted by the action function `display-buffer-reuse-window` (see Section 28.12 [Display Action Functions], page 40).

`pop-up-windows` [User Option]
: If the value of this variable is non-`nil`, `display-buffer` is allowed to split an existing window to make a new window for displaying in. This is the default.

  This variable is provided mainly for backward compatibility. It is obeyed by `display-buffer` via a special mechanism in `display-buffer-fallback-action`, which only calls the action function `display-buffer-pop-up-window` (see Section 28.12 [Display Action Functions], page 40) when the value is `nil`. It is not consulted by `display-buffer-pop-up-window` itself, which the user may specify directly in `display-buffer-alist` etc.

`split-window-preferred-function` [User Option]
: This variable specifies a function for splitting a window, in order to make a new window for displaying a buffer. It is used by the `display-buffer-pop-up-window` action function to actually split the window (see Section 28.12 [Display Action Functions], page 40).

  The default value is `split-window-sensibly`, which is documented below. The value must be a function that takes one argument, a window, and return either a new window (which is used to display the desired buffer) or `nil` (which means the splitting failed).

`split-window-sensibly` *window* [Function]
: This function tries to split *window*, and return the newly created window. If *window* cannot be split, it returns `nil`.

  This function obeys the usual rules that determine when a window may be split (see Section 28.5 [Splitting Windows], page 26). It first tries to split by placing the new window below, subject to the restriction imposed by `split-height-threshold` (see below), in addition to any other restrictions. If that fails, it tries to split by placing the new window to the right, subject to `split-width-threshold` (see below). If that fails, and the window is the only window on its frame, this function again tries to split and place the new window below, disregarding `split-height-threshold`. If this fails as well, this function gives up and returns `nil`.

`split-height-threshold` [User Option]
: This variable, used by `split-window-sensibly`, specifies whether to split the window placing the new window below. If it is an integer, that means to split only if the original window has at least that many lines. If it is `nil`, that means not to split this way.

`split-width-threshold` [User Option]
: This variable, used by `split-window-sensibly`, specifies whether to split the window placing the new window to the right. If the value is an integer, that means to split only if the original window has at least that many columns. If the value is `nil`, that means not to split this way.

`pop-up-frames` [User Option]
: If the value of this variable is non-`nil`, that means `display-buffer` may display buffers by making new frames. The default is `nil`.

A non-`nil` value also means that when `display-buffer` is looking for a window already displaying *buffer-or-name*, it can search any visible or iconified frame, not just the selected frame.

This variable is provided mainly for backward compatibility. It is obeyed by `display-buffer` via a special mechanism in `display-buffer-fallback-action`, which calls the action function `display-buffer-pop-up-frame` (see Section 28.12 [Display Action Functions], page 40) if the value is non-`nil`. (This is done before attempting to split a window.) This variable is not consulted by `display-buffer-pop-up-frame` itself, which the user may specify directly in `display-buffer-alist` etc.

`pop-up-frame-function` [User Option]

This variable specifies a function for creating a new frame, in order to make a new window for displaying a buffer. It is used by the `display-buffer-pop-up-frame` action function (see Section 28.12 [Display Action Functions], page 40).

The value should be a function that takes no arguments and returns a frame, or `nil` if no frame could be created. The default value is a function that creates a frame using the parameters specified by `pop-up-frame-alist` (see below).

`pop-up-frame-alist` [User Option]

This variable holds an alist of frame parameters (see Section 29.3 [Frame Parameters], page 70), which is used by the default function in `pop-up-frame-function` to make a new frame. The default is `nil`.

`special-display-buffer-names` [User Option]

A list of buffer names identifying buffers that should be displayed specially. If the name of *buffer-or-name* is in this list, `display-buffer` handles the buffer specially. By default, special display means to give the buffer a dedicated frame.

If an element is a list, instead of a string, then the CAR of that list is the buffer name, and the rest of that list says how to create the frame. There are two possibilities for the rest of that list (its CDR): It can be an alist, specifying frame parameters, or it can contain a function and arguments to give to it. (The function's first argument is always the buffer to be displayed; the arguments from the list come after that.)

For example:

```
(("myfile" (minibuffer) (menu-bar-lines . 0)))
```

specifies to display a buffer named '`myfile`' in a dedicated frame with specified `minibuffer` and `menu-bar-lines` parameters.

The list of frame parameters can also use the phony frame parameters `same-frame` and `same-window`. If the specified frame parameters include `(same-window . value)` and *value* is non-`nil`, that means to display the buffer in the current selected window. Otherwise, if they include `(same-frame . value)` and *value* is non-`nil`, that means to display the buffer in a new window in the currently selected frame.

`special-display-regexps` [User Option]

A list of regular expressions specifying buffers that should be displayed specially. If the buffer's name matches any of the regular expressions in this list, `display-buffer` handles the buffer specially. By default, special display means to give the buffer a dedicated frame.

If an element is a list, instead of a string, then the CAR of the list is the regular expression, and the rest of the list says how to create the frame. See `special-display-buffer-names` above.

`special-display-p` *buffer-name* [Function]
This function returns non-`nil` if displaying a buffer named *buffer-name* with `display-buffer` would create a special frame. The value is `t` if it would use the default frame parameters, or else the specified list of frame parameters.

`special-display-function` [User Option]
This variable holds the function to call to display a buffer specially. It receives the buffer as an argument, and should return the window in which it is displayed. The default value of this variable is `special-display-popup-frame`, see below.

`special-display-popup-frame` *buffer* **&optional** *args* [Function]
This function tries to make *buffer* visible in a frame of its own. If *buffer* is already displayed in some window, it makes that window's frame visible and raises it. Otherwise, it creates a frame that is dedicated to *buffer*. The return value is the window used to display *buffer*.

If *args* is an alist, it specifies frame parameters for the new frame. If *args* is a list whose CAR is a symbol, then `(car args)` is a function to actually create and set up the frame; it is called with *buffer* as first argument, and `(cdr args)` as additional arguments.

This function always uses an existing window displaying *buffer*, whether or not it is in a frame of its own; but if you set up the above variables in your init file, before *buffer* was created, then presumably the window was previously made by this function.

`special-display-frame-alist` [User Option]
This variable holds frame parameters for `special-display-popup-frame` to use when it creates a frame.

`same-window-buffer-names` [User Option]
A list of buffer names for buffers that should be displayed in the selected window. If a buffer's name is in this list, `display-buffer` handles the buffer by switching to it in the selected window.

`same-window-regexps` [User Option]
A list of regular expressions that specify buffers that should be displayed in the selected window. If the buffer's name matches any of the regular expressions in this list, `display-buffer` handles the buffer by switching to it in the selected window.

`same-window-p` *buffer-name* [Function]
This function returns `t` if displaying a buffer named *buffer-name* with `display-buffer` would put it in the selected window.

`display-buffer-function` [User Option]
This variable is the most flexible way to customize the behavior of `display-buffer`. If it is non-`nil`, it should be a function that `display-buffer` calls to do the work.

The function should accept two arguments, the first two arguments that `display-buffer` received. It should choose or create a window, display the specified buffer in it, and then return the window.

This variable takes precedence over all the other options described above.

28.14 Window History

Each window remembers the buffers it has previously displayed, and the order in which these buffers were removed from it. This history is used, for example, by `replace-buffer-in-windows` (see Section 28.9 [Buffers and Windows], page 36). This list is automatically maintained by Emacs, but you can use the following functions to explicitly inspect or alter it:

`window-prev-buffers` **&optional** *window* [Function]

This function returns a list specifying the previous contents of *window*, which should be a live window and defaults to the selected window.

Each list element has the form `(buffer window-start window-pos)`, where *buffer* is a buffer previously shown in the window, *window-start* is the window start position when that buffer was last shown, and *window-pos* is the point position when that buffer was last shown.

The list is ordered so that earlier elements correspond to more recently-shown buffers, and the first element usually corresponds to the buffer most recently removed from the window.

`set-window-prev-buffers` *window prev-buffers* [Function]

This function sets *window*'s previous buffers to the value of *prev-buffers*. The argument *window* must be a live window and defaults to the selected one. The argument *prev-buffers* should be a list of the same form as that returned by `window-prev-buffers`.

In addition, each buffer maintains a list of *next buffers*, which is a list of buffers re-shown by `switch-to-prev-buffer` (see below). This list is mainly used by `switch-to-prev-buffer` and `switch-to-next-buffer` for choosing buffers to switch to.

`window-next-buffers` **&optional** *window* [Function]

This function returns the list of buffers recently re-shown in *window* via `switch-to-prev-buffer`. The *window* argument must denote a live window or `nil` (meaning the selected window).

`set-window-next-buffers` *window next-buffers* [Function]

This function sets the next buffer list of *window* to *next-buffers*. The *window* argument should be a live window or `nil` (meaning the selected window). The argument *next-buffers* should be a list of buffers.

The following commands can be used to cycle through the global buffer list, much like `bury-buffer` and `unbury-buffer`. However, they cycle according to the specified window's history list, rather than the global buffer list. In addition, they restore window-specific

window start and point positions, and may show a buffer even if it is already shown in another window. The `switch-to-prev-buffer` command, in particular, is used by `replace-buffer-in-windows`, `bury-buffer` and `quit-window` to find a replacement buffer for a window.

`switch-to-prev-buffer` **&optional** *window bury-or-kill* [Command]
: This command displays the previous buffer in *window*. The argument *window* should be a live window or `nil` (meaning the selected window). If the optional argument *bury-or-kill* is non-`nil`, this means that the buffer currently shown in *window* is about to be buried or killed and consequently should not be switched to in future invocations of this command.

  The previous buffer is usually the buffer shown before the buffer currently shown in *window*. However, a buffer that has been buried or killed, or has been already shown by a recent invocation of `switch-to-prev-buffer`, does not qualify as previous buffer.

  If repeated invocations of this command have already shown all buffers previously shown in *window*, further invocations will show buffers from the buffer list of the frame *window* appears on (see Section 27.8 [The Buffer List], page 10), trying to skip buffers that are already shown in another window on that frame.

`switch-to-next-buffer` **&optional** *window* [Command]
: This command switches to the next buffer in *window*, thus undoing the effect of the last `switch-to-prev-buffer` command in *window*. The argument *window* must be a live window and defaults to the selected one.

  If there is no recent invocation of `switch-to-prev-buffer` that can be undone, this function tries to show a buffer from the buffer list of the frame *window* appears on (see Section 27.8 [The Buffer List], page 10).

By default `switch-to-prev-buffer` and `switch-to-next-buffer` can switch to a buffer that is already shown in another window on the same frame. The following option can be used to override this behavior.

`switch-to-visible-buffer` [User Option]
: If this variable is non-`nil`, `switch-to-prev-buffer` and `switch-to-next-buffer` may switch to a buffer that is already visible on the same frame, provided the buffer was shown in the relevant window before. If it is `nil`, `switch-to-prev-buffer` and `switch-to-next-buffer` always try to avoid switching to a buffer that is already visible in another window on the same frame.

28.15 Dedicated Windows

Functions for displaying a buffer can be told to not use specific windows by marking these windows as *dedicated* to their buffers. `display-buffer` (see Section 28.11 [Choosing Window], page 39) never uses a dedicated window for displaying another buffer in it. `get-lru-window` and `get-largest-window` (see Section 28.7 [Selecting Windows], page 33) do not consider dedicated windows as candidates when their *dedicated* argument is non-`nil`. The behavior of `set-window-buffer` (see Section 28.9 [Buffers and Windows], page 36) with respect to dedicated windows is slightly different, see below.

When `delete-windows-on` (see Section 28.6 [Deleting Windows], page 31) wants to delete a dedicated window and that window is the only window on its frame, it deletes the window's frame too, provided there are other frames left. `replace-buffer-in-windows` (see Section 28.10 [Switching Buffers], page 37) tries to delete all dedicated windows showing its buffer argument. When such a window is the only window on its frame, that frame is deleted, provided there are other frames left. If there are no more frames left, some other buffer is displayed in the window, and the window is marked as non-dedicated.

When you kill a buffer (see Section 27.10 [Killing Buffers], page 13) displayed in a dedicated window, any such window usually gets deleted too, since `kill-buffer` calls `replace-buffer-in-windows` for cleaning up windows. Burying a buffer (see Section 27.8 [The Buffer List], page 10) deletes the selected window if it is dedicated to that buffer. If, however, that window is the only window on its frame, `bury-buffer` displays another buffer in it and iconifies the frame.

`window-dedicated-p` **&optional** *window* [Function]

This function returns non-`nil` if *window* is dedicated to its buffer and `nil` otherwise. More precisely, the return value is the value assigned by the last call of `set-window-dedicated-p` for *window*, or `nil` if that function was never called with *window* as its argument. The default for *window* is the selected window.

`set-window-dedicated-p` *window flag* [Function]

This function marks *window* as dedicated to its buffer if *flag* is non-`nil`, and non-dedicated otherwise.

As a special case, if *flag* is `t`, *window* becomes *strongly* dedicated to its buffer. `set-window-buffer` signals an error when the window it acts upon is strongly dedicated to its buffer and does not already display the buffer it is asked to display. Other functions do not treat `t` differently from any non-`nil` value.

28.16 Quitting Windows

When you want to get rid of a window used for displaying a buffer, you can call `delete-window` or `delete-windows-on` (see Section 28.6 [Deleting Windows], page 31) to remove that window from its frame. If the buffer is shown on a separate frame, you might want to call `delete-frame` (see Section 29.6 [Deleting Frames], page 82) instead. If, on the other hand, a window has been reused for displaying the buffer, you might prefer showing the buffer previously shown in that window, by calling the function `switch-to-prev-buffer` (see Section 28.14 [Window History], page 45). Finally, you might want to either bury (see Section 27.8 [The Buffer List], page 10) or kill (see Section 27.10 [Killing Buffers], page 13) the window's buffer.

The following function uses information on how the window for displaying the buffer was obtained in the first place, thus attempting to automate the above decisions for you.

`quit-window` **&optional** *kill window* [Command]

This command quits *window* and buries its buffer. The argument *window* must be a live window and defaults to the selected one. With prefix argument *kill* non-`nil`, it kills the buffer instead of burying it.

Quitting *window* means to proceed as follows: If *window* was created specially for displaying its current buffer, delete *window* provided its frame contains at least one

other live window. If *window* is the only window on its frame and there are other frames on the frame's terminal, the value of *kill* determines how to proceed with the window. If *kill* is `nil`, the fate of the frame is determined by calling `frame-auto-hide-function` (see below) with that frame as sole argument. If *kill* is non-`nil`, the frame is deleted unconditionally.

If *window* was reused for displaying its buffer, this command tries to display the buffer previously shown in it. It also tries to restore the window start (see Section 28.18 [Window Start and End], page 49) and point (see Section 28.17 [Window Point], page 48) positions of the previously shown buffer. If, in addition, the current buffer was temporarily resized, this command will also try to restore the original height of *window*.

The three cases described so far require that the buffer shown in *window* is still the buffer displayed by the last buffer display function for this window. If another buffer has been shown in the meantime, or the buffer previously shown no longer exists, this command calls `switch-to-prev-buffer` (see Section 28.14 [Window History], page 45) to show some other buffer instead.

The function `quit-window` bases its decisions on information stored in *window*'s `quit-restore` window parameter (see Section 28.24 [Window Parameters], page 62), and resets that parameter to `nil` after it's done.

The following option specifies how to deal with a frame containing just one window that should be either quit, or whose buffer should be buried.

`frame-auto-hide-function` [User Option]

The function specified by this option is called to automatically hide frames. This function is called with one argument—a frame.

The function specified here is called by `bury-buffer` (see Section 27.8 [The Buffer List], page 10) when the selected window is dedicated and shows the buffer that should be buried. It is also called by `quit-window` (see above) when the frame of the window that should be quit has been specially created for displaying that window's buffer and the buffer should be buried.

The default is to call `iconify-frame` (see Section 29.10 [Visibility of Frames], page 85). Alternatively, you may specify either `delete-frame` (see Section 29.6 [Deleting Frames], page 82) to remove the frame from its display, `ignore` to leave the frame unchanged, or any other function that can take a frame as its sole argument.

Note that the function specified by this option is called if and only if there is at least one other frame on the terminal of the frame it's supposed to handle, and that frame contains only one live window.

28.17 Windows and Point

Each window has its own value of point (see Section 30.1 [Point], page 99), independent of the value of point in other windows displaying the same buffer. This makes it useful to have multiple windows showing one buffer.

- The window point is established when a window is first created; it is initialized from the buffer's point, or from the window point of another window opened on the buffer if such a window exists.

- Selecting a window sets the value of point in its buffer from the window's value of point. Conversely, deselecting a window sets the window's value of point from that of the buffer. Thus, when you switch between windows that display a given buffer, the point value for the selected window is in effect in the buffer, while the point values for the other windows are stored in those windows.
- As long as the selected window displays the current buffer, the window's point and the buffer's point always move together; they remain equal.

As far as the user is concerned, point is where the cursor is, and when the user switches to another buffer, the cursor jumps to the position of point in that buffer.

`window-point` **&optional** *window* [Function]
This function returns the current position of point in *window*. For a nonselected window, this is the value point would have (in that window's buffer) if that window were selected. The default for *window* is the selected window.

When *window* is the selected window and its buffer is also the current buffer, the value returned is the same as point in that buffer. Strictly speaking, it would be more correct to return the "top-level" value of point, outside of any `save-excursion` forms. But that value is hard to find.

`set-window-point` *window position* [Function]
This function positions point in *window* at position *position* in *window*'s buffer. It returns *position*.

If *window* is selected, and its buffer is current, this simply does `goto-char`.

`window-point-insertion-type` [Variable]
This variable specifies the marker insertion type (see Section 31.5 [Marker Insertion Types], page 116) of `window-point`. The default is `nil`, so `window-point` will stay behind text inserted there.

28.18 The Window Start and End Positions

Each window maintains a marker used to keep track of a buffer position that specifies where in the buffer display should start. This position is called the *display-start* position of the window (or just the *start*). The character after this position is the one that appears at the upper left corner of the window. It is usually, but not inevitably, at the beginning of a text line.

After switching windows or buffers, and in some other cases, if the window start is in the middle of a line, Emacs adjusts the window start to the start of a line. This prevents certain operations from leaving the window start at a meaningless point within a line. This feature may interfere with testing some Lisp code by executing it using the commands of Lisp mode, because they trigger this readjustment. To test such code, put it into a command and bind the command to a key.

`window-start` **&optional** *window* [Function]
This function returns the display-start position of window *window*. If *window* is `nil`, the selected window is used.

When you create a window, or display a different buffer in it, the display-start position is set to a display-start position recently used for the same buffer, or to `point-min` if the buffer doesn't have any.

Redisplay updates the window-start position (if you have not specified it explicitly since the previous redisplay)—to make sure point appears on the screen. Nothing except redisplay automatically changes the window-start position; if you move point, do not expect the window-start position to change in response until after the next redisplay.

`window-end` **&optional** *window update* [Function]
This function returns the position where display of its buffer ends in *window*. The default for *window* is the selected window.

Simply changing the buffer text or moving point does not update the value that `window-end` returns. The value is updated only when Emacs redisplays and redisplay completes without being preempted.

If the last redisplay of *window* was preempted, and did not finish, Emacs does not know the position of the end of display in that window. In that case, this function returns `nil`.

If *update* is non-`nil`, `window-end` always returns an up-to-date value for where display ends, based on the current `window-start` value. If a previously saved value of that position is still valid, `window-end` returns that value; otherwise it computes the correct value by scanning the buffer text.

Even if *update* is non-`nil`, `window-end` does not attempt to scroll the display if point has moved off the screen, the way real redisplay would do. It does not alter the `window-start` value. In effect, it reports where the displayed text will end if scrolling is not required.

`set-window-start` *window position* **&optional** *noforce* [Function]
This function sets the display-start position of *window* to *position* in *window*'s buffer. It returns *position*.

The display routines insist that the position of point be visible when a buffer is displayed. Normally, they change the display-start position (that is, scroll the window) whenever necessary to make point visible. However, if you specify the start position with this function using `nil` for *noforce*, it means you want display to start at *position* even if that would put the location of point off the screen. If this does place point off screen, the display routines move point to the left margin on the middle line in the window.

For example, if point is 1 and you set the start of the window to 37, the start of the next line, point will be "above" the top of the window. The display routines will automatically move point if it is still 1 when redisplay occurs. Here is an example:

```
;; Here is what 'foo' looks like before executing
;;    the set-window-start expression.
```

```
---------- Buffer: foo ----------
⋆This is the contents of buffer foo.
2
3
4
5
6
---------- Buffer: foo ----------

(set-window-start
 (selected-window)
 (save-excursion
   (goto-char 1)
   (forward-line 1)
   (point)))
⇒ 37

;; Here is what 'foo' looks like after executing
;;    the set-window-start expression.
---------- Buffer: foo ----------
2
3
⋆4
5
6
---------- Buffer: foo ----------
```

If *noforce* is non-`nil`, and *position* would place point off screen at the next redisplay, then redisplay computes a new window-start position that works well with point, and thus *position* is not used.

`pos-visible-in-window-p` **&optional** *position window partially* [Function]

This function returns non-`nil` if *position* is within the range of text currently visible on the screen in *window*. It returns `nil` if *position* is scrolled vertically out of view. Locations that are partially obscured are not considered visible unless *partially* is non-`nil`. The argument *position* defaults to the current position of point in *window*; *window*, to the selected window. If *position* is `t`, that means to check the last visible position in *window*.

This function considers only vertical scrolling. If *position* is out of view only because *window* has been scrolled horizontally, `pos-visible-in-window-p` returns non-`nil` anyway. See Section 28.21 [Horizontal Scrolling], page 56.

If *position* is visible, `pos-visible-in-window-p` returns `t` if *partially* is `nil`; if *partially* is non-`nil`, and the character following *position* is fully visible, it returns a list of the form `(x y)`, where *x* and *y* are the pixel coordinates relative to the top left corner of the window; otherwise it returns an extended list of the form `(x y rtop rbot rowh vpos)`, where *rtop* and *rbot* specify the number of off-window pixels at the top and bottom of the row at *position*, *rowh* specifies the visible height of that row, and *vpos* specifies the vertical position (zero-based row number) of that row.

Here is an example:

```
;; If point is off the screen now, recenter it now.
(or (pos-visible-in-window-p
     (point) (selected-window))
    (recenter 0))
```

`window-line-height` **&optional** *line window* [Function]

This function returns the height of text line *line* in *window*. If *line* is one of `header-line` or `mode-line`, `window-line-height` returns information about the corresponding line of the window. Otherwise, *line* is a text line number starting from 0. A negative number counts from the end of the window. The default for *line* is the current line in *window*; the default for *window* is the selected window.

If the display is not up to date, `window-line-height` returns `nil`. In that case, `pos-visible-in-window-p` may be used to obtain related information.

If there is no line corresponding to the specified *line*, `window-line-height` returns `nil`. Otherwise, it returns a list `(`*`height vpos ypos offbot`*`)`, where *height* is the height in pixels of the visible part of the line, *vpos* and *ypos* are the vertical position in lines and pixels of the line relative to the top of the first text line, and *offbot* is the number of off-window pixels at the bottom of the text line. If there are off-window pixels at the top of the (first) text line, *ypos* is negative.

28.19 Textual Scrolling

Textual scrolling means moving the text up or down through a window. It works by changing the window's display-start location. It may also change the value of `window-point` to keep point on the screen (see Section 28.17 [Window Point], page 48).

The basic textual scrolling functions are `scroll-up` (which scrolls forward) and `scroll-down` (which scrolls backward). In these function names, "up" and "down" refer to the direction of motion of the buffer text relative to the window. Imagine that the text is written on a long roll of paper and that the scrolling commands move the paper up and down. Thus, if you are looking at the middle of a buffer and repeatedly call `scroll-down`, you will eventually see the beginning of the buffer.

Unfortunately, this sometimes causes confusion, because some people tend to think in terms of the opposite convention: they imagine the window moving over text that remains in place, so that "down" commands take you to the end of the buffer. This convention is consistent with fact that such a command is bound to a key named `PageDown` on modern keyboards.

Textual scrolling functions (aside from `scroll-other-window`) have unpredictable results if the current buffer is not the one displayed in the selected window. See Section 27.2 [Current Buffer], page 1.

If the window contains a row taller than the height of the window (for example in the presence of a large image), the scroll functions will adjust the window's vertical scroll position to scroll the partially visible row. Lisp callers can disable this feature by binding the variable `auto-window-vscroll` to `nil` (see Section 28.20 [Vertical Scrolling], page 55).

`scroll-up` **&optional** *count* [Command]

This function scrolls forward by *count* lines in the selected window.

If *count* is negative, it scrolls backward instead. If *count* is `nil` (or omitted), the distance scrolled is `next-screen-context-lines` lines less than the height of the window's text area.

If the selected window cannot be scrolled any further, this function signals an error. Otherwise, it returns `nil`.

`scroll-down` **&optional** *count* [Command]

This function scrolls backward by *count* lines in the selected window.

If *count* is negative, it scrolls forward instead. In other respects, it behaves the same way as `scroll-up` does.

`scroll-up-command` **&optional** *count* [Command]

This behaves like `scroll-up`, except that if the selected window cannot be scrolled any further and the value of the variable `scroll-error-top-bottom` is `t`, it tries to move to the end of the buffer instead. If point is already there, it signals an error.

`scroll-down-command` **&optional** *count* [Command]

This behaves like `scroll-down`, except that if the selected window cannot be scrolled any further and the value of the variable `scroll-error-top-bottom` is `t`, it tries to move to the beginning of the buffer instead. If point is already there, it signals an error.

`scroll-other-window` **&optional** *count* [Command]

This function scrolls the text in another window upward *count* lines. Negative values of *count*, or `nil`, are handled as in `scroll-up`.

You can specify which buffer to scroll by setting the variable `other-window-scroll-buffer` to a buffer. If that buffer isn't already displayed, `scroll-other-window` displays it in some window.

When the selected window is the minibuffer, the next window is normally the leftmost one immediately above it. You can specify a different window to scroll, when the minibuffer is selected, by setting the variable `minibuffer-scroll-window`. This variable has no effect when any other window is selected. When it is non-`nil` and the minibuffer is selected, it takes precedence over `other-window-scroll-buffer`. See [Definition of minibuffer-scroll-window], page 313, vol. 1.

When the minibuffer is active, it is the next window if the selected window is the one at the bottom right corner. In this case, `scroll-other-window` attempts to scroll the minibuffer. If the minibuffer contains just one line, it has nowhere to scroll to, so the line reappears after the echo area momentarily displays the message '`End of buffer`'.

`other-window-scroll-buffer` [Variable]

If this variable is non-`nil`, it tells `scroll-other-window` which buffer's window to scroll.

`scroll-margin` [User Option]

This option specifies the size of the scroll margin—a minimum number of lines between point and the top or bottom of a window. Whenever point gets within this many lines of the top or bottom of the window, redisplay scrolls the text automatically (if possible) to move point out of the margin, closer to the center of the window.

`scroll-conservatively` [User Option]

This variable controls how scrolling is done automatically when point moves off the screen (or into the scroll margin). If the value is a positive integer *n*, then redisplay scrolls the text up to *n* lines in either direction, if that will bring point back into proper view. This behavior is called *conservative scrolling*. Otherwise, scrolling happens in the usual way, under the control of other variables such as `scroll-up-aggressively` and `scroll-down-aggressively`.

The default value is zero, which means that conservative scrolling never happens.

`scroll-down-aggressively` [User Option]

The value of this variable should be either `nil` or a fraction *f* between 0 and 1. If it is a fraction, that specifies where on the screen to put point when scrolling down. More precisely, when a window scrolls down because point is above the window start, the new start position is chosen to put point *f* part of the window height from the top. The larger *f*, the more aggressive the scrolling.

A value of `nil` is equivalent to .5, since its effect is to center point. This variable automatically becomes buffer-local when set in any fashion.

`scroll-up-aggressively` [User Option]

Likewise, for scrolling up. The value, *f*, specifies how far point should be placed from the bottom of the window; thus, as with `scroll-up-aggressively`, a larger value scrolls more aggressively.

`scroll-step` [User Option]

This variable is an older variant of `scroll-conservatively`. The difference is that if its value is *n*, that permits scrolling only by precisely *n* lines, not a smaller number. This feature does not work with `scroll-margin`. The default value is zero.

`scroll-preserve-screen-position` [User Option]

If this option is `t`, whenever a scrolling command moves point off-window, Emacs tries to adjust point to keep the cursor at its old vertical position in the window, rather than the window edge.

If the value is non-`nil` and not `t`, Emacs adjusts point to keep the cursor at the same vertical position, even if the scrolling command didn't move point off-window.

This option affects all scroll commands that have a non-`nil` `scroll-command` symbol property.

`next-screen-context-lines` [User Option]

The value of this variable is the number of lines of continuity to retain when scrolling by full screens. For example, `scroll-up` with an argument of `nil` scrolls so that this many lines at the bottom of the window appear instead at the top. The default value is 2.

`scroll-error-top-bottom` [User Option]

If this option is `nil` (the default), `scroll-up-command` and `scroll-down-command` simply signal an error when no more scrolling is possible.

If the value is `t`, these commands instead move point to the beginning or end of the buffer (depending on scrolling direction); only if point is already on that position do they signal an error.

`recenter` **&optional** *count* [Command]
: This function scrolls the text in the selected window so that point is displayed at a specified vertical position within the window. It does not "move point" with respect to the text.

  If *count* is a non-negative number, that puts the line containing point *count* lines down from the top of the window. If *count* is a negative number, then it counts upward from the bottom of the window, so that −1 stands for the last usable line in the window.

  If *count* is `nil` (or a non-`nil` list), `recenter` puts the line containing point in the middle of the window. If *count* is `nil`, this function may redraw the frame, according to the value of `recenter-redisplay`.

  When `recenter` is called interactively, *count* is the raw prefix argument. Thus, typing `C-u` as the prefix sets the *count* to a non-`nil` list, while typing `C-u 4` sets *count* to 4, which positions the current line four lines from the top.

  With an argument of zero, `recenter` positions the current line at the top of the window. The command `recenter-top-bottom` offers a more convenient way to achieve this.

`recenter-redisplay` [User Option]
: If this variable is non-`nil`, calling `recenter` with a `nil` argument redraws the frame. The default value is `tty`, which means only redraw the frame if it is a tty frame.

`recenter-top-bottom` **&optional** *count* [Command]
: This command, which is the default binding for `C-l`, acts like `recenter`, except if called with no argument. In that case, successive calls place point according to the cycling order defined by the variable `recenter-positions`.

`recenter-positions` [User Option]
: This variable controls how `recenter-top-bottom` behaves when called with no argument. The default value is `(middle top bottom)`, which means that successive calls of `recenter-top-bottom` with no argument cycle between placing point at the middle, top, and bottom of the window.

28.20 Vertical Fractional Scrolling

Vertical fractional scrolling means shifting text in a window up or down by a specified multiple or fraction of a line. Each window has a *vertical scroll position*, which is a number, never less than zero. It specifies how far to raise the contents of the window. Raising the window contents generally makes all or part of some lines disappear off the top, and all or part of some other lines appear at the bottom. The usual value is zero.

The vertical scroll position is measured in units of the normal line height, which is the height of the default font. Thus, if the value is .5, that means the window contents are scrolled up half the normal line height. If it is 3.3, that means the window contents are scrolled up somewhat over three times the normal line height.

What fraction of a line the vertical scrolling covers, or how many lines, depends on what the lines contain. A value of .5 could scroll a line whose height is very short off the screen, while a value of 3.3 could scroll just part of the way through a tall line or an image.

`window-vscroll` **&optional** *window pixels-p* [Function]
This function returns the current vertical scroll position of *window*. The default for *window* is the selected window. If *pixels-p* is non-`nil`, the return value is measured in pixels, rather than in units of the normal line height.

```
(window-vscroll)
     ⇒ 0
```

`set-window-vscroll` *window lines* **&optional** *pixels-p* [Function]
This function sets *window*'s vertical scroll position to *lines*. If *window* is `nil`, the selected window is used. The argument *lines* should be zero or positive; if not, it is taken as zero.

The actual vertical scroll position must always correspond to an integral number of pixels, so the value you specify is rounded accordingly.

The return value is the result of this rounding.

```
(set-window-vscroll (selected-window) 1.2)
     ⇒ 1.13
```

If *pixels-p* is non-`nil`, *lines* specifies a number of pixels. In this case, the return value is *lines*.

`auto-window-vscroll` [Variable]
If this variable is non-`nil`, the line-move, scroll-up, and scroll-down functions will automatically modify the vertical scroll position to scroll through display rows that are taller than the height of the window, for example in the presence of large images.

28.21 Horizontal Scrolling

Horizontal scrolling means shifting the image in the window left or right by a specified multiple of the normal character width. Each window has a *horizontal scroll position*, which is a number, never less than zero. It specifies how far to shift the contents left. Shifting the window contents left generally makes all or part of some characters disappear off the left, and all or part of some other characters appear at the right. The usual value is zero.

The horizontal scroll position is measured in units of the normal character width, which is the width of space in the default font. Thus, if the value is 5, that means the window contents are scrolled left by 5 times the normal character width. How many characters actually disappear off to the left depends on their width, and could vary from line to line.

Because we read from side to side in the "inner loop", and from top to bottom in the "outer loop", the effect of horizontal scrolling is not like that of textual or vertical scrolling. Textual scrolling involves selection of a portion of text to display, and vertical scrolling moves the window contents contiguously; but horizontal scrolling causes part of *each line* to go off screen.

Usually, no horizontal scrolling is in effect; then the leftmost column is at the left edge of the window. In this state, scrolling to the right is meaningless, since there is no data to the left of the edge to be revealed by it; so this is not allowed. Scrolling to the left is allowed; it scrolls the first columns of text off the edge of the window and can reveal additional columns on the right that were truncated before. Once a window has a nonzero amount of leftward

horizontal scrolling, you can scroll it back to the right, but only so far as to reduce the net horizontal scroll to zero. There is no limit to how far left you can scroll, but eventually all the text will disappear off the left edge.

If `auto-hscroll-mode` is set, redisplay automatically alters the horizontal scrolling of a window as necessary to ensure that point is always visible. However, you can still set the horizontal scrolling value explicitly. The value you specify serves as a lower bound for automatic scrolling, i.e. automatic scrolling will not scroll a window to a column less than the specified one.

`scroll-left` **&optional** *count set-minimum* [Command]

This function scrolls the selected window *count* columns to the left (or to the right if *count* is negative). The default for *count* is the window width, minus 2.

The return value is the total amount of leftward horizontal scrolling in effect after the change—just like the value returned by `window-hscroll` (below).

Once you scroll a window as far right as it can go, back to its normal position where the total leftward scrolling is zero, attempts to scroll any farther right have no effect.

If *set-minimum* is non-`nil`, the new scroll amount becomes the lower bound for automatic scrolling; that is, automatic scrolling will not scroll a window to a column less than the value returned by this function. Interactive calls pass non-`nil` for *set-minimum*.

`scroll-right` **&optional** *count set-minimum* [Command]

This function scrolls the selected window *count* columns to the right (or to the left if *count* is negative). The default for *count* is the window width, minus 2. Aside from the direction of scrolling, this works just like `scroll-left`.

`window-hscroll` **&optional** *window* [Function]

This function returns the total leftward horizontal scrolling of *window*—the number of columns by which the text in *window* is scrolled left past the left margin. The default for *window* is the selected window.

The return value is never negative. It is zero when no horizontal scrolling has been done in *window* (which is usually the case).

```
(window-hscroll)
     ⇒ 0
(scroll-left 5)
     ⇒ 5
(window-hscroll)
     ⇒ 5
```

`set-window-hscroll` *window columns* [Function]

This function sets horizontal scrolling of *window*. The value of *columns* specifies the amount of scrolling, in terms of columns from the left margin. The argument *columns* should be zero or positive; if not, it is taken as zero. Fractional values of *columns* are not supported at present.

Note that `set-window-hscroll` may appear not to work if you test it by evaluating a call with `M-:` in a simple way. What happens is that the function sets the horizontal scroll value and returns, but then redisplay adjusts the horizontal scrolling to make

point visible, and this overrides what the function did. You can observe the function's effect if you call it while point is sufficiently far from the left margin that it will remain visible.

The value returned is *columns*.

```
(set-window-hscroll (selected-window) 10)
     ⇒ 10
```

Here is how you can determine whether a given position *position* is off the screen due to horizontal scrolling:

```
(defun hscroll-on-screen (window position)
  (save-excursion
    (goto-char position)
    (and
     (>= (- (current-column) (window-hscroll window)) 0)
     (< (- (current-column) (window-hscroll window))
        (window-width window)))))
```

28.22 Coordinates and Windows

This section describes functions that report the position of a window. Most of these functions report positions relative to the window's frame. In this case, the coordinate origin '`(0,0)`' lies near the upper left corner of the frame. For technical reasons, on graphical displays the origin is not located at the exact corner of the graphical window as it appears on the screen. If Emacs is built with the GTK+ toolkit, the origin is at the upper left corner of the frame area used for displaying Emacs windows, below the title-bar, GTK+ menu bar, and tool bar (since these are drawn by the window manager and/or GTK+, not by Emacs). But if Emacs is not built with GTK+, the origin is at the upper left corner of the tool bar (since in this case Emacs itself draws the tool bar). In both cases, the X and Y coordinates increase rightward and downward respectively.

Except where noted, X and Y coordinates are reported in integer character units, i.e. numbers of lines and columns respectively. On a graphical display, each "line" and "column" corresponds to the height and width of a default character specified by the frame's default font.

`window-edges` **&optional** *window* [Function]

This function returns a list of the edge coordinates of *window*. If *window* is omitted or `nil`, it defaults to the selected window.

The return value has the form (*left top right bottom*). These list elements are, respectively, the X coordinate of the leftmost column occupied by the window, the Y coordinate of the topmost row, the X coordinate one column to the right of the rightmost column, and the Y coordinate one row down from the bottommost row.

Note that these are the actual outer edges of the window, including any header line, mode line, scroll bar, fringes, and display margins. On a text terminal, if the window has a neighbor on its right, its right edge includes the separator line between the window and its neighbor.

`window-inside-edges` **&optional** *window* [Function]
: This function is similar to `window-edges`, but the returned edge values are for the text area of the window. They exclude any header line, mode line, scroll bar, fringes, display margins, and vertical separator.

`window-top-line` **&optional** *window* [Function]
: This function returns the Y coordinate of the topmost row of *window*, equivalent to the *top* entry in the list returned by `window-edges`.

`window-left-column` **&optional** *window* [Function]
: This function returns the X coordinate of the leftmost column of *window*, equivalent to the *left* entry in the list returned by `window-edges`.

The following functions can be used to relate a set of frame-relative coordinates to a window:

`window-at` *x y* **&optional** *frame* [Function]
: This function returns the live window at the frame-relative coordinates *x* and *y*, on frame *frame*. If there is no window at that position, the return value is `nil`. If *frame* is omitted or `nil`, it defaults to the selected frame.

`coordinates-in-window-p` *coordinates window* [Function]
: This function checks whether a window *window* occupies the frame-relative coordinates *coordinates*, and if so, which part of the window that is. *window* should be a live window. *coordinates* should be a cons cell of the form `(x . y)`, where *x* and *y* are frame-relative coordinates.

  If there is no window at the specified position, the return value is `nil` . Otherwise, the return value is one of the following:

  `(relx . rely)`
  : The coordinates are inside *window*. The numbers *relx* and *rely* are the equivalent window-relative coordinates for the specified position, counting from 0 at the top left corner of the window.

  `mode-line`
  : The coordinates are in the mode line of *window*.

  `header-line`
  : The coordinates are in the header line of *window*.

  `vertical-line`
  : The coordinates are in the vertical line between *window* and its neighbor to the right. This value occurs only if the window doesn't have a scroll bar; positions in a scroll bar are considered outside the window for these purposes.

  `left-fringe`
  `right-fringe`
  : The coordinates are in the left or right fringe of the window.

  `left-margin`
  `right-margin`
  : The coordinates are in the left or right margin of the window.

`nil` The coordinates are not in any part of *window*.

The function `coordinates-in-window-p` does not require a frame as argument because it always uses the frame that *window* is on.

The following functions return window positions in pixels, rather than character units. Though mostly useful on graphical displays, they can also be called on text terminals, where the screen area of each text character is taken to be "one pixel".

`window-pixel-edges` **&optional** *window* [Function]

This function returns a list of pixel coordinates for the edges of *window*. If *window* is omitted or `nil`, it defaults to the selected window.

The return value has the form `(left top right bottom)`. The list elements are, respectively, the X pixel coordinate of the left window edge, the Y pixel coordinate of the top edge, one more than the X pixel coordinate of the right edge, and one more than the Y pixel coordinate of the bottom edge.

`window-inside-pixel-edges` **&optional** *window* [Function]

This function is like `window-pixel-edges`, except that it returns the pixel coordinates for the edges of the window's text area, rather than the pixel coordinates for the edges of the window itself. *window* must specify a live window.

The following functions return window positions in pixels, relative to the display screen rather than the frame:

`window-absolute-pixel-edges` **&optional** *window* [Function]

This function is like `window-pixel-edges`, except that it returns the edge pixel coordinates relative to the top left corner of the display screen.

`window-inside-absolute-pixel-edges` **&optional** *window* [Function]

This function is like `window-inside-pixel-edges`, except that it returns the edge pixel coordinates relative to the top left corner of the display screen. *window* must specify a live window.

28.23 Window Configurations

A *window configuration* records the entire layout of one frame—all windows, their sizes, which buffers they contain, how those buffers are scrolled, and their values of point and the mark; also their fringes, margins, and scroll bar settings. It also includes the value of `minibuffer-scroll-window`. As a special exception, the window configuration does not record the value of point in the selected window for the current buffer.

You can bring back an entire frame layout by restoring a previously saved window configuration. If you want to record the layout of all frames instead of just one, use a frame configuration instead of a window configuration. See Section 29.12 [Frame Configurations], page 86.

`current-window-configuration` **&optional** *frame* [Function]

This function returns a new object representing *frame*'s current window configuration. The default for *frame* is the selected frame. The variable `window-persistent-parameters` specifies which window parameters (if any) are saved by this function. See Section 28.24 [Window Parameters], page 62.

`set-window-configuration` *configuration* [Function]
This function restores the configuration of windows and buffers as specified by *configuration*, for the frame that *configuration* was created for.

The argument *configuration* must be a value that was previously returned by `current-window-configuration`. The configuration is restored in the frame from which *configuration* was made, whether that frame is selected or not. This always counts as a window size change and triggers execution of the `window-size-change-functions` (see Section 28.25 [Window Hooks], page 64), because `set-window-configuration` doesn't know how to tell whether the new configuration actually differs from the old one.

If the frame from which *configuration* was saved is dead, all this function does is restore the three variables `window-min-height`, `window-min-width` and `minibuffer-scroll-window`. In this case, the function returns `nil`. Otherwise, it returns `t`.

Here is a way of using this function to get the same effect as `save-window-excursion`:

```
(let ((config (current-window-configuration)))
  (unwind-protect
      (progn (split-window-below nil)
             ...)
    (set-window-configuration config)))
```

`save-window-excursion` *forms...* [Macro]
This special form records the window configuration, executes *forms* in sequence, then restores the earlier window configuration. The window configuration includes, for each window, the value of point and the portion of the buffer that is visible. It also includes the choice of selected window. However, it does not include the value of point in the current buffer; use `save-excursion` also, if you wish to preserve that.

Don't use this construct when `save-selected-window` is sufficient.

Exit from `save-window-excursion` always triggers execution of `window-size-change-functions`. (It doesn't know how to tell whether the restored configuration actually differs from the one in effect at the end of the *forms*.)

The return value is the value of the final form in *forms*. For example:

```
(split-window)
     ⇒ #<window 25 on control.texi>
(setq w (selected-window))
     ⇒ #<window 19 on control.texi>
(save-window-excursion
  (delete-other-windows w)
  (switch-to-buffer "foo")
  'do-something)
     ⇒ do-something
     ;; The screen is now split again.
```

`window-configuration-p` *object* [Function]
This function returns `t` if *object* is a window configuration.

`compare-window-configurations` *config1 config2* [Function]
: This function compares two window configurations as regards the structure of windows, but ignores the values of point and mark and the saved scrolling positions—it can return `t` even if those aspects differ.

  The function `equal` can also compare two window configurations; it regards configurations as unequal if they differ in any respect, even a saved point or mark.

`window-configuration-frame` *config* [Function]
: This function returns the frame for which the window configuration *config* was made.

Other primitives to look inside of window configurations would make sense, but are not implemented because we did not need them. See the file '`winner.el`' for some more operations on windows configurations.

The objects returned by `current-window-configuration` die together with the Emacs process. In order to store a window configuration on disk and read it back in another Emacs session, you can use the functions described next. These functions are also useful to clone the state of a frame into an arbitrary live window (`set-window-configuration` effectively clones the windows of a frame into the root window of that very frame only).

`window-state-get` **&optional** *window writable* [Function]
: This function returns the state of *window* as a Lisp object. The argument *window* can be any window and defaults to the root window of the selected frame.

  If the optional argument *writable* is non-`nil`, this means to not use markers for sampling positions like `window-point` or `window-start`. This argument should be non-`nil` when the state will be written to disk and read back in another session.

  Together, the argument *writable* and the variable `window-persistent-parameters` specify which window parameters are saved by this function. See Section 28.24 [Window Parameters], page 62.

The value returned by `window-state-get` can be used in the same session to make a clone of a window in another window. It can be also written to disk and read back in another session. In either case, use the following function to restore the state of the window.

`window-state-put` *state* **&optional** *window ignore* [Function]
: This function puts the window state *state* into *window*. The argument *state* should be the state of a window returned by an earlier invocation of `window-state-get`, see above. The optional argument *window* must specify a live window and defaults to the selected one.

  If the optional argument *ignore* is non-`nil`, it means to ignore minimum window sizes and fixed-size restrictions. If *ignore* is `safe`, this means windows can get as small as one line and/or two columns.

28.24 Window Parameters

This section describes how window parameters can be used to associate additional information with windows.

`window-parameter` *window parameter* [Function]
This function returns *window*'s value for *parameter*. The default for *window* is the selected window. If *window* has no setting for *parameter*, this function returns `nil`.

`window-parameters` **&optional** *window* [Function]
This function returns all parameters of *window* and their values. The default for *window* is the selected window. The return value is either `nil`, or an association list whose elements have the form `(parameter . value)`.

`set-window-parameter` *window parameter value* [Function]
This function sets *window*'s value of *parameter* to *value* and returns *value*. The default for *window* is the selected window.

By default, the functions that save and restore window configurations or the states of windows (see Section 28.23 [Window Configurations], page 60) do not care about window parameters. This means that when you change the value of a parameter within the body of a `save-window-excursion`, the previous value is not restored when that macro exits. It also means that when you restore via `window-state-put` a window state saved earlier by `window-state-get`, all cloned windows have their parameters reset to `nil`. The following variable allows you to override the standard behavior:

`window-persistent-parameters` [Variable]
This variable is an alist specifying which parameters get saved by `current-window-configuration` and `window-state-get`, and subsequently restored by `set-window-configuration` and `window-state-put`. See Section 28.23 [Window Configurations], page 60.

The CAR of each entry of this alist is a symbol specifying the parameter. The CDR should be one of the following:

`nil` This value means the parameter is saved neither by `window-state-get` nor by `current-window-configuration`.

`t` This value specifies that the parameter is saved by `current-window-configuration` and (provided its *writable* argument is `nil`) by `window-state-get`.

`writable` This means that the parameter is saved unconditionally by both `current-window-configuration` and `window-state-get`. This value should not be used for parameters whose values do not have a read syntax. Otherwise, invoking `window-state-put` in another session may fail with an `invalid-read-syntax` error.

Some functions (notably `delete-window`, `delete-other-windows` and `split-window`), may behave specially when their *window* argument has a parameter set. You can override such special behavior by binding the following variable to a non-`nil` value:

`ignore-window-parameters` [Variable]
If this variable is non-`nil`, some standard functions do not process window parameters. The functions currently affected by this are `split-window`, `delete-window`, `delete-other-windows`, and `other-window`.

An application can bind this variable to a non-`nil` value around calls to these functions. If it does so, the application is fully responsible for correctly assigning the parameters of all involved windows when exiting that function.

The following parameters are currently used by the window management code:

`delete-window`
: This parameter affects the execution of `delete-window` (see Section 28.6 [Deleting Windows], page 31).

`delete-other-windows`
: This parameter affects the execution of `delete-other-windows` (see Section 28.6 [Deleting Windows], page 31).

`split-window`
: This parameter affects the execution of `split-window` (see Section 28.5 [Splitting Windows], page 26).

`other-window`
: This parameter affects the execution of `other-window` (see Section 28.8 [Cyclic Window Ordering], page 34).

`no-other-window`
: This parameter marks the window as not selectable by `other-window` (see Section 28.8 [Cyclic Window Ordering], page 34).

`clone-of`
: This parameter specifies the window that this one has been cloned from. It is installed by `window-state-get` (see Section 28.23 [Window Configurations], page 60).

`quit-restore`
: This parameter specifies what to do with a window when the buffer it shows is not needed any more. It is installed by the buffer display functions (see Section 28.11 [Choosing Window], page 39), and consulted by the function `quit-window` (see Section 28.16 [Quitting Windows], page 47).

There are additional parameters `window-atom` and `window-side`; these are reserved and should not be used by applications.

28.25 Hooks for Window Scrolling and Changes

This section describes how a Lisp program can take action whenever a window displays a different part of its buffer or a different buffer. There are three actions that can change this: scrolling the window, switching buffers in the window, and changing the size of the window. The first two actions run `window-scroll-functions`; the last runs `window-size-change-functions`.

`window-scroll-functions` [Variable]
: This variable holds a list of functions that Emacs should call before redisplaying a window with scrolling. Displaying a different buffer in the window also runs these functions.

  This variable is not a normal hook, because each function is called with two arguments: the window, and its new display-start position.

These functions must take care when using `window-end` (see Section 28.18 [Window Start and End], page 49); if you need an up-to-date value, you must use the *update* argument to ensure you get it.

Warning: don't use this feature to alter the way the window is scrolled. It's not designed for that, and such use probably won't work.

`window-size-change-functions` [Variable]

This variable holds a list of functions to be called if the size of any window changes for any reason. The functions are called just once per redisplay, and just once for each frame on which size changes have occurred.

Each function receives the frame as its sole argument. There is no direct way to find out which windows on that frame have changed size, or precisely how. However, if a size-change function records, at each call, the existing windows and their sizes, it can also compare the present sizes and the previous sizes.

Creating or deleting windows counts as a size change, and therefore causes these functions to be called. Changing the frame size also counts, because it changes the sizes of the existing windows.

It is not a good idea to use `save-window-excursion` (see Section 28.23 [Window Configurations], page 60) in these functions, because that always counts as a size change, and it would cause these functions to be called over and over. In most cases, `save-selected-window` (see Section 28.7 [Selecting Windows], page 33) is what you need here.

`window-configuration-change-hook` [Variable]

A normal hook that is run every time you change the window configuration of an existing frame. This includes splitting or deleting windows, changing the sizes of windows, or displaying a different buffer in a window.

The buffer-local part of this hook is run once for each window on the affected frame, with the relevant window selected and its buffer current. The global part is run once for the modified frame, with that frame selected.

In addition, you can use `jit-lock-register` to register a Font Lock fontification function, which will be called whenever parts of a buffer are (re)fontified because a window was scrolled or its size changed. See Section 23.6.4 [Other Font Lock Variables], page 435, vol. 1.

29 Frames

A *frame* is a screen object that contains one or more Emacs windows (see Chapter 28 [Windows], page 18). It is the kind of object called a "window" in the terminology of graphical environments; but we can't call it a "window" here, because Emacs uses that word in a different way. In Emacs Lisp, a *frame object* is a Lisp object that represents a frame on the screen. See Section 2.4.4 [Frame Type], page 25, vol. 1.

A frame initially contains a single main window and/or a minibuffer window; you can subdivide the main window vertically or horizontally into smaller windows. See Section 28.5 [Splitting Windows], page 26.

A *terminal* is a display device capable of displaying one or more Emacs frames. In Emacs Lisp, a *terminal object* is a Lisp object that represents a terminal. See Section 2.4.5 [Terminal Type], page 25, vol. 1.

There are two classes of terminals: *text terminals* and *graphical terminals*. Text terminals are non-graphics-capable displays, including `xterm` and other terminal emulators. On a text terminal, each Emacs frame occupies the terminal's entire screen; although you can create additional frames and switch between them, the terminal only shows one frame at a time. Graphical terminals, on the other hand, are managed by graphical display systems such as the X Window System, which allow Emacs to show multiple frames simultaneously on the same display.

On GNU and Unix systems, you can create additional frames on any available terminal, within a single Emacs session, regardless of whether Emacs was started on a text or graphical terminal. Emacs can display on both graphical and text terminals simultaneously. This comes in handy, for instance, when you connect to the same session from several remote locations. See Section 29.2 [Multiple Terminals], page 67.

`framep` *object* [Function]

This predicate returns a non-`nil` value if *object* is a frame, and `nil` otherwise. For a frame, the value indicates which kind of display the frame uses:

`t`	The frame is displayed on a text terminal.
`x`	The frame is displayed on an X graphical terminal.
`w32`	The frame is displayed on a MS-Windows graphical terminal.
`ns`	The frame is displayed on a GNUstep or Macintosh Cocoa graphical terminal.
`pc`	The frame is displayed on an MS-DOS terminal.

`frame-terminal` **&optional** *frame* [Function]

This function returns the terminal object that displays *frame*. If *frame* is `nil` or unspecified, it defaults to the selected frame.

`terminal-live-p` *object* [Function]

This predicate returns a non-`nil` value if *object* is a terminal that is live (i.e. not deleted), and `nil` otherwise. For live terminals, the return value indicates what kind of frames are displayed on that terminal; the list of possible values is the same as for `framep` above.

29.1 Creating Frames

To create a new frame, call the function `make-frame`.

`make-frame` **&optional** *alist* [Command]
: This function creates and returns a new frame, displaying the current buffer.

  The *alist* argument is an alist that specifies frame parameters for the new frame. See Section 29.3 [Frame Parameters], page 70. If you specify the `terminal` parameter in *alist*, the new frame is created on that terminal. Otherwise, if you specify the `window-system` frame parameter in *alist*, that determines whether the frame should be displayed on a text terminal or a graphical terminal. See Section 38.22 [Window Systems], page 382. If neither is specified, the new frame is created in the same terminal as the selected frame.

  Any parameters not mentioned in *alist* default to the values in the alist `default-frame-alist` (see Section 29.3.2 [Initial Parameters], page 70); parameters not specified there default from the X resources or its equivalent on your operating system (see Section "X Resources" in *The GNU Emacs Manual*). After the frame is created, Emacs applies any parameters listed in `frame-inherited-parameters` (see below) and not present in the argument, taking the values from the frame that was selected when `make-frame` was called.

  This function itself does not make the new frame the selected frame. See Section 29.9 [Input Focus], page 83. The previously selected frame remains selected. On graphical terminals, however, the windowing system may select the new frame for its own reasons.

`before-make-frame-hook` [Variable]
: A normal hook run by `make-frame` before it creates the frame.

`after-make-frame-functions` [Variable]
: An abnormal hook run by `make-frame` after it creates the frame. Each function in `after-make-frame-functions` receives one argument, the frame just created.

`frame-inherited-parameters` [Variable]
: This variable specifies the list of frame parameters that a newly created frame inherits from the currently selected frame. For each parameter (a symbol) that is an element in the list and is not present in the argument to `make-frame`, the function sets the value of that parameter in the created frame to its value in the selected frame.

29.2 Multiple Terminals

Emacs represents each terminal as a *terminal object* data type (see Section 2.4.5 [Terminal Type], page 25, vol. 1). On GNU and Unix systems, Emacs can use multiple terminals simultaneously in each session. On other systems, it can only use a single terminal. Each terminal object has the following attributes:

- The name of the device used by the terminal (e.g. '`:0.0`' or '`/dev/tty`').
- The terminal and keyboard coding systems used on the terminal. See Section 33.9.8 [Terminal I/O Encoding], page 205.

- The kind of display associated with the terminal. This is the symbol returned by the function `terminal-live-p` (i.e. `x`, `t`, `w32`, `ns`, or `pc`). See Chapter 29 [Frames], page 66.
- A list of terminal parameters. See Section 29.4 [Terminal Parameters], page 80.

There is no primitive for creating terminal objects. Emacs creates them as needed, such as when you call `make-frame-on-display` (described below).

`terminal-name` **&optional** *terminal* [Function]
: This function returns the file name of the device used by *terminal*. If *terminal* is omitted or `nil`, it defaults to the selected frame's terminal. *terminal* can also be a frame, meaning that frame's terminal.

`terminal-list` [Function]
: This function returns a list of all live terminal objects.

`get-device-terminal` *device* [Function]
: This function returns a terminal whose device name is given by *device*. If *device* is a string, it can be either the file name of a terminal device, or the name of an X display of the form '*`host:server.screen`*'. If *device* is a frame, this function returns that frame's terminal; `nil` means the selected frame. Finally, if *device* is a terminal object that represents a live terminal, that terminal is returned. The function signals an error if its argument is none of the above.

`delete-terminal` **&optional** *terminal force* [Function]
: This function deletes all frames on *terminal* and frees the resources used by it. It runs the abnormal hook `delete-terminal-functions`, passing *terminal* as the argument to each function.

  If *terminal* is omitted or `nil`, it defaults to the selected frame's terminal. *terminal* can also be a frame, meaning that frame's terminal.

  Normally, this function signals an error if you attempt to delete the sole active terminal, but if *force* is non-`nil`, you are allowed to do so. Emacs automatically calls this function when the last frame on a terminal is deleted (see Section 29.6 [Deleting Frames], page 82).

`delete-terminal-functions` [Variable]
: An abnormal hook run by `delete-terminal`. Each function receives one argument, the *terminal* argument passed to `delete-terminal`. Due to technical details, the functions may be called either just before the terminal is deleted, or just afterwards.

A few Lisp variables are *terminal-local*; that is, they have a separate binding for each terminal. The binding in effect at any time is the one for the terminal that the currently selected frame belongs to. These variables include `default-minibuffer-frame`, `defining-kbd-macro`, `last-kbd-macro`, and `system-key-alist`. They are always terminal-local, and can never be buffer-local (see Section 11.10 [Buffer-Local Variables], page 150, vol. 1).

On GNU and Unix systems, each X display is a separate graphical terminal. When Emacs is started from within the X window system, it uses the X display specified by the `DISPLAY` environment variable, or by the '`--display`' option (see Section "Initial Options" in *The GNU Emacs Manual*). Emacs can connect to other X displays via the command

`make-frame-on-display`. Each X display has its own selected frame and its own minibuffer windows; however, only one of those frames is "*the* selected frame" at any given moment (see Section 29.9 [Input Focus], page 83). Emacs can even connect to other text terminals, by interacting with the `emacsclient` program. See Section "Emacs Server" in *The GNU Emacs Manual*.

A single X server can handle more than one display. Each X display has a three-part name, '*`host:server.screen`*'. The first two parts, *host* and *server*, identify the X server; the third part, *screen*, identifies a screen number on that X server. When you use two or more screens belonging to one server, Emacs knows by the similarity in their names that they share a single keyboard.

On some "multi-monitor" setups, a single X display outputs to more than one physical monitor. Currently, there is no way for Emacs to distinguish between the different physical monitors.

`make-frame-on-display` *display* **&optional** *parameters* [Command]
: This function creates and returns a new frame on *display*, taking the other frame parameters from the alist *parameters*. *display* should be the name of an X display (a string).

  Before creating the frame, this function ensures that Emacs is "set up" to display graphics. For instance, if Emacs has not processed X resources (e.g. if it was started on a text terminal), it does so at this time. In all other respects, this function behaves like `make-frame` (see Section 29.1 [Creating Frames], page 67).

`x-display-list` [Function]
: This function returns a list that indicates which X displays Emacs has a connection to. The elements of the list are strings, and each one is a display name.

`x-open-connection` *display* **&optional** *xrm-string must-succeed* [Function]
: This function opens a connection to the X display *display*, without creating a frame on that display. Normally, Emacs Lisp programs need not call this function, as `make-frame-on-display` calls it automatically. The only reason for calling it is to check whether communication can be established with a given X display.

  The optional argument *xrm-string*, if not `nil`, is a string of resource names and values, in the same format used in the '`.Xresources`' file. See Section "X Resources" in *The GNU Emacs Manual*. These values apply to all Emacs frames created on this display, overriding the resource values recorded in the X server. Here's an example of what this string might look like:

  ```
  "*BorderWidth: 3\n*InternalBorder: 2\n"
  ```

  If *must-succeed* is non-`nil`, failure to open the connection terminates Emacs. Otherwise, it is an ordinary Lisp error.

`x-close-connection` *display* [Function]
: This function closes the connection to display *display*. Before you can do this, you must first delete all the frames that were open on that display (see Section 29.6 [Deleting Frames], page 82).

29.3 Frame Parameters

A frame has many parameters that control its appearance and behavior. Just what parameters a frame has depends on what display mechanism it uses.

Frame parameters exist mostly for the sake of graphical displays. Most frame parameters have no effect when applied to a frame on a text terminal; only the `height`, `width`, `name`, `title`, `menu-bar-lines`, `buffer-list` and `buffer-predicate` parameters do something special. If the terminal supports colors, the parameters `foreground-color`, `background-color`, `background-mode` and `display-type` are also meaningful. If the terminal supports frame transparency, the parameter `alpha` is also meaningful.

29.3.1 Access to Frame Parameters

These functions let you read and change the parameter values of a frame.

`frame-parameter` *frame parameter* [Function]
: This function returns the value of the parameter *parameter* (a symbol) of *frame*. If *frame* is `nil`, it returns the selected frame's parameter. If *frame* has no setting for *parameter*, this function returns `nil`.

`frame-parameters` **&optional** *frame* [Function]
: The function `frame-parameters` returns an alist listing all the parameters of *frame* and their values. If *frame* is `nil` or omitted, this returns the selected frame's parameters

`modify-frame-parameters` *frame alist* [Function]
: This function alters the parameters of frame *frame* based on the elements of *alist*. Each element of *alist* has the form `(parm . value)`, where *parm* is a symbol naming a parameter. If you don't mention a parameter in *alist*, its value doesn't change. If *frame* is `nil`, it defaults to the selected frame.

`set-frame-parameter` *frame parm value* [Function]
: This function sets the frame parameter *parm* to the specified *value*. If *frame* is `nil`, it defaults to the selected frame.

`modify-all-frames-parameters` *alist* [Function]
: This function alters the frame parameters of all existing frames according to *alist*, then modifies `default-frame-alist` (and, if necessary, `initial-frame-alist`) to apply the same parameter values to frames that will be created henceforth.

29.3.2 Initial Frame Parameters

You can specify the parameters for the initial startup frame by setting `initial-frame-alist` in your init file (see Section 39.1.2 [Init File], page 389).

`initial-frame-alist` [User Option]
: This variable's value is an alist of parameter values used when creating the initial frame. You can set this variable to specify the appearance of the initial frame without altering subsequent frames. Each element has the form:

```
(parameter . value)
```

Emacs creates the initial frame before it reads your init file. After reading that file, Emacs checks `initial-frame-alist`, and applies the parameter settings in the altered value to the already created initial frame.

If these settings affect the frame geometry and appearance, you'll see the frame appear with the wrong ones and then change to the specified ones. If that bothers you, you can specify the same geometry and appearance with X resources; those do take effect before the frame is created. See Section "X Resources" in *The GNU Emacs Manual*.

X resource settings typically apply to all frames. If you want to specify some X resources solely for the sake of the initial frame, and you don't want them to apply to subsequent frames, here's how to achieve this. Specify parameters in `default-frame-alist` to override the X resources for subsequent frames; then, to prevent these from affecting the initial frame, specify the same parameters in `initial-frame-alist` with values that match the X resources.

If these parameters specify a separate *minibuffer-only frame* with `(minibuffer . nil)`, and you have not created one, Emacs creates one for you.

`minibuffer-frame-alist` [User Option]

This variable's value is an alist of parameter values used when creating an initial minibuffer-only frame. This is the minibuffer-only frame that Emacs creates if `initial-frame-alist` specifies a frame with no minibuffer.

`default-frame-alist` [User Option]

This is an alist specifying default values of frame parameters for all Emacs frames—the first frame, and subsequent frames. When using the X Window System, you can get the same results by means of X resources in many cases.

Setting this variable does not affect existing frames.

Functions that display a buffer in a separate frame can override the default parameters by supplying their own parameters. See [Definition of special-display-frame-alist], page 44.

If you invoke Emacs with command-line options that specify frame appearance, those options take effect by adding elements to either `initial-frame-alist` or `default-frame-alist`. Options which affect just the initial frame, such as '`-geometry`' and '`--maximized`', add to `initial-frame-alist`; the others add to `default-frame-alist`. see Section "Command Line Arguments for Emacs Invocation" in *The GNU Emacs Manual*.

29.3.3 Window Frame Parameters

Just what parameters a frame has depends on what display mechanism it uses. This section describes the parameters that have special meanings on some or all kinds of terminals. Of these, `name`, `title`, `height`, `width`, `buffer-list` and `buffer-predicate` provide meaningful information in terminal frames, and `tty-color-mode` is meaningful only for frames on text terminals.

29.3.3.1 Basic Parameters

These frame parameters give the most basic information about the frame. `title` and `name` are meaningful on all terminals.

`display` The display on which to open this frame. It should be a string of the form `"host:dpy.screen"`, just like the `DISPLAY` environment variable.

`display-type`
This parameter describes the range of possible colors that can be used in this frame. Its value is `color`, `grayscale` or `mono`.

`title` If a frame has a non-`nil` title, it appears in the window system's title bar at the top of the frame, and also in the mode line of windows in that frame if `mode-line-frame-identification` uses '`%F`' (see Section 23.4.5 [%-Constructs], page 424, vol. 1). This is normally the case when Emacs is not using a window system, and can only display one frame at a time. See Section 29.5 [Frame Titles], page 81.

`name` The name of the frame. The frame name serves as a default for the frame title, if the `title` parameter is unspecified or `nil`. If you don't specify a name, Emacs sets the frame name automatically (see Section 29.5 [Frame Titles], page 81).

If you specify the frame name explicitly when you create the frame, the name is also used (instead of the name of the Emacs executable) when looking up X resources for the frame.

`explicit-name`
If the frame name was specified explicitly when the frame was created, this parameter will be that name. If the frame wasn't explicitly named, this parameter will be `nil`.

29.3.3.2 Position Parameters

Position parameters' values are normally measured in pixels, but on text terminals they count characters or lines instead.

`left` The position, in pixels, of the left (or right) edge of the frame with respect to the left (or right) edge of the screen. The value may be:

an integer A positive integer relates the left edge of the frame to the left edge of the screen. A negative integer relates the right frame edge to the right screen edge.

`(+ pos)` This specifies the position of the left frame edge relative to the left screen edge. The integer *pos* may be positive or negative; a negative value specifies a position outside the screen.

`(- pos)` This specifies the position of the right frame edge relative to the right screen edge. The integer *pos* may be positive or negative; a negative value specifies a position outside the screen.

Some window managers ignore program-specified positions. If you want to be sure the position you specify is not ignored, specify a non-`nil` value for the `user-position` parameter as well.

`top` The screen position of the top (or bottom) edge, in pixels, with respect to the top (or bottom) edge of the screen. It works just like `left`, except vertically instead of horizontally.

`icon-left`
: The screen position of the left edge of the frame's icon, in pixels, counting from the left edge of the screen. This takes effect when the frame is iconified, if the window manager supports this feature. If you specify a value for this parameter, then you must also specify a value for `icon-top` and vice versa.

`icon-top`
: The screen position of the top edge of the frame's icon, in pixels, counting from the top edge of the screen. This takes effect when the frame is iconified, if the window manager supports this feature.

`user-position`
: When you create a frame and specify its screen position with the `left` and `top` parameters, use this parameter to say whether the specified position was user-specified (explicitly requested in some way by a human user) or merely program-specified (chosen by a program). A non-`nil` value says the position was user-specified.

  Window managers generally heed user-specified positions, and some heed program-specified positions too. But many ignore program-specified positions, placing the window in a default fashion or letting the user place it with the mouse. Some window managers, including `twm`, let the user specify whether to obey program-specified positions or ignore them.

  When you call `make-frame`, you should specify a non-`nil` value for this parameter if the values of the `left` and `top` parameters represent the user's stated preference; otherwise, use `nil`.

29.3.3.3 Size Parameters

Frame parameters specify frame sizes in character units. On graphical displays, the `default` face determines the actual pixel sizes of these character units (see Section 38.12.2 [Face Attributes], page 327).

`height`
: The height of the frame contents, in characters. (To get the height in pixels, call `frame-pixel-height`; see Section 29.3.4 [Size and Position], page 79.)

`width`
: The width of the frame contents, in characters. (To get the width in pixels, call `frame-pixel-width`; see Section 29.3.4 [Size and Position], page 79.)

`user-size`
: This does for the size parameters `height` and `width` what the `user-position` parameter (see Section 29.3.3.2 [Position Parameters], page 72) does for the position parameters `top` and `left`.

`fullscreen`
: Specify that width, height or both shall be maximized. The value `fullwidth` specifies that width shall be as wide as possible. The value `fullheight` specifies that height shall be as tall as possible. The value `fullboth` specifies that both the width and the height shall be set to the size of the screen. The value `maximized` specifies that the frame shall be maximized. The difference between `maximized` and `fullboth` is that the former still has window manager decorations while the latter really covers the whole screen.

29.3.3.4 Layout Parameters

These frame parameters enable or disable various parts of the frame, or control their sizes.

`border-width`
: The width in pixels of the frame's border.

`internal-border-width`
: The distance in pixels between text (or fringe) and the frame's border.

`vertical-scroll-bars`
: Whether the frame has scroll bars for vertical scrolling, and which side of the frame they should be on. The possible values are `left`, `right`, and `nil` for no scroll bars.

`scroll-bar-width`
: The width of vertical scroll bars, in pixels, or `nil` meaning to use the default width.

`left-fringe`
`right-fringe`
: The default width of the left and right fringes of windows in this frame (see Section 38.13 [Fringes], page 344). If either of these is zero, that effectively removes the corresponding fringe.

  When you use `frame-parameter` to query the value of either of these two frame parameters, the return value is always an integer. When using `set-frame-parameter`, passing a `nil` value imposes an actual default value of 8 pixels.

  The combined fringe widths must add up to an integral number of columns, so the actual default fringe widths for the frame, as reported by `frame-parameter`, may be larger than what you specify. Any extra width is distributed evenly between the left and right fringe. However, you can force one fringe or the other to a precise width by specifying that width as a negative integer. If both widths are negative, only the left fringe gets the specified width.

`menu-bar-lines`
: The number of lines to allocate at the top of the frame for a menu bar. The default is 1 if Menu Bar mode is enabled, and 0 otherwise. See Section "Menu Bars" in *The GNU Emacs Manual*.

`tool-bar-lines`
: The number of lines to use for the tool bar. The default is 1 if Tool Bar mode is enabled, and 0 otherwise. See Section "Tool Bars" in *The GNU Emacs Manual*.

`tool-bar-position`
: The position of the tool bar. Currently only for the GTK tool bar. Value can be one of `top`, `bottom` `left`, `right`. The default is `top`.

`line-spacing`
: Additional space to leave below each text line, in pixels (a positive integer). See Section 38.11 [Line Height], page 324, for more information.

29.3.3.5 Buffer Parameters

These frame parameters, meaningful on all kinds of terminals, deal with which buffers have been, or should, be displayed in the frame.

`minibuffer`
: Whether this frame has its own minibuffer. The value `t` means yes, `nil` means no, `only` means this frame is just a minibuffer. If the value is a minibuffer window (in some other frame), the frame uses that minibuffer.

  This frame parameter takes effect when the frame is created, and can not be changed afterwards.

`buffer-predicate`
: The buffer-predicate function for this frame. The function `other-buffer` uses this predicate (from the selected frame) to decide which buffers it should consider, if the predicate is not `nil`. It calls the predicate with one argument, a buffer, once for each buffer; if the predicate returns a non-`nil` value, it considers that buffer.

`buffer-list`
: A list of buffers that have been selected in this frame, ordered most-recently-selected first.

`unsplittable`
: If non-`nil`, this frame's window is never split automatically.

29.3.3.6 Window Management Parameters

The following frame parameters control various aspects of the frame's interaction with the window manager. They have no effect on text terminals.

`visibility`
: The state of visibility of the frame. There are three possibilities: `nil` for invisible, `t` for visible, and `icon` for iconified. See Section 29.10 [Visibility of Frames], page 85.

`auto-raise`
: If non-`nil`, Emacs automatically raises the frame when it is selected. Some window managers do not allow this.

`auto-lower`
: If non-`nil`, Emacs automatically lowers the frame when it is deselected. Some window managers do not allow this.

`icon-type`
: The type of icon to use for this frame. If the value is a string, that specifies a file containing a bitmap to use; `nil` specifies no icon (in which case the window manager decides what to show); any other non-`nil` value specifies the default Emacs icon.

`icon-name`
: The name to use in the icon for this frame, when and if the icon appears. If this is `nil`, the frame's title is used.

`window-id`
: The ID number which the graphical display uses for this frame. Emacs assigns this parameter when the frame is created; changing the parameter has no effect on the actual ID number.

`outer-window-id`
: The ID number of the outermost window-system window in which the frame exists. As with `window-id`, changing this parameter has no actual effect.

`wait-for-wm`
: If non-`nil`, tell Xt to wait for the window manager to confirm geometry changes. Some window managers, including versions of Fvwm2 and KDE, fail to confirm, so Xt hangs. Set this to `nil` to prevent hanging with those window managers.

`sticky`
: If non-`nil`, the frame is visible on all virtual desktops on systems with virtual desktops.

29.3.3.7 Cursor Parameters

This frame parameter controls the way the cursor looks.

`cursor-type`
: How to display the cursor. Legitimate values are:

  `box` — Display a filled box. (This is the default.)

  `hollow` — Display a hollow box.

  `nil` — Don't display a cursor.

  `bar` — Display a vertical bar between characters.

  `(bar . width)` — Display a vertical bar *width* pixels wide between characters.

  `hbar` — Display a horizontal bar.

  `(hbar . height)` — Display a horizontal bar *height* pixels high.

The `cursor-type` frame parameter may be overridden by the variables `cursor-type` and `cursor-in-non-selected-windows`:

`cursor-type` [Variable]
: This buffer-local variable controls how the cursor looks in a selected window showing the buffer. If its value is `t`, that means to use the cursor specified by the `cursor-type` frame parameter. Otherwise, the value should be one of the cursor types listed above, and it overrides the `cursor-type` frame parameter.

`cursor-in-non-selected-windows` [User Option]
: This buffer-local variable controls how the cursor looks in a window that is not selected. It supports the same values as the `cursor-type` frame parameter; also, `nil` means don't display a cursor in nonselected windows, and `t` (the default) means use a standard modification of the usual cursor type (solid box becomes hollow box, and bar becomes a narrower bar).

`blink-cursor-alist` [User Option]

This variable specifies how to blink the cursor. Each element has the form `(`*`on-state . off-state`*`)`. Whenever the cursor type equals *on-state* (comparing using `equal`), the corresponding *off-state* specifies what the cursor looks like when it blinks "off". Both *on-state* and *off-state* should be suitable values for the `cursor-type` frame parameter.

There are various defaults for how to blink each type of cursor, if the type is not mentioned as an *on-state* here. Changes in this variable do not take effect immediately, only when you specify the `cursor-type` frame parameter.

29.3.3.8 Font and Color Parameters

These frame parameters control the use of fonts and colors.

`font-backend`

A list of symbols, specifying the *font backends* to use for drawing fonts in the frame, in order of priority. On X, there are currently two available font backends: `x` (the X core font driver) and `xft` (the Xft font driver). On Windows, there are currently two available font backends: `gdi` and `uniscribe` (see Section "Windows Fonts" in *The GNU Emacs Manual*). On other systems, there is only one available font backend, so it does not make sense to modify this frame parameter.

`background-mode`

This parameter is either `dark` or `light`, according to whether the background color is a light one or a dark one.

`tty-color-mode`

This parameter overrides the terminal's color support as given by the system's terminal capabilities database in that this parameter's value specifies the color mode to use on a text terminal. The value can be either a symbol or a number. A number specifies the number of colors to use (and, indirectly, what commands to issue to produce each color). For example, `(tty-color-mode . 8)` specifies use of the ANSI escape sequences for 8 standard text colors. A value of -1 turns off color support.

If the parameter's value is a symbol, it specifies a number through the value of `tty-color-mode-alist`, and the associated number is used instead.

`screen-gamma`

If this is a number, Emacs performs "gamma correction" which adjusts the brightness of all colors. The value should be the screen gamma of your display, a floating point number.

Usual PC monitors have a screen gamma of 2.2, so color values in Emacs, and in X windows generally, are calibrated to display properly on a monitor with that gamma value. If you specify 2.2 for `screen-gamma`, that means no correction is needed. Other values request correction, designed to make the corrected colors appear on your screen the way they would have appeared without correction on an ordinary monitor with a gamma value of 2.2.

If your monitor displays colors too light, you should specify a `screen-gamma` value smaller than 2.2. This requests correction that makes colors darker. A screen gamma value of 1.5 may give good results for LCD color displays.

`alpha` This parameter specifies the opacity of the frame, on graphical displays that support variable opacity. It should be an integer between 0 and 100, where 0 means completely transparent and 100 means completely opaque. It can also have a `nil` value, which tells Emacs not to set the frame opacity (leaving it to the window manager).

To prevent the frame from disappearing completely from view, the variable `frame-alpha-lower-limit` defines a lower opacity limit. If the value of the frame parameter is less than the value of this variable, Emacs uses the latter. By default, `frame-alpha-lower-limit` is 20.

The `alpha` frame parameter can also be a cons cell `('active' . 'inactive')`, where '`active`' is the opacity of the frame when it is selected, and '`inactive`' is the opacity when it is not selected.

The following frame parameters are semi-obsolete in that they are automatically equivalent to particular face attributes of particular faces (see Section "Standard Faces" in *The Emacs Manual*):

`font` The name of the font for displaying text in the frame. This is a string, either a valid font name for your system or the name of an Emacs fontset (see Section 38.12.11 [Fontsets], page 339). It is equivalent to the `font` attribute of the `default` face.

`foreground-color`
The color to use for the image of a character. It is equivalent to the `:foreground` attribute of the `default` face.

`background-color`
The color to use for the background of characters. It is equivalent to the `:background` attribute of the `default` face.

`mouse-color`
The color for the mouse pointer. It is equivalent to the `:background` attribute of the `mouse` face.

`cursor-color`
The color for the cursor that shows point. It is equivalent to the `:background` attribute of the `cursor` face.

`border-color`
The color for the border of the frame. It is equivalent to the `:background` attribute of the `border` face.

`scroll-bar-foreground`
If non-`nil`, the color for the foreground of scroll bars. It is equivalent to the `:foreground` attribute of the `scroll-bar` face.

`scroll-bar-background`
If non-`nil`, the color for the background of scroll bars. It is equivalent to the `:background` attribute of the `scroll-bar` face.

29.3.4 Frame Size And Position

You can read or change the size and position of a frame using the frame parameters `left`, `top`, `height`, and `width`. Whatever geometry parameters you don't specify are chosen by the window manager in its usual fashion.

Here are some special features for working with sizes and positions. (For the precise meaning of "selected frame" used by these functions, see Section 29.9 [Input Focus], page 83.)

`set-frame-position` *frame left top* [Function]

This function sets the position of the top left corner of *frame* to *left* and *top*. These arguments are measured in pixels, and normally count from the top left corner of the screen.

Negative parameter values position the bottom edge of the window up from the bottom edge of the screen, or the right window edge to the left of the right edge of the screen. It would probably be better if the values were always counted from the left and top, so that negative arguments would position the frame partly off the top or left edge of the screen, but it seems inadvisable to change that now.

`frame-height` **&optional** *frame* [Function]
`frame-width` **&optional** *frame* [Function]

These functions return the height and width of *frame*, measured in lines and columns. If you don't supply *frame*, they use the selected frame.

`frame-pixel-height` **&optional** *frame* [Function]
`frame-pixel-width` **&optional** *frame* [Function]

These functions return the height and width of the main display area of *frame*, measured in pixels. If you don't supply *frame*, they use the selected frame. For a text terminal, the results are in characters rather than pixels.

These values include the internal borders, and windows' scroll bars and fringes (which belong to individual windows, not to the frame itself). The exact value of the heights depends on the window-system and toolkit in use. With GTK+, the height does not include any tool bar or menu bar. With the Motif or Lucid toolkits, it includes the tool bar but not the menu bar. In a graphical version with no toolkit, it includes both the tool bar and menu bar. For a text terminal, the result includes the menu bar.

`frame-char-height` **&optional** *frame* [Function]
`frame-char-width` **&optional** *frame* [Function]

These functions return the height and width of a character in *frame*, measured in pixels. The values depend on the choice of font. If you don't supply *frame*, these functions use the selected frame.

`set-frame-size` *frame cols rows* [Function]

This function sets the size of *frame*, measured in characters; *cols* and *rows* specify the new width and height.

To set the size based on values measured in pixels, use `frame-char-height` and `frame-char-width` to convert them to units of characters.

`set-frame-height` *frame lines* **&optional** *pretend* [Function]
This function resizes *frame* to a height of *lines* lines. The sizes of existing windows in *frame* are altered proportionally to fit.

If *pretend* is non-`nil`, then Emacs displays *lines* lines of output in *frame*, but does not change its value for the actual height of the frame. This is only useful on text terminals. Using a smaller height than the terminal actually implements may be useful to reproduce behavior observed on a smaller screen, or if the terminal malfunctions when using its whole screen. Setting the frame height "for real" does not always work, because knowing the correct actual size may be necessary for correct cursor positioning on text terminals.

`set-frame-width` *frame width* **&optional** *pretend* [Function]
This function sets the width of *frame*, measured in characters. The argument *pretend* has the same meaning as in `set-frame-height`.

29.3.5 Geometry

Here's how to examine the data in an X-style window geometry specification:

`x-parse-geometry` *geom* [Function]
The function `x-parse-geometry` converts a standard X window geometry string to an alist that you can use as part of the argument to `make-frame`.

The alist describes which parameters were specified in *geom*, and gives the values specified for them. Each element looks like (*parameter* . *value*). The possible *parameter* values are `left`, `top`, `width`, and `height`.

For the size parameters, the value must be an integer. The position parameter names `left` and `top` are not totally accurate, because some values indicate the position of the right or bottom edges instead. The *value* possibilities for the position parameters are: an integer, a list (`+` *pos*), or a list (`-` *pos*); as previously described (see Section 29.3.3.2 [Position Parameters], page 72).

Here is an example:

```
(x-parse-geometry "35x70+0-0")
     ⇒ ((height . 70) (width . 35)
         (top - 0) (left . 0))
```

29.4 Terminal Parameters

Each terminal has a list of associated parameters. These *terminal parameters* are mostly a convenient way of storage for terminal-local variables, but some terminal parameters have a special meaning.

This section describes functions to read and change the parameter values of a terminal. They all accept as their argument either a terminal or a frame; the latter means use that frame's terminal. An argument of `nil` means the selected frame's terminal.

`terminal-parameters` **&optional** *terminal* [Function]
This function returns an alist listing all the parameters of *terminal* and their values.

`terminal-parameter` *terminal parameter* [Function]
: This function returns the value of the parameter *parameter* (a symbol) of *terminal*. If *terminal* has no setting for *parameter*, this function returns `nil`.

`set-terminal-parameter` *terminal parameter value* [Function]
: This function sets the parameter *parm* of *terminal* to the specified *value*, and returns the previous value of that parameter.

Here's a list of a few terminal parameters that have a special meaning:

`background-mode`
: The classification of the terminal's background color, either `light` or `dark`.

`normal-erase-is-backspace`
: Value is either 1 or 0, depending on whether `normal-erase-is-backspace-mode` is turned on or off on this terminal. See Section "DEL Does Not Delete" in *The Emacs Manual*.

`terminal-initted`
: After the terminal is initialized, this is set to the terminal-specific initialization function.

29.5 Frame Titles

Every frame has a `name` parameter; this serves as the default for the frame title which window systems typically display at the top of the frame. You can specify a name explicitly by setting the `name` frame property.

Normally you don't specify the name explicitly, and Emacs computes the frame name automatically based on a template stored in the variable `frame-title-format`. Emacs recomputes the name each time the frame is redisplayed.

`frame-title-format` [Variable]
: This variable specifies how to compute a name for a frame when you have not explicitly specified one. The variable's value is actually a mode line construct, just like `mode-line-format`, except that the '`%c`' and '`%l`' constructs are ignored. See Section 23.4.2 [Mode Line Data], page 419, vol. 1.

`icon-title-format` [Variable]
: This variable specifies how to compute the name for an iconified frame, when you have not explicitly specified the frame title. This title appears in the icon itself.

`multiple-frames` [Variable]
: This variable is set automatically by Emacs. Its value is `t` when there are two or more frames (not counting minibuffer-only frames or invisible frames). The default value of `frame-title-format` uses `multiple-frames` so as to put the buffer name in the frame title only when there is more than one frame.

  The value of this variable is not guaranteed to be accurate except while processing `frame-title-format` or `icon-title-format`.

29.6 Deleting Frames

A *live frame* is one that has not been deleted. When a frame is deleted, it is removed from its terminal display, although it may continue to exist as a Lisp object until there are no more references to it.

`delete-frame` **&optional** *frame force* [Command]
: This function deletes the frame *frame*. Unless *frame* is a tooltip, it first runs the hook `delete-frame-functions` (each function gets one argument, *frame*). By default, *frame* is the selected frame.

  A frame cannot be deleted if its minibuffer is used by other frames. Normally, you cannot delete a frame if all other frames are invisible, but if *force* is non-`nil`, then you are allowed to do so.

`frame-live-p` *frame* [Function]
: The function `frame-live-p` returns non-`nil` if the frame *frame* has not been deleted. The possible non-`nil` return values are like those of `framep`. See Chapter 29 [Frames], page 66.

Some window managers provide a command to delete a window. These work by sending a special message to the program that operates the window. When Emacs gets one of these commands, it generates a `delete-frame` event, whose normal definition is a command that calls the function `delete-frame`. See Section 21.7.10 [Misc Events], page 334, vol. 1.

29.7 Finding All Frames

`frame-list` [Function]
: This function returns a list of all the live frames, i.e. those that have not been deleted. It is analogous to `buffer-list` for buffers, and includes frames on all terminals. The list that you get is newly created, so modifying the list doesn't have any effect on the internals of Emacs.

`visible-frame-list` [Function]
: This function returns a list of just the currently visible frames. See Section 29.10 [Visibility of Frames], page 85. Frames on text terminals always count as "visible", even though only the selected one is actually displayed.

`next-frame` **&optional** *frame minibuf* [Function]
: This function lets you cycle conveniently through all the frames on the current display from an arbitrary starting point. It returns the "next" frame after *frame* in the cycle. If *frame* is omitted or `nil`, it defaults to the selected frame (see Section 29.9 [Input Focus], page 83).

  The second argument, *minibuf*, says which frames to consider:

  `nil` Exclude minibuffer-only frames.

  `visible` Consider all visible frames.

  0 Consider all visible or iconified frames.

  a window Consider only the frames using that particular window as their minibuffer.

anything else
: Consider all frames.

`previous-frame` **&optional** *frame minibuf* [Function]
: Like `next-frame`, but cycles through all frames in the opposite direction.

See also `next-window` and `previous-window`, in Section 28.8 [Cyclic Window Ordering], page 34.

29.8 Minibuffers and Frames

Normally, each frame has its own minibuffer window at the bottom, which is used whenever that frame is selected. If the frame has a minibuffer, you can get it with `minibuffer-window` (see [Definition of minibuffer-window], page 312, vol. 1).

However, you can also create a frame with no minibuffer. Such a frame must use the minibuffer window of some other frame. When you create the frame, you can explicitly specify the minibuffer window to use (in some other frame). If you don't, then the minibuffer is found in the frame which is the value of the variable `default-minibuffer-frame`. Its value should be a frame that does have a minibuffer.

If you use a minibuffer-only frame, you might want that frame to raise when you enter the minibuffer. If so, set the variable `minibuffer-auto-raise` to `t`. See Section 29.11 [Raising and Lowering], page 86.

`default-minibuffer-frame` [Variable]
: This variable specifies the frame to use for the minibuffer window, by default. It does not affect existing frames. It is always local to the current terminal and cannot be buffer-local. See Section 29.2 [Multiple Terminals], page 67.

29.9 Input Focus

At any time, one frame in Emacs is the *selected frame*. The selected window always resides on the selected frame.

When Emacs displays its frames on several terminals (see Section 29.2 [Multiple Terminals], page 67), each terminal has its own selected frame. But only one of these is "*the* selected frame": it's the frame that belongs to the terminal from which the most recent input came. That is, when Emacs runs a command that came from a certain terminal, the selected frame is the one of that terminal. Since Emacs runs only a single command at any given time, it needs to consider only one selected frame at a time; this frame is what we call *the selected frame* in this manual. The display on which the selected frame is shown is the *selected frame's display*.

`selected-frame` [Function]
: This function returns the selected frame.

Some window systems and window managers direct keyboard input to the window object that the mouse is in; others require explicit clicks or commands to *shift the focus* to various window objects. Either way, Emacs automatically keeps track of which frame has the focus. To explicitly switch to a different frame from a Lisp function, call `select-frame-set-input-focus`.

Lisp programs can also switch frames "temporarily" by calling the function `select-frame`. This does not alter the window system's concept of focus; rather, it escapes from the window manager's control until that control is somehow reasserted.

When using a text terminal, only one frame can be displayed at a time on the terminal, so after a call to `select-frame`, the next redisplay actually displays the newly selected frame. This frame remains selected until a subsequent call to `select-frame`. Each frame on a text terminal has a number which appears in the mode line before the buffer name (see Section 23.4.4 [Mode Line Variables], page 422, vol. 1).

`select-frame-set-input-focus` *frame* **&optional** *norecord* [Function]
: This function selects *frame*, raises it (should it happen to be obscured by other frames) and tries to give it the X server's focus. On a text terminal, the next redisplay displays the new frame on the entire terminal screen. The optional argument *norecord* has the same meaning as for `select-frame` (see below). The return value of this function is not significant.

`select-frame` *frame* **&optional** *norecord* [Command]
: This function selects frame *frame*, temporarily disregarding the focus of the X server if any. The selection of *frame* lasts until the next time the user does something to select a different frame, or until the next time this function is called. (If you are using a window system, the previously selected frame may be restored as the selected frame after return to the command loop, because it still may have the window system's input focus.)

  The specified *frame* becomes the selected frame, and its terminal becomes the selected terminal. This function then calls `select-window` as a subroutine, passing the window selected within *frame* as its first argument and *norecord* as its second argument (hence, if *norecord* is non-`nil`, this avoids changing the order of recently selected windows nor the buffer list). See Section 28.7 [Selecting Windows], page 33.

  This function returns *frame*, or `nil` if *frame* has been deleted.

  In general, you should never use `select-frame` in a way that could switch to a different terminal without switching back when you're done.

Emacs cooperates with the window system by arranging to select frames as the server and window manager request. It does so by generating a special kind of input event, called a *focus* event, when appropriate. The command loop handles a focus event by calling `handle-switch-frame`. See Section 21.7.9 [Focus Events], page 334, vol. 1.

`handle-switch-frame` *frame* [Command]
: This function handles a focus event by selecting frame *frame*.

  Focus events normally do their job by invoking this command. Don't call it for any other reason.

`redirect-frame-focus` *frame* **&optional** *focus-frame* [Function]
: This function redirects focus from *frame* to *focus-frame*. This means that *focus-frame* will receive subsequent keystrokes and events intended for *frame*. After such an event, the value of `last-event-frame` will be *focus-frame*. Also, switch-frame events specifying *frame* will instead select *focus-frame*.

If *focus-frame* is omitted or `nil`, that cancels any existing redirection for *frame*, which therefore once again receives its own events.

One use of focus redirection is for frames that don't have minibuffers. These frames use minibuffers on other frames. Activating a minibuffer on another frame redirects focus to that frame. This puts the focus on the minibuffer's frame, where it belongs, even though the mouse remains in the frame that activated the minibuffer.

Selecting a frame can also change focus redirections. Selecting frame `bar`, when `foo` had been selected, changes any redirections pointing to `foo` so that they point to `bar` instead. This allows focus redirection to work properly when the user switches from one frame to another using `select-window`.

This means that a frame whose focus is redirected to itself is treated differently from a frame whose focus is not redirected. `select-frame` affects the former but not the latter.

The redirection lasts until `redirect-frame-focus` is called to change it.

`focus-follows-mouse` [User Option]

This option is how you inform Emacs whether the window manager transfers focus when the user moves the mouse. Non-`nil` says that it does. When this is so, the command `other-frame` moves the mouse to a position consistent with the new selected frame.

29.10 Visibility of Frames

A frame on a graphical display may be *visible*, *invisible*, or *iconified*. If it is visible, its contents are displayed in the usual manner. If it is iconified, its contents are not displayed, but there is a little icon somewhere to bring the frame back into view (some window managers refer to this state as *minimized* rather than *iconified*, but from Emacs' point of view they are the same thing). If a frame is invisible, it is not displayed at all.

Visibility is meaningless on text terminals, since only the selected one is actually displayed in any case.

`frame-visible-p` *frame* [Function]

This function returns the visibility status of frame *frame*. The value is `t` if *frame* is visible, `nil` if it is invisible, and `icon` if it is iconified.

On a text terminal, all frames are considered visible, whether they are currently being displayed or not.

`iconify-frame` **&optional** *frame* [Command]

This function iconifies frame *frame*. If you omit *frame*, it iconifies the selected frame.

`make-frame-visible` **&optional** *frame* [Command]

This function makes frame *frame* visible. If you omit *frame*, it makes the selected frame visible. This does not raise the frame, but you can do that with `raise-frame` if you wish (see Section 29.11 [Raising and Lowering], page 86).

`make-frame-invisible` **&optional** *frame force* [Command]

This function makes frame *frame* invisible. If you omit *frame*, it makes the selected frame invisible.

Unless *force* is non-`nil`, this function refuses to make *frame* invisible if all other frames are invisible..

The visibility status of a frame is also available as a frame parameter. You can read or change it as such. See Section 29.3.3.6 [Management Parameters], page 75. The user can also iconify and deiconify frames with the window manager. This happens below the level at which Emacs can exert any control, but Emacs does provide events that you can use to keep track of such changes. See Section 21.7.10 [Misc Events], page 334, vol. 1.

29.11 Raising and Lowering Frames

Most window systems use a desktop metaphor. Part of this metaphor is the idea that windows are stacked in a notional third dimension perpendicular to the screen surface, and thus ordered from "highest" to "lowest". Where two windows overlap, the one higher up covers the one underneath. Even a window at the bottom of the stack can be seen if no other window overlaps it.

A window's place in this ordering is not fixed; in fact, users tend to change the order frequently. *Raising* a window means moving it "up", to the top of the stack. *Lowering* a window means moving it to the bottom of the stack. This motion is in the notional third dimension only, and does not change the position of the window on the screen.

With Emacs, frames constitute the windows in the metaphor sketched above. You can raise and lower frames using these functions:

`raise-frame` **&optional** *frame* [Command]
This function raises frame *frame* (default, the selected frame). If *frame* is invisible or iconified, this makes it visible.

`lower-frame` **&optional** *frame* [Command]
This function lowers frame *frame* (default, the selected frame).

`minibuffer-auto-raise` [User Option]
If this is non-`nil`, activation of the minibuffer raises the frame that the minibuffer window is in.

You can also enable auto-raise (raising automatically when a frame is selected) or auto-lower (lowering automatically when it is deselected) for any frame using frame parameters. See Section 29.3.3.6 [Management Parameters], page 75.

29.12 Frame Configurations

A *frame configuration* records the current arrangement of frames, all their properties, and the window configuration of each one. (See Section 28.23 [Window Configurations], page 60.)

`current-frame-configuration` [Function]
This function returns a frame configuration list that describes the current arrangement of frames and their contents.

`set-frame-configuration` *configuration* **&optional** *nodelete* [Function]
This function restores the state of frames described in *configuration*. However, this function does not restore deleted frames.

Ordinarily, this function deletes all existing frames not listed in *configuration*. But if *nodelete* is non-`nil`, the unwanted frames are iconified instead.

29.13 Mouse Tracking

Sometimes it is useful to *track* the mouse, which means to display something to indicate where the mouse is and move the indicator as the mouse moves. For efficient mouse tracking, you need a way to wait until the mouse actually moves.

The convenient way to track the mouse is to ask for events to represent mouse motion. Then you can wait for motion by waiting for an event. In addition, you can easily handle any other sorts of events that may occur. That is useful, because normally you don't want to track the mouse forever—only until some other event, such as the release of a button.

`track-mouse` *body...* [Special Form]

This special form executes *body*, with generation of mouse motion events enabled. Typically, *body* would use `read-event` to read the motion events and modify the display accordingly. See Section 21.7.8 [Motion Events], page 334, vol. 1, for the format of mouse motion events.

The value of `track-mouse` is that of the last form in *body*. You should design *body* to return when it sees the up-event that indicates the release of the button, or whatever kind of event means it is time to stop tracking.

The usual purpose of tracking mouse motion is to indicate on the screen the consequences of pushing or releasing a button at the current position.

In many cases, you can avoid the need to track the mouse by using the `mouse-face` text property (see Section 32.19.4 [Special Properties], page 162). That works at a much lower level and runs more smoothly than Lisp-level mouse tracking.

29.14 Mouse Position

The functions `mouse-position` and `set-mouse-position` give access to the current position of the mouse.

`mouse-position` [Function]

This function returns a description of the position of the mouse. The value looks like (*frame x . y*), where *x* and *y* are integers giving the position in characters relative to the top left corner of the inside of *frame*.

`mouse-position-function` [Variable]

If non-`nil`, the value of this variable is a function for `mouse-position` to call. `mouse-position` calls this function just before returning, with its normal return value as the sole argument, and it returns whatever this function returns to it.

This abnormal hook exists for the benefit of packages like '`xt-mouse.el`' that need to do mouse handling at the Lisp level.

`set-mouse-position` *frame x y* [Function]

This function *warps the mouse* to position *x*, *y* in frame *frame*. The arguments *x* and *y* are integers, giving the position in characters relative to the top left corner of the inside of *frame*. If *frame* is not visible, this function does nothing. The return value is not significant.

`mouse-pixel-position` [Function]
: This function is like `mouse-position` except that it returns coordinates in units of pixels rather than units of characters.

`set-mouse-pixel-position` *frame x y* [Function]
: This function warps the mouse like `set-mouse-position` except that *x* and *y* are in units of pixels rather than units of characters. These coordinates are not required to be within the frame.

  If *frame* is not visible, this function does nothing. The return value is not significant.

`frame-pointer-visible-p` **&optional** *frame* [Function]
: This predicate function returns non-`nil` if the mouse pointer displayed on *frame* is visible; otherwise it returns `nil`. *frame* omitted or `nil` means the selected frame. This is useful when `make-pointer-invisible` is set to `t`: it allows to know if the pointer has been hidden. See Section "Mouse Avoidance" in *The Emacs Manual*.

29.15 Pop-Up Menus

When using a window system, a Lisp program can pop up a menu so that the user can choose an alternative with the mouse.

`x-popup-menu` *position menu* [Function]
: This function displays a pop-up menu and returns an indication of what selection the user makes.

  The argument *position* specifies where on the screen to put the top left corner of the menu. It can be either a mouse button event (which says to put the menu where the user actuated the button) or a list of this form:

  ```
  ((xoffset yoffset) window)
  ```

  where *xoffset* and *yoffset* are coordinates, measured in pixels, counting from the top left corner of *window*. *window* may be a window or a frame.

  If *position* is `t`, it means to use the current mouse position. If *position* is `nil`, it means to precompute the key binding equivalents for the keymaps specified in *menu*, without actually displaying or popping up the menu.

  The argument *menu* says what to display in the menu. It can be a keymap or a list of keymaps (see Section 22.17 [Menu Keymaps], page 384, vol. 1). In this case, the return value is the list of events corresponding to the user's choice. This list has more than one element if the choice occurred in a submenu. (Note that `x-popup-menu` does not actually execute the command bound to that sequence of events.) On toolkits that support menu titles, the title is taken from the prompt string of *menu* if *menu* is a keymap, or from the prompt string of the first keymap in *menu* if it is a list of keymaps (see Section 22.17.1 [Defining Menus], page 384, vol. 1).

  Alternatively, *menu* can have the following form:

  ```
  (title pane1 pane2...)
  ```

  where each pane is a list of form

```
(title item1 item2...)
```

Each item should normally be a cons cell `(line . value)`, where *line* is a string, and *value* is the value to return if that *line* is chosen. An item can also be a string; this makes a non-selectable line in the menu.

If the user gets rid of the menu without making a valid choice, for instance by clicking the mouse away from a valid choice or by typing keyboard input, then this normally results in a quit and `x-popup-menu` does not return. But if *position* is a mouse button event (indicating that the user invoked the menu with the mouse) then no quit occurs and `x-popup-menu` returns `nil`.

Usage note: Don't use `x-popup-menu` to display a menu if you could do the job with a prefix key defined with a menu keymap. If you use a menu keymap to implement a menu, `C-h c` and `C-h a` can see the individual items in that menu and provide help for them. If instead you implement the menu by defining a command that calls `x-popup-menu`, the help facilities cannot know what happens inside that command, so they cannot give any help for the menu's items.

The menu bar mechanism, which lets you switch between submenus by moving the mouse, cannot look within the definition of a command to see that it calls `x-popup-menu`. Therefore, if you try to implement a submenu using `x-popup-menu`, it cannot work with the menu bar in an integrated fashion. This is why all menu bar submenus are implemented with menu keymaps within the parent menu, and never with `x-popup-menu`. See Section 22.17.5 [Menu Bar], page 391, vol. 1.

If you want a menu bar submenu to have contents that vary, you should still use a menu keymap to implement it. To make the contents vary, add a hook function to `menu-bar-update-hook` to update the contents of the menu keymap as necessary.

29.16 Dialog Boxes

A dialog box is a variant of a pop-up menu—it looks a little different, it always appears in the center of a frame, and it has just one level and one or more buttons. The main use of dialog boxes is for asking questions that the user can answer with "yes", "no", and a few other alternatives. With a single button, they can also force the user to acknowledge important information. The functions `y-or-n-p` and `yes-or-no-p` use dialog boxes instead of the keyboard, when called from commands invoked by mouse clicks.

`x-popup-dialog` *position contents* **&optional** *header* [Function]

This function displays a pop-up dialog box and returns an indication of what selection the user makes. The argument *contents* specifies the alternatives to offer; it has this format:

```
(title (string . value)...)
```

which looks like the list that specifies a single pane for `x-popup-menu`.

The return value is *value* from the chosen alternative.

As for `x-popup-menu`, an element of the list may be just a string instead of a cons cell `(string . value)`. That makes a box that cannot be selected.

If `nil` appears in the list, it separates the left-hand items from the right-hand items; items that precede the `nil` appear on the left, and items that follow the `nil` appear

on the right. If you don't include a `nil` in the list, then approximately half the items appear on each side.

Dialog boxes always appear in the center of a frame; the argument *position* specifies which frame. The possible values are as in `x-popup-menu`, but the precise coordinates or the individual window don't matter; only the frame matters.

If *header* is non-`nil`, the frame title for the box is '`Information`', otherwise it is '`Question`'. The former is used for `message-box` (see [message-box], page 303).

In some configurations, Emacs cannot display a real dialog box; so instead it displays the same items in a pop-up menu in the center of the frame.

If the user gets rid of the dialog box without making a valid choice, for instance using the window manager, then this produces a quit and `x-popup-dialog` does not return.

29.17 Pointer Shape

You can specify the mouse pointer style for particular text or images using the `pointer` text property, and for images with the `:pointer` and `:map` image properties. The values you can use in these properties are `text` (or `nil`), `arrow`, `hand`, `vdrag`, `hdrag`, `modeline`, and `hourglass`. `text` stands for the usual mouse pointer style used over text.

Over void parts of the window (parts that do not correspond to any of the buffer contents), the mouse pointer usually uses the `arrow` style, but you can specify a different style (one of those above) by setting `void-text-area-pointer`.

`void-text-area-pointer` [User Option]
: This variable specifies the mouse pointer style for void text areas. These include the areas after the end of a line or below the last line in the buffer. The default is to use the `arrow` (non-text) pointer style.

When using X, you can specify what the `text` pointer style really looks like by setting the variable `x-pointer-shape`.

`x-pointer-shape` [Variable]
: This variable specifies the pointer shape to use ordinarily in the Emacs frame, for the `text` pointer style.

`x-sensitive-text-pointer-shape` [Variable]
: This variable specifies the pointer shape to use when the mouse is over mouse-sensitive text.

These variables affect newly created frames. They do not normally affect existing frames; however, if you set the mouse color of a frame, that also installs the current value of those two variables. See Section 29.3.3.8 [Font and Color Parameters], page 77.

The values you can use, to specify either of these pointer shapes, are defined in the file '`lisp/term/x-win.el`'. Use *M-x apropos RET x-pointer RET* to see a list of them.

29.18 Window System Selections

In the X window system, data can be transferred between different applications by means of *selections*. X defines an arbitrary number of *selection types*, each of which can store its own data; however, only three are commonly used: the *clipboard*, *primary selection*, and *secondary selection*. See Section "Cut and Paste" in *The GNU Emacs Manual*, for Emacs commands that make use of these selections. This section documents the low-level functions for reading and setting X selections.

`x-set-selection` *type data* [Command]
: This function sets an X selection. It takes two arguments: a selection type *type*, and the value to assign to it, *data*.

  type should be a symbol; it is usually one of `PRIMARY`, `SECONDARY` or `CLIPBOARD`. These are symbols with upper-case names, in accord with X Window System conventions. If *type* is `nil`, that stands for `PRIMARY`.

  If *data* is `nil`, it means to clear out the selection. Otherwise, *data* may be a string, a symbol, an integer (or a cons of two integers or list of two integers), an overlay, or a cons of two markers pointing to the same buffer. An overlay or a pair of markers stands for text in the overlay or between the markers. The argument *data* may also be a vector of valid non-vector selection values.

  This function returns *data*.

`x-get-selection` **&optional** *type data-type* [Function]
: This function accesses selections set up by Emacs or by other X clients. It takes two optional arguments, *type* and *data-type*. The default for *type*, the selection type, is `PRIMARY`.

  The *data-type* argument specifies the form of data conversion to use, to convert the raw data obtained from another X client into Lisp data. Meaningful values include `TEXT`, `STRING`, `UTF8_STRING`, `TARGETS`, `LENGTH`, `DELETE`, `FILE_NAME`, `CHARACTER_POSITION`, `NAME`, `LINE_NUMBER`, `COLUMN_NUMBER`, `OWNER_OS`, `HOST_NAME`, `USER`, `CLASS`, `ATOM`, and `INTEGER`. (These are symbols with upper-case names in accord with X conventions.) The default for *data-type* is `STRING`.

`selection-coding-system` [User Option]
: This variable specifies the coding system to use when reading and writing selections or the clipboard. See Section 33.9 [Coding Systems], page 193. The default is `compound-text-with-extensions`, which converts to the text representation that X11 normally uses.

When Emacs runs on MS-Windows, it does not implement X selections in general, but it does support the clipboard. `x-get-selection` and `x-set-selection` on MS-Windows support the text data type only; if the clipboard holds other types of data, Emacs treats the clipboard as empty.

29.19 Drag and Drop

When a user drags something from another application over Emacs, that other application expects Emacs to tell it if Emacs can handle the data that is dragged. The variable `x-dnd-test-function` is used by Emacs to determine what to reply. The default value is

`x-dnd-default-test-function` which accepts drops if the type of the data to be dropped is present in `x-dnd-known-types`. You can customize `x-dnd-test-function` and/or `x-dnd-known-types` if you want Emacs to accept or reject drops based on some other criteria.

If you want to change the way Emacs handles drop of different types or add a new type, customize `x-dnd-types-alist`. This requires detailed knowledge of what types other applications use for drag and drop.

When an URL is dropped on Emacs it may be a file, but it may also be another URL type (ftp, http, etc.). Emacs first checks `dnd-protocol-alist` to determine what to do with the URL. If there is no match there and if `browse-url-browser-function` is an alist, Emacs looks for a match there. If no match is found the text for the URL is inserted. If you want to alter Emacs behavior, you can customize these variables.

29.20 Color Names

A color name is text (usually in a string) that specifies a color. Symbolic names such as '`black`', '`white`', '`red`', etc., are allowed; use *`M-x list-colors-display`* to see a list of defined names. You can also specify colors numerically in forms such as '`#rgb`' and '`RGB:r/g/b`', where *r* specifies the red level, *g* specifies the green level, and *b* specifies the blue level. You can use either one, two, three, or four hex digits for *r*; then you must use the same number of hex digits for all *g* and *b* as well, making either 3, 6, 9 or 12 hex digits in all. (See the documentation of the X Window System for more details about numerical RGB specification of colors.)

These functions provide a way to determine which color names are valid, and what they look like. In some cases, the value depends on the *selected frame*, as described below; see Section 29.9 [Input Focus], page 83, for the meaning of the term "selected frame".

To read user input of color names with completion, use `read-color` (see Section 20.6.4 [High-Level Completion], page 298, vol. 1).

`color-defined-p` *color* **&optional** *frame* [Function]

This function reports whether a color name is meaningful. It returns `t` if so; otherwise, `nil`. The argument *frame* says which frame's display to ask about; if *frame* is omitted or `nil`, the selected frame is used.

Note that this does not tell you whether the display you are using really supports that color. When using X, you can ask for any defined color on any kind of display, and you will get some result—typically, the closest it can do. To determine whether a frame can really display a certain color, use `color-supported-p` (see below).

This function used to be called `x-color-defined-p`, and that name is still supported as an alias.

`defined-colors` **&optional** *frame* [Function]

This function returns a list of the color names that are defined and supported on frame *frame* (default, the selected frame). If *frame* does not support colors, the value is `nil`.

This function used to be called `x-defined-colors`, and that name is still supported as an alias.

`color-supported-p` *color* **&optional** *frame background-p* [Function]
: This returns `t` if *frame* can really display the color *color* (or at least something close to it). If *frame* is omitted or `nil`, the question applies to the selected frame.

  Some terminals support a different set of colors for foreground and background. If *background-p* is non-`nil`, that means you are asking whether *color* can be used as a background; otherwise you are asking whether it can be used as a foreground.

  The argument *color* must be a valid color name.

`color-gray-p` *color* **&optional** *frame* [Function]
: This returns `t` if *color* is a shade of gray, as defined on *frame*'s display. If *frame* is omitted or `nil`, the question applies to the selected frame. If *color* is not a valid color name, this function returns `nil`.

`color-values` *color* **&optional** *frame* [Function]
: This function returns a value that describes what *color* should ideally look like on *frame*. If *color* is defined, the value is a list of three integers, which give the amount of red, the amount of green, and the amount of blue. Each integer ranges in principle from 0 to 65535, but some displays may not use the full range. This three-element list is called the *rgb values* of the color.

  If *color* is not defined, the value is `nil`.

```
(color-values "black")
     ⇒ (0 0 0)
(color-values "white")
     ⇒ (65280 65280 65280)
(color-values "red")
     ⇒ (65280 0 0)
(color-values "pink")
     ⇒ (65280 49152 51968)
(color-values "hungry")
     ⇒ nil
```

  The color values are returned for *frame*'s display. If *frame* is omitted or `nil`, the information is returned for the selected frame's display. If the frame cannot display colors, the value is `nil`.

  This function used to be called `x-color-values`, and that name is still supported as an alias.

29.21 Text Terminal Colors

Text terminals usually support only a small number of colors, and the computer uses small integers to select colors on the terminal. This means that the computer cannot reliably tell what the selected color looks like; instead, you have to inform your application which small integers correspond to which colors. However, Emacs does know the standard set of colors and will try to use them automatically.

The functions described in this section control how terminal colors are used by Emacs.

Several of these functions use or return *rgb values*, described in Section 29.20 [Color Names], page 92.

These functions accept a display (either a frame or the name of a terminal) as an optional argument. We hope in the future to make Emacs support different colors on different text terminals; then this argument will specify which terminal to operate on (the default being the selected frame's terminal; see Section 29.9 [Input Focus], page 83). At present, though, the *frame* argument has no effect.

`tty-color-define` *name number* **&optional** *rgb frame* [Function]
This function associates the color name *name* with color number *number* on the terminal.

The optional argument *rgb*, if specified, is an rgb value, a list of three numbers that specify what the color actually looks like. If you do not specify *rgb*, then this color cannot be used by `tty-color-approximate` to approximate other colors, because Emacs will not know what it looks like.

`tty-color-clear` **&optional** *frame* [Function]
This function clears the table of defined colors for a text terminal.

`tty-color-alist` **&optional** *frame* [Function]
This function returns an alist recording the known colors supported by a text terminal.

Each element has the form `(`*`name number . rgb`*`)` or `(`*`name number`*`)`. Here, *name* is the color name, *number* is the number used to specify it to the terminal. If present, *rgb* is a list of three color values (for red, green, and blue) that says what the color actually looks like.

`tty-color-approximate` *rgb* **&optional** *frame* [Function]
This function finds the closest color, among the known colors supported for *display*, to that described by the rgb value *rgb* (a list of color values). The return value is an element of `tty-color-alist`.

`tty-color-translate` *color* **&optional** *frame* [Function]
This function finds the closest color to *color* among the known colors supported for *display* and returns its index (an integer). If the name *color* is not defined, the value is `nil`.

29.22 X Resources

This section describes some of the functions and variables for querying and using X resources, or their equivalent on your operating system. See Section "X Resources" in *The GNU Emacs Manual*, for more information about X resources.

`x-get-resource` *attribute class* **&optional** *component subclass* [Function]
The function `x-get-resource` retrieves a resource value from the X Window defaults database.

Resources are indexed by a combination of a *key* and a *class*. This function searches using a key of the form '*`instance.attribute`*' (where *instance* is the name under which Emacs was invoked), and using '`Emacs.`*`class`*' as the class.

The optional arguments *component* and *subclass* add to the key and the class, respectively. You must specify both of them or neither. If you specify them, the key is '*`instance.component.attribute`*', and the class is '`Emacs.`*`class.subclass`*'.

`x-resource-class` [Variable]
: This variable specifies the application name that `x-get-resource` should look up. The default value is `"Emacs"`. You can examine X resources for application names other than "Emacs" by binding this variable to some other string, around a call to `x-get-resource`.

`x-resource-name` [Variable]
: This variable specifies the instance name that `x-get-resource` should look up. The default value is the name Emacs was invoked with, or the value specified with the '`-name`' or '`-rn`' switches.

To illustrate some of the above, suppose that you have the line:

```
xterm.vt100.background: yellow
```

in your X resources file (whose name is usually '`~/.Xdefaults`' or '`~/.Xresources`'). Then:

```
(let ((x-resource-class "XTerm") (x-resource-name "xterm"))
  (x-get-resource "vt100.background" "VT100.Background"))
     ⇒ "yellow"
(let ((x-resource-class "XTerm") (x-resource-name "xterm"))
  (x-get-resource "background" "VT100" "vt100" "Background"))
     ⇒ "yellow"
```

`inhibit-x-resources` [Variable]
: If this variable is non-`nil`, Emacs does not look up X resources, and X resources do not have any effect when creating new frames.

29.23 Display Feature Testing

The functions in this section describe the basic capabilities of a particular display. Lisp programs can use them to adapt their behavior to what the display can do. For example, a program that ordinarily uses a popup menu could use the minibuffer if popup menus are not supported.

The optional argument *display* in these functions specifies which display to ask the question about. It can be a display name, a frame (which designates the display that frame is on), or `nil` (which refers to the selected frame's display, see Section 29.9 [Input Focus], page 83).

See Section 29.20 [Color Names], page 92, Section 29.21 [Text Terminal Colors], page 93, for other functions to obtain information about displays.

`display-popup-menus-p` **&optional** *display* [Function]
: This function returns `t` if popup menus are supported on *display*, `nil` if not. Support for popup menus requires that the mouse be available, since the user cannot choose menu items without a mouse.

`display-graphic-p` **&optional** *display* [Function]
: This function returns `t` if *display* is a graphic display capable of displaying several frames and several different fonts at once. This is true for displays that use a window system such as X, and false for text terminals.

`display-mouse-p` **&optional** *display* [Function]
This function returns `t` if *display* has a mouse available, `nil` if not.

`display-color-p` **&optional** *display* [Function]
This function returns `t` if the screen is a color screen. It used to be called `x-display-color-p`, and that name is still supported as an alias.

`display-grayscale-p` **&optional** *display* [Function]
This function returns `t` if the screen can display shades of gray. (All color displays can do this.)

`display-supports-face-attributes-p` *attributes* **&optional** *display* [Function]
This function returns non-`nil` if all the face attributes in *attributes* are supported (see Section 38.12.2 [Face Attributes], page 327).

The definition of 'supported' is somewhat heuristic, but basically means that a face containing all the attributes in *attributes*, when merged with the default face for display, can be represented in a way that's

1. different in appearance than the default face, and
2. 'close in spirit' to what the attributes specify, if not exact.

Point (2) implies that a `:weight black` attribute will be satisfied by any display that can display bold, as will `:foreground "yellow"` as long as some yellowish color can be displayed, but `:slant italic` will *not* be satisfied by the tty display code's automatic substitution of a 'dim' face for italic.

`display-selections-p` **&optional** *display* [Function]
This function returns `t` if *display* supports selections. Windowed displays normally support selections, but they may also be supported in some other cases.

`display-images-p` **&optional** *display* [Function]
This function returns `t` if *display* can display images. Windowed displays ought in principle to handle images, but some systems lack the support for that. On a display that does not support images, Emacs cannot display a tool bar.

`display-screens` **&optional** *display* [Function]
This function returns the number of screens associated with the display.

`display-pixel-height` **&optional** *display* [Function]
This function returns the height of the screen in pixels. On a character terminal, it gives the height in characters.

For graphical terminals, note that on "multi-monitor" setups this refers to the pixel width for all physical monitors associated with *display*. See Section 29.2 [Multiple Terminals], page 67.

`display-pixel-width` **&optional** *display* [Function]
This function returns the width of the screen in pixels. On a character terminal, it gives the width in characters.

For graphical terminals, note that on "multi-monitor" setups this refers to the pixel width for all physical monitors associated with *display*. See Section 29.2 [Multiple Terminals], page 67.

`display-mm-height` **&optional** *display* [Function]
: This function returns the height of the screen in millimeters, or `nil` if Emacs cannot get that information.

`display-mm-width` **&optional** *display* [Function]
: This function returns the width of the screen in millimeters, or `nil` if Emacs cannot get that information.

`display-mm-dimensions-alist` [User Option]
: This variable allows the user to specify the dimensions of graphical displays returned by `display-mm-height` and `display-mm-width` in case the system provides incorrect values.

`display-backing-store` **&optional** *display* [Function]
: This function returns the backing store capability of the display. Backing store means recording the pixels of windows (and parts of windows) that are not exposed, so that when exposed they can be displayed very quickly.

  Values can be the symbols `always`, `when-mapped`, or `not-useful`. The function can also return `nil` when the question is inapplicable to a certain kind of display.

`display-save-under` **&optional** *display* [Function]
: This function returns non-`nil` if the display supports the SaveUnder feature. That feature is used by pop-up windows to save the pixels they obscure, so that they can pop down quickly.

`display-planes` **&optional** *display* [Function]
: This function returns the number of planes the display supports. This is typically the number of bits per pixel. For a tty display, it is log to base two of the number of colors supported.

`display-visual-class` **&optional** *display* [Function]
: This function returns the visual class for the screen. The value is one of the symbols `static-gray` (a limited, unchangeable number of grays), `gray-scale` (a full range of grays), `static-color` (a limited, unchangeable number of colors), `pseudo-color` (a limited number of colors), `true-color` (a full range of colors), and `direct-color` (a full range of colors).

`display-color-cells` **&optional** *display* [Function]
: This function returns the number of color cells the screen supports.

These functions obtain additional information specifically about X displays.

`x-server-version` **&optional** *display* [Function]
: This function returns the list of version numbers of the X server running the display. The value is a list of three integers: the major and minor version numbers of the X protocol, and the distributor-specific release number of the X server software itself.

`x-server-vendor` **&optional** *display* [Function]
: This function returns the "vendor" that provided the X server software (as a string). Really this means whoever distributes the X server.

When the developers of X labeled software distributors as “vendors”, they showed their false assumption that no system could ever be developed and distributed non-commercially.

30 Positions

A *position* is the index of a character in the text of a buffer. More precisely, a position identifies the place between two characters (or before the first character, or after the last character), so we can speak of the character before or after a given position. However, we often speak of the character "at" a position, meaning the character after that position.

Positions are usually represented as integers starting from 1, but can also be represented as *markers*—special objects that relocate automatically when text is inserted or deleted so they stay with the surrounding characters. Functions that expect an argument to be a position (an integer), but accept a marker as a substitute, normally ignore which buffer the marker points into; they convert the marker to an integer, and use that integer, exactly as if you had passed the integer as the argument, even if the marker points to the "wrong" buffer. A marker that points nowhere cannot convert to an integer; using it instead of an integer causes an error. See Chapter 31 [Markers], page 112.

See also the "field" feature (see Section 32.19.9 [Fields], page 172), which provides functions that are used by many cursor-motion commands.

30.1 Point

Point is a special buffer position used by many editing commands, including the self-inserting typed characters and text insertion functions. Other commands move point through the text to allow editing and insertion at different places.

Like other positions, point designates a place between two characters (or before the first character, or after the last character), rather than a particular character. Usually terminals display the cursor over the character that immediately follows point; point is actually before the character on which the cursor sits.

The value of point is a number no less than 1, and no greater than the buffer size plus 1. If narrowing is in effect (see Section 30.4 [Narrowing], page 109), then point is constrained to fall within the accessible portion of the buffer (possibly at one end of it).

Each buffer has its own value of point, which is independent of the value of point in other buffers. Each window also has a value of point, which is independent of the value of point in other windows on the same buffer. This is why point can have different values in various windows that display the same buffer. When a buffer appears in only one window, the buffer's point and the window's point normally have the same value, so the distinction is rarely important. See Section 28.17 [Window Point], page 48, for more details.

`point` [Function]

This function returns the value of point in the current buffer, as an integer.

```
(point)
     ⇒ 175
```

`point-min` [Function]

This function returns the minimum accessible value of point in the current buffer. This is normally 1, but if narrowing is in effect, it is the position of the start of the region that you narrowed to. (See Section 30.4 [Narrowing], page 109.)

`point-max` [Function]
: This function returns the maximum accessible value of point in the current buffer. This is `(1+ (buffer-size))`, unless narrowing is in effect, in which case it is the position of the end of the region that you narrowed to. (See Section 30.4 [Narrowing], page 109.)

`buffer-end` *flag* [Function]
: This function returns `(point-max)` if *flag* is greater than 0, `(point-min)` otherwise. The argument *flag* must be a number.

`buffer-size` **&optional** *buffer* [Function]
: This function returns the total number of characters in the current buffer. In the absence of any narrowing (see Section 30.4 [Narrowing], page 109), `point-max` returns a value one larger than this.

  If you specify a buffer, *buffer*, then the value is the size of *buffer*.

```
(buffer-size)
     ⇒ 35
(point-max)
     ⇒ 36
```

30.2 Motion

Motion functions change the value of point, either relative to the current value of point, relative to the beginning or end of the buffer, or relative to the edges of the selected window. See Section 30.1 [Point], page 99.

30.2.1 Motion by Characters

These functions move point based on a count of characters. `goto-char` is the fundamental primitive; the other functions use that.

`goto-char` *position* [Command]
: This function sets point in the current buffer to the value *position*. If *position* is less than 1, it moves point to the beginning of the buffer. If *position* is greater than the length of the buffer, it moves point to the end.

  If narrowing is in effect, *position* still counts from the beginning of the buffer, but point cannot go outside the accessible portion. If *position* is out of range, `goto-char` moves point to the beginning or the end of the accessible portion.

  When this function is called interactively, *position* is the numeric prefix argument, if provided; otherwise it is read from the minibuffer.

  `goto-char` returns *position*.

`forward-char` **&optional** *count* [Command]
: This function moves point *count* characters forward, towards the end of the buffer (or backward, towards the beginning of the buffer, if *count* is negative). If *count* is `nil`, the default is 1.

  If this attempts to move past the beginning or end of the buffer (or the limits of the accessible portion, when narrowing is in effect), it signals an error with error symbol `beginning-of-buffer` or `end-of-buffer`.

In an interactive call, *count* is the numeric prefix argument.

`backward-char` **&optional** *count* [Command]
This is just like `forward-char` except that it moves in the opposite direction.

30.2.2 Motion by Words

These functions for parsing words use the syntax table to decide whether a given character is part of a word. See Chapter 35 [Syntax Tables], page 234.

`forward-word` **&optional** *count* [Command]
This function moves point forward *count* words (or backward if *count* is negative). If *count* is `nil`, it moves forward one word.

"Moving one word" means moving until point crosses a word-constituent character and then encounters a word-separator character. However, this function cannot move point past the boundary of the accessible portion of the buffer, or across a field boundary (see Section 32.19.9 [Fields], page 172). The most common case of a field boundary is the end of the prompt in the minibuffer.

If it is possible to move *count* words, without being stopped prematurely by the buffer boundary or a field boundary, the value is `t`. Otherwise, the return value is `nil` and point stops at the buffer boundary or field boundary.

If `inhibit-field-text-motion` is non-`nil`, this function ignores field boundaries.

In an interactive call, *count* is specified by the numeric prefix argument. If *count* is omitted or `nil`, it defaults to 1.

`backward-word` **&optional** *count* [Command]
This function is just like `forward-word`, except that it moves backward until encountering the front of a word, rather than forward.

`words-include-escapes` [User Option]
This variable affects the behavior of `forward-word` and everything that uses it. If it is non-`nil`, then characters in the "escape" and "character quote" syntax classes count as part of words. Otherwise, they do not.

`inhibit-field-text-motion` [Variable]
If this variable is non-`nil`, certain motion functions including `forward-word`, `forward-sentence`, and `forward-paragraph` ignore field boundaries.

30.2.3 Motion to an End of the Buffer

To move point to the beginning of the buffer, write:

```
(goto-char (point-min))
```

Likewise, to move to the end of the buffer, use:

```
(goto-char (point-max))
```

Here are two commands that users use to do these things. They are documented here to warn you not to use them in Lisp programs, because they set the mark and display messages in the echo area.

`beginning-of-buffer` **&optional** *n* [Command]
: This function moves point to the beginning of the buffer (or the limits of the accessible portion, when narrowing is in effect), setting the mark at the previous position (except in Transient Mark mode, if the mark is already active, it does not set the mark.)

  If *n* is non-`nil`, then it puts point *n* tenths of the way from the beginning of the accessible portion of the buffer. In an interactive call, *n* is the numeric prefix argument, if provided; otherwise *n* defaults to `nil`.

  Warning: Don't use this function in Lisp programs!

`end-of-buffer` **&optional** *n* [Command]
: This function moves point to the end of the buffer (or the limits of the accessible portion, when narrowing is in effect), setting the mark at the previous position (except in Transient Mark mode when the mark is already active). If *n* is non-`nil`, then it puts point *n* tenths of the way from the end of the accessible portion of the buffer.

  In an interactive call, *n* is the numeric prefix argument, if provided; otherwise *n* defaults to `nil`.

  Warning: Don't use this function in Lisp programs!

30.2.4 Motion by Text Lines

Text lines are portions of the buffer delimited by newline characters, which are regarded as part of the previous line. The first text line begins at the beginning of the buffer, and the last text line ends at the end of the buffer whether or not the last character is a newline. The division of the buffer into text lines is not affected by the width of the window, by line continuation in display, or by how tabs and control characters are displayed.

`beginning-of-line` **&optional** *count* [Command]
: This function moves point to the beginning of the current line. With an argument *count* not `nil` or 1, it moves forward *count*−1 lines and then to the beginning of the line.

  This function does not move point across a field boundary (see Section 32.19.9 [Fields], page 172) unless doing so would move beyond there to a different line; therefore, if *count* is `nil` or 1, and point starts at a field boundary, point does not move. To ignore field boundaries, either bind `inhibit-field-text-motion` to `t`, or use the `forward-line` function instead. For instance, `(forward-line 0)` does the same thing as `(beginning-of-line)`, except that it ignores field boundaries.

  If this function reaches the end of the buffer (or of the accessible portion, if narrowing is in effect), it positions point there. No error is signaled.

`line-beginning-position` **&optional** *count* [Function]
: Return the position that `(beginning-of-line` *`count`*`)` would move to.

`end-of-line` **&optional** *count* [Command]
: This function moves point to the end of the current line. With an argument *count* not `nil` or 1, it moves forward *count*−1 lines and then to the end of the line.

  This function does not move point across a field boundary (see Section 32.19.9 [Fields], page 172) unless doing so would move beyond there to a different line; therefore, if

count is `nil` or 1, and point starts at a field boundary, point does not move. To ignore field boundaries, bind `inhibit-field-text-motion` to `t`.

If this function reaches the end of the buffer (or of the accessible portion, if narrowing is in effect), it positions point there. No error is signaled.

`line-end-position` **&optional** *count* [Function]
Return the position that `(end-of-line` *count*`)` would move to.

`forward-line` **&optional** *count* [Command]
This function moves point forward *count* lines, to the beginning of the line. If *count* is negative, it moves point −*count* lines backward, to the beginning of a line. If *count* is zero, it moves point to the beginning of the current line. If *count* is `nil`, that means 1.

If `forward-line` encounters the beginning or end of the buffer (or of the accessible portion) before finding that many lines, it sets point there. No error is signaled.

`forward-line` returns the difference between *count* and the number of lines actually moved. If you attempt to move down five lines from the beginning of a buffer that has only three lines, point stops at the end of the last line, and the value will be 2.

In an interactive call, *count* is the numeric prefix argument.

`count-lines` *start end* [Function]
This function returns the number of lines between the positions *start* and *end* in the current buffer. If *start* and *end* are equal, then it returns 0. Otherwise it returns at least 1, even if *start* and *end* are on the same line. This is because the text between them, considered in isolation, must contain at least one line unless it is empty.

`count-words` *start end* [Command]
This function returns the number of words between the positions *start* and *end* in the current buffer.

This function can also be called interactively. In that case, it prints a message reporting the number of lines, words, and characters in the buffer, or in the region if the region is active.

`line-number-at-pos` **&optional** *pos* [Function]
This function returns the line number in the current buffer corresponding to the buffer position *pos*. If *pos* is `nil` or omitted, the current buffer position is used.

Also see the functions `bolp` and `eolp` in Section 32.1 [Near Point], page 122. These functions do not move point, but test whether it is already at the beginning or end of a line.

30.2.5 Motion by Screen Lines

The line functions in the previous section count text lines, delimited only by newline characters. By contrast, these functions count screen lines, which are defined by the way the text appears on the screen. A text line is a single screen line if it is short enough to fit the width of the selected window, but otherwise it may occupy several screen lines.

In some cases, text lines are truncated on the screen rather than continued onto additional screen lines. In these cases, `vertical-motion` moves point much like `forward-line`. See Section 38.3 [Truncation], page 300.

Because the width of a given string depends on the flags that control the appearance of certain characters, `vertical-motion` behaves differently, for a given piece of text, depending on the buffer it is in, and even on the selected window (because the width, the truncation flag, and display table may vary between windows). See Section 38.20.1 [Usual Display], page 376.

These functions scan text to determine where screen lines break, and thus take time proportional to the distance scanned. If you intend to use them heavily, Emacs provides caches which may improve the performance of your code. See Section 38.3 [Truncation], page 300.

`vertical-motion` *count* **&optional** *window* [Function]

This function moves point to the start of the screen line *count* screen lines down from the screen line containing point. If *count* is negative, it moves up instead.

The *count* argument can be a cons cell, `(cols . lines)`, instead of an integer. Then the function moves by *lines* screen lines, and puts point *cols* columns from the start of that screen line.

The return value is the number of screen lines over which point was moved. The value may be less in absolute value than *count* if the beginning or end of the buffer was reached.

The window *window* is used for obtaining parameters such as the width, the horizontal scrolling, and the display table. But `vertical-motion` always operates on the current buffer, even if *window* currently displays some other buffer.

`count-screen-lines` **&optional** *beg end count-final-newline window* [Function]

This function returns the number of screen lines in the text from *beg* to *end*. The number of screen lines may be different from the number of actual lines, due to line continuation, the display table, etc. If *beg* and *end* are `nil` or omitted, they default to the beginning and end of the accessible portion of the buffer.

If the region ends with a newline, that is ignored unless the optional third argument *count-final-newline* is non-`nil`.

The optional fourth argument *window* specifies the window for obtaining parameters such as width, horizontal scrolling, and so on. The default is to use the selected window's parameters.

Like `vertical-motion`, `count-screen-lines` always uses the current buffer, regardless of which buffer is displayed in *window*. This makes possible to use `count-screen-lines` in any buffer, whether or not it is currently displayed in some window.

`move-to-window-line` *count* [Command]

This function moves point with respect to the text currently displayed in the selected window. It moves point to the beginning of the screen line *count* screen lines from the top of the window. If *count* is negative, that specifies a position $-count$ lines from the bottom (or the last line of the buffer, if the buffer ends above the specified screen position).

If *count* is `nil`, then point moves to the beginning of the line in the middle of the window. If the absolute value of *count* is greater than the size of the window, then point moves to the place that would appear on that screen line if the window were tall enough. This will probably cause the next redisplay to scroll to bring that location onto the screen.

In an interactive call, *count* is the numeric prefix argument.

The value returned is the window line number point has moved to, with the top line in the window numbered 0.

`compute-motion` *from frompos to topos width offsets window* [Function]

This function scans the current buffer, calculating screen positions. It scans the buffer forward from position *from*, assuming that is at screen coordinates *frompos*, to position *to* or coordinates *topos*, whichever comes first. It returns the ending buffer position and screen coordinates.

The coordinate arguments *frompos* and *topos* are cons cells of the form `(`*`hpos`*` . `*`vpos`*`)`.

The argument *width* is the number of columns available to display text; this affects handling of continuation lines. `nil` means the actual number of usable text columns in the window, which is equivalent to the value returned by `(window-width window)`.

The argument *offsets* is either `nil` or a cons cell of the form `(`*`hscroll`*` . `*`tab-offset`*`)`. Here *hscroll* is the number of columns not being displayed at the left margin; most callers get this by calling `window-hscroll`. Meanwhile, *tab-offset* is the offset between column numbers on the screen and column numbers in the buffer. This can be nonzero in a continuation line, when the previous screen lines' widths do not add up to a multiple of `tab-width`. It is always zero in a non-continuation line.

The window *window* serves only to specify which display table to use. `compute-motion` always operates on the current buffer, regardless of what buffer is displayed in *window*.

The return value is a list of five elements:

```
(pos hpos vpos prevhpos contin)
```

Here *pos* is the buffer position where the scan stopped, *vpos* is the vertical screen position, and *hpos* is the horizontal screen position.

The result *prevhpos* is the horizontal position one character back from *pos*. The result *contin* is `t` if the last line was continued after (or within) the previous character.

For example, to find the buffer position of column *col* of screen line *line* of a certain window, pass the window's display start location as *from* and the window's upper-left coordinates as *frompos*. Pass the buffer's `(point-max)` as *to*, to limit the scan to the end of the accessible portion of the buffer, and pass *line* and *col* as *topos*. Here's a function that does this:

```
(defun coordinates-of-position (col line)
  (car (compute-motion (window-start)
                       '(0 . 0)
                       (point-max)
                       (cons col line)
```

```
                    (window-width)
                    (cons (window-hscroll) 0)
                    (selected-window))))
```

When you use `compute-motion` for the minibuffer, you need to use `minibuffer-prompt-width` to get the horizontal position of the beginning of the first screen line. See Section 20.12 [Minibuffer Contents], page 312, vol. 1.

30.2.6 Moving over Balanced Expressions

Here are several functions concerned with balanced-parenthesis expressions (also called *sexps* in connection with moving across them in Emacs). The syntax table controls how these functions interpret various characters; see Chapter 35 [Syntax Tables], page 234. See Section 35.6 [Parsing Expressions], page 242, for lower-level primitives for scanning sexps or parts of sexps. For user-level commands, see Section "Commands for Editing with Parentheses" in *The GNU Emacs Manual*.

`forward-list` **&optional** *arg* [Command]
: This function moves forward across *arg* (default 1) balanced groups of parentheses. (Other syntactic entities such as words or paired string quotes are ignored.)

`backward-list` **&optional** *arg* [Command]
: This function moves backward across *arg* (default 1) balanced groups of parentheses. (Other syntactic entities such as words or paired string quotes are ignored.)

`up-list` **&optional** *arg* [Command]
: This function moves forward out of *arg* (default 1) levels of parentheses. A negative argument means move backward but still to a less deep spot.

`down-list` **&optional** *arg* [Command]
: This function moves forward into *arg* (default 1) levels of parentheses. A negative argument means move backward but still go deeper in parentheses ($-$*arg* levels).

`forward-sexp` **&optional** *arg* [Command]
: This function moves forward across *arg* (default 1) balanced expressions. Balanced expressions include both those delimited by parentheses and other kinds, such as words and string constants. See Section 35.6 [Parsing Expressions], page 242. For example,

```
---------- Buffer: foo ----------
(concat* "foo " (car x) y z)
---------- Buffer: foo ----------

(forward-sexp 3)
     ⇒ nil

---------- Buffer: foo ----------
(concat "foo " (car x) y* z)
---------- Buffer: foo ----------
```

`backward-sexp` **&optional** *arg* [Command]
: This function moves backward across *arg* (default 1) balanced expressions.

`beginning-of-defun` **&optional** *arg* [Command]
: This function moves back to the *arg*th beginning of a defun. If *arg* is negative, this actually moves forward, but it still moves to the beginning of a defun, not to the end of one. *arg* defaults to 1.

`end-of-defun` **&optional** *arg* [Command]
: This function moves forward to the *arg*th end of a defun. If *arg* is negative, this actually moves backward, but it still moves to the end of a defun, not to the beginning of one. *arg* defaults to 1.

`defun-prompt-regexp` [User Option]
: If non-`nil`, this buffer-local variable holds a regular expression that specifies what text can appear before the open-parenthesis that starts a defun. That is to say, a defun begins on a line that starts with a match for this regular expression, followed by a character with open-parenthesis syntax.

`open-paren-in-column-0-is-defun-start` [User Option]
: If this variable's value is non-`nil`, an open parenthesis in column 0 is considered to be the start of a defun. If it is `nil`, an open parenthesis in column 0 has no special meaning. The default is `t`.

`beginning-of-defun-function` [Variable]
: If non-`nil`, this variable holds a function for finding the beginning of a defun. The function `beginning-of-defun` calls this function instead of using its normal method, passing it its optional argument. If the argument is non-`nil`, the function should move back by that many functions, like `beginning-of-defun` does.

`end-of-defun-function` [Variable]
: If non-`nil`, this variable holds a function for finding the end of a defun. The function `end-of-defun` calls this function instead of using its normal method.

30.2.7 Skipping Characters

The following two functions move point over a specified set of characters. For example, they are often used to skip whitespace. For related functions, see Section 35.5 [Motion and Syntax], page 241.

These functions convert the set string to multibyte if the buffer is multibyte, and they convert it to unibyte if the buffer is unibyte, as the search functions do (see Chapter 34 [Searching and Matching], page 209).

`skip-chars-forward` *character-set* **&optional** *limit* [Function]
: This function moves point in the current buffer forward, skipping over a given set of characters. It examines the character following point, then advances point if the character matches *character-set*. This continues until it reaches a character that does not match. The function returns the number of characters moved over.

  The argument *character-set* is a string, like the inside of a '`[...]`' in a regular expression except that '`]`' does not terminate it, and '`\`' quotes '`^`', '`-`' or '`\`'. Thus, `"a-zA-Z"` skips over all letters, stopping before the first nonletter, and `"^a-zA-Z"`

skips nonletters stopping before the first letter. See See Section 34.3 [Regular Expressions], page 211. Character classes can also be used, e.g. `"[:alnum:]"`. See see Section 34.3.1.2 [Char Classes], page 215.

If *limit* is supplied (it must be a number or a marker), it specifies the maximum position in the buffer that point can be skipped to. Point will stop at or before *limit*.

In the following example, point is initially located directly before the '`T`'. After the form is evaluated, point is located at the end of that line (between the '`t`' of '`hat`' and the newline). The function skips all letters and spaces, but not newlines.

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(skip-chars-forward "a-zA-Z ")
     ⇒ 18

---------- Buffer: foo ----------
I read "The cat in the hat⋆
comes back" twice.
---------- Buffer: foo ----------
```

`skip-chars-backward` *character-set* **&optional** *limit* [Function]

This function moves point backward, skipping characters that match *character-set*, until *limit*. It is just like `skip-chars-forward` except for the direction of motion.

The return value indicates the distance traveled. It is an integer that is zero or less.

30.3 Excursions

It is often useful to move point "temporarily" within a localized portion of the program. This is called an *excursion*, and it is done with the `save-excursion` special form. This construct remembers the initial identity of the current buffer, and its values of point and the mark, and restores them after the excursion completes. It is the standard way to move point within one part of a program and avoid affecting the rest of the program, and is used thousands of times in the Lisp sources of Emacs.

If you only need to save and restore the identity of the current buffer, use `save-current-buffer` or `with-current-buffer` instead (see Section 27.2 [Current Buffer], page 1). If you need to save or restore window configurations, see the forms described in Section 28.23 [Window Configurations], page 60 and in Section 29.12 [Frame Configurations], page 86.

`save-excursion` *body...* [Special Form]

This special form saves the identity of the current buffer and the values of point and the mark in it, evaluates *body*, and finally restores the buffer and its saved values of point and the mark. All three saved values are restored even in case of an abnormal exit via `throw` or error (see Section 10.5 [Nonlocal Exits], page 126, vol. 1).

The value returned by `save-excursion` is the result of the last form in *body*, or `nil` if no body forms were given.

Because `save-excursion` only saves point and mark for the buffer that was current at the start of the excursion, any changes made to point and/or mark in other buffers, during the excursion, will remain in effect afterward. This frequently leads to unintended consequences, so the byte compiler warns if you call `set-buffer` during an excursion:

```
Warning: Use ‘with-current-buffer’ rather than
         save-excursion+set-buffer
```

To avoid such problems, you should call `save-excursion` only after setting the desired current buffer, as in the following example:

```
(defun append-string-to-buffer (string buffer)
  "Append STRING to the end of BUFFER."
  (with-current-buffer buffer
    (save-excursion
      (goto-char (point-max))
      (insert string))))
```

Likewise, `save-excursion` does not restore window-buffer correspondences altered by functions such as `switch-to-buffer`. One way to restore these correspondences, and the selected window, is to use `save-window-excursion` inside `save-excursion` (see Section 28.23 [Window Configurations], page 60).

Warning: Ordinary insertion of text adjacent to the saved point value relocates the saved value, just as it relocates all markers. More precisely, the saved value is a marker with insertion type `nil`. See Section 31.5 [Marker Insertion Types], page 116. Therefore, when the saved point value is restored, it normally comes before the inserted text.

Although `save-excursion` saves the location of the mark, it does not prevent functions which modify the buffer from setting `deactivate-mark`, and thus causing the deactivation of the mark after the command finishes. See Section 31.7 [The Mark], page 117.

30.4 Narrowing

Narrowing means limiting the text addressable by Emacs editing commands to a limited range of characters in a buffer. The text that remains addressable is called the *accessible portion* of the buffer.

Narrowing is specified with two buffer positions which become the beginning and end of the accessible portion. For most editing commands and most Emacs primitives, these positions replace the values of the beginning and end of the buffer. While narrowing is in effect, no text outside the accessible portion is displayed, and point cannot move outside the accessible portion.

Values such as positions or line numbers, which usually count from the beginning of the buffer, do so despite narrowing, but the functions which use them refuse to operate on text that is inaccessible.

The commands for saving buffers are unaffected by narrowing; they save the entire buffer regardless of any narrowing.

If you need to display in a single buffer several very different types of text, consider using an alternative facility described in Section 27.12 [Swapping Text], page 16.

`narrow-to-region` *start end* [Command]
This function sets the accessible portion of the current buffer to start at *start* and end at *end*. Both arguments should be character positions.

In an interactive call, *start* and *end* are set to the bounds of the current region (point and the mark, with the smallest first).

`narrow-to-page` **&optional** *move-count* [Command]
This function sets the accessible portion of the current buffer to include just the current page. An optional first argument *move-count* non-`nil` means to move forward or backward by *move-count* pages and then narrow to one page. The variable `page-delimiter` specifies where pages start and end (see Section 34.8 [Standard Regexps], page 233).

In an interactive call, *move-count* is set to the numeric prefix argument.

`widen` [Command]
This function cancels any narrowing in the current buffer, so that the entire contents are accessible. This is called *widening*. It is equivalent to the following expression:

```
(narrow-to-region 1 (1+ (buffer-size)))
```

`save-restriction` *body...* [Special Form]
This special form saves the current bounds of the accessible portion, evaluates the *body* forms, and finally restores the saved bounds, thus restoring the same state of narrowing (or absence thereof) formerly in effect. The state of narrowing is restored even in the event of an abnormal exit via `throw` or error (see Section 10.5 [Nonlocal Exits], page 126, vol. 1). Therefore, this construct is a clean way to narrow a buffer temporarily.

The value returned by `save-restriction` is that returned by the last form in *body*, or `nil` if no body forms were given.

Caution: it is easy to make a mistake when using the `save-restriction` construct. Read the entire description here before you try it.

If *body* changes the current buffer, `save-restriction` still restores the restrictions on the original buffer (the buffer whose restrictions it saved from), but it does not restore the identity of the current buffer.

`save-restriction` does *not* restore point and the mark; use `save-excursion` for that. If you use both `save-restriction` and `save-excursion` together, `save-excursion` should come first (on the outside). Otherwise, the old point value would be restored with temporary narrowing still in effect. If the old point value were outside the limits of the temporary narrowing, this would fail to restore it accurately.

Here is a simple example of correct use of `save-restriction`:

```
---------- Buffer: foo ----------
This is the contents of foo
This is the contents of foo
This is the contents of foo*
---------- Buffer: foo ----------
```

```
(save-excursion
  (save-restriction
    (goto-char 1)
    (forward-line 2)
    (narrow-to-region 1 (point))
    (goto-char (point-min))
    (replace-string "foo" "bar")))

---------- Buffer: foo ----------
This is the contents of bar
This is the contents of bar
This is the contents of foo*
---------- Buffer: foo ----------
```

31 Markers

A *marker* is a Lisp object used to specify a position in a buffer relative to the surrounding text. A marker changes its offset from the beginning of the buffer automatically whenever text is inserted or deleted, so that it stays with the two characters on either side of it.

31.1 Overview of Markers

A marker specifies a buffer and a position in that buffer. A marker can be used to represent a position in functions that require one, just as an integer could be used. In that case, the marker's buffer is normally ignored. Of course, a marker used in this way usually points to a position in the buffer that the function operates on, but that is entirely the programmer's responsibility. See Chapter 30 [Positions], page 99, for a complete description of positions.

A marker has three attributes: the marker position, the marker buffer, and the insertion type. The marker position is an integer that is equivalent (at a given time) to the marker as a position in that buffer. But the marker's position value can change during the life of the marker, and often does. Insertion and deletion of text in the buffer relocate the marker. The idea is that a marker positioned between two characters remains between those two characters despite insertion and deletion elsewhere in the buffer. Relocation changes the integer equivalent of the marker.

Deleting text around a marker's position leaves the marker between the characters immediately before and after the deleted text. Inserting text at the position of a marker normally leaves the marker either in front of or after the new text, depending on the marker's *insertion type* (see Section 31.5 [Marker Insertion Types], page 116)—unless the insertion is done with `insert-before-markers` (see Section 32.4 [Insertion], page 126).

Insertion and deletion in a buffer must check all the markers and relocate them if necessary. This slows processing in a buffer with a large number of markers. For this reason, it is a good idea to make a marker point nowhere if you are sure you don't need it any more. Markers that can no longer be accessed are eventually removed (see Section E.3 [Garbage Collection], page 459).

Because it is common to perform arithmetic operations on a marker position, most of these operations (including `+` and `-`) accept markers as arguments. In such cases, the marker stands for its current position.

Here are examples of creating markers, setting markers, and moving point to markers:

```
;; Make a new marker that initially does not point anywhere:
(setq m1 (make-marker))
     ⇒ #<marker in no buffer>

;; Set m1 to point between the 99th and 100th characters
;;    in the current buffer:
(set-marker m1 100)
     ⇒ #<marker at 100 in markers.texi>
```

```
;; Now insert one character at the beginning of the buffer:
(goto-char (point-min))
     ⇒ 1
(insert "Q")
     ⇒ nil

;; m1 is updated appropriately.
m1
     ⇒ #<marker at 101 in markers.texi>

;; Two markers that point to the same position
;;    are not eq, but they are equal.
(setq m2 (copy-marker m1))
     ⇒ #<marker at 101 in markers.texi>
(eq m1 m2)
     ⇒ nil
(equal m1 m2)
     ⇒ t

;; When you are finished using a marker, make it point nowhere.
(set-marker m1 nil)
     ⇒ #<marker in no buffer>
```

31.2 Predicates on Markers

You can test an object to see whether it is a marker, or whether it is either an integer or a marker. The latter test is useful in connection with the arithmetic functions that work with both markers and integers.

`markerp` *object* [Function]
: This function returns `t` if *object* is a marker, `nil` otherwise. Note that integers are not markers, even though many functions will accept either a marker or an integer.

`integer-or-marker-p` *object* [Function]
: This function returns `t` if *object* is an integer or a marker, `nil` otherwise.

`number-or-marker-p` *object* [Function]
: This function returns `t` if *object* is a number (either integer or floating point) or a marker, `nil` otherwise.

31.3 Functions that Create Markers

When you create a new marker, you can make it point nowhere, or point to the present position of point, or to the beginning or end of the accessible portion of the buffer, or to the same place as another given marker.

The next four functions all return markers with insertion type `nil`. See Section 31.5 [Marker Insertion Types], page 116.

`make-marker` [Function]

This function returns a newly created marker that does not point anywhere.

```
(make-marker)
     ⇒ #<marker in no buffer>
```

`point-marker` [Function]

This function returns a new marker that points to the present position of point in the current buffer. See Section 30.1 [Point], page 99. For an example, see `copy-marker`, below.

`point-min-marker` [Function]

This function returns a new marker that points to the beginning of the accessible portion of the buffer. This will be the beginning of the buffer unless narrowing is in effect. See Section 30.4 [Narrowing], page 109.

`point-max-marker` [Function]

This function returns a new marker that points to the end of the accessible portion of the buffer. This will be the end of the buffer unless narrowing is in effect. See Section 30.4 [Narrowing], page 109.

Here are examples of this function and `point-min-marker`, shown in a buffer containing a version of the source file for the text of this chapter.

```
(point-min-marker)
     ⇒ #<marker at 1 in markers.texi>
(point-max-marker)
     ⇒ #<marker at 24080 in markers.texi>

(narrow-to-region 100 200)
     ⇒ nil
(point-min-marker)
     ⇒ #<marker at 100 in markers.texi>
(point-max-marker)
     ⇒ #<marker at 200 in markers.texi>
```

`copy-marker` **&optional** *marker-or-integer insertion-type* [Function]

If passed a marker as its argument, `copy-marker` returns a new marker that points to the same place and the same buffer as does *marker-or-integer*. If passed an integer as its argument, `copy-marker` returns a new marker that points to position *marker-or-integer* in the current buffer.

The new marker's insertion type is specified by the argument *insertion-type*. See Section 31.5 [Marker Insertion Types], page 116.

If passed an integer argument less than 1, `copy-marker` returns a new marker that points to the beginning of the current buffer. If passed an integer argument greater than the length of the buffer, `copy-marker` returns a new marker that points to the end of the buffer.

```
(copy-marker 0)
     ⇒ #<marker at 1 in markers.texi>
```

```
(copy-marker 90000)
     ⇒ #<marker at 24080 in markers.texi>
```

An error is signaled if *marker* is neither a marker nor an integer.

Two distinct markers are considered `equal` (even though not `eq`) to each other if they have the same position and buffer, or if they both point nowhere.

```
(setq p (point-marker))
     ⇒ #<marker at 2139 in markers.texi>

(setq q (copy-marker p))
     ⇒ #<marker at 2139 in markers.texi>

(eq p q)
     ⇒ nil

(equal p q)
     ⇒ t
```

31.4 Information from Markers

This section describes the functions for accessing the components of a marker object.

`marker-position` *marker* [Function]
This function returns the position that *marker* points to, or `nil` if it points nowhere.

`marker-buffer` *marker* [Function]
This function returns the buffer that *marker* points into, or `nil` if it points nowhere.

```
(setq m (make-marker))
     ⇒ #<marker in no buffer>
(marker-position m)
     ⇒ nil
(marker-buffer m)
     ⇒ nil

(set-marker m 3770 (current-buffer))
     ⇒ #<marker at 3770 in markers.texi>
(marker-buffer m)
     ⇒ #<buffer markers.texi>
(marker-position m)
     ⇒ 3770
```

`buffer-has-markers-at` *position* [Function]
This function returns `t` if one or more markers point at position *position* in the current buffer.

31.5 Marker Insertion Types

When you insert text directly at the place where a marker points, there are two possible ways to relocate that marker: it can point before the inserted text, or point after it. You can specify which one a given marker should do by setting its *insertion type*. Note that use of `insert-before-markers` ignores markers' insertion types, always relocating a marker to point after the inserted text.

`set-marker-insertion-type` *marker type* [Function]
: This function sets the insertion type of marker *marker* to *type*. If *type* is `t`, *marker* will advance when text is inserted at its position. If *type* is `nil`, *marker* does not advance when text is inserted there.

`marker-insertion-type` *marker* [Function]
: This function reports the current insertion type of *marker*.

Most functions that create markers, without an argument allowing to specify the insertion type, create them with insertion type `nil`. Also, the mark has, by default, insertion type `nil`.

31.6 Moving Marker Positions

This section describes how to change the position of an existing marker. When you do this, be sure you know whether the marker is used outside of your program, and, if so, what effects will result from moving it—otherwise, confusing things may happen in other parts of Emacs.

`set-marker` *marker position* **&optional** *buffer* [Function]
: This function moves *marker* to *position* in *buffer*. If *buffer* is not provided, it defaults to the current buffer.

  If *position* is less than 1, `set-marker` moves *marker* to the beginning of the buffer. If *position* is greater than the size of the buffer, `set-marker` moves marker to the end of the buffer. If *position* is `nil` or a marker that points nowhere, then *marker* is set to point nowhere.

  The value returned is *marker*.

```
(setq m (point-marker))
     ⇒ #<marker at 4714 in markers.texi>
(set-marker m 55)
     ⇒ #<marker at 55 in markers.texi>
(setq b (get-buffer "foo"))
     ⇒ #<buffer foo>
(set-marker m 0 b)
     ⇒ #<marker at 1 in foo>
```

`move-marker` *marker position* **&optional** *buffer* [Function]
: This is another name for `set-marker`.

31.7 The Mark

Each buffer has a special marker, which is designated *the mark*. When a buffer is newly created, this marker exists but does not point anywhere; this means that the mark "doesn't exist" in that buffer yet. Subsequent commands can set the mark.

The mark specifies a position to bound a range of text for many commands, such as `kill-region` and `indent-rigidly`. These commands typically act on the text between point and the mark, which is called the *region*. If you are writing a command that operates on the region, don't examine the mark directly; instead, use `interactive` with the '`r`' specification. This provides the values of point and the mark as arguments to the command in an interactive call, but permits other Lisp programs to specify arguments explicitly. See Section 21.2.2 [Interactive Codes], page 318, vol. 1.

Some commands set the mark as a side-effect. Commands should do this only if it has a potential use to the user, and never for their own internal purposes. For example, the `replace-regexp` command sets the mark to the value of point before doing any replacements, because this enables the user to move back there conveniently after the replace is finished.

Once the mark "exists" in a buffer, it normally never ceases to exist. However, it may become *inactive*, if Transient Mark mode is enabled. The buffer-local variable `mark-active`, if non-`nil`, means that the mark is active. A command can call the function `deactivate-mark` to deactivate the mark directly, or it can request deactivation of the mark upon return to the editor command loop by setting the variable `deactivate-mark` to a non-`nil` value.

If Transient Mark mode is enabled, certain editing commands that normally apply to text near point, apply instead to the region when the mark is active. This is the main motivation for using Transient Mark mode. (Another is that this enables highlighting of the region when the mark is active. See Chapter 38 [Display], page 299.)

In addition to the mark, each buffer has a *mark ring* which is a list of markers containing previous values of the mark. When editing commands change the mark, they should normally save the old value of the mark on the mark ring. The variable `mark-ring-max` specifies the maximum number of entries in the mark ring; once the list becomes this long, adding a new element deletes the last element.

There is also a separate global mark ring, but that is used only in a few particular user-level commands, and is not relevant to Lisp programming. So we do not describe it here.

`mark` **&optional** *force* [Function]

This function returns the current buffer's mark position as an integer, or `nil` if no mark has ever been set in this buffer.

If Transient Mark mode is enabled, and `mark-even-if-inactive` is `nil`, `mark` signals an error if the mark is inactive. However, if *force* is non-`nil`, then `mark` disregards inactivity of the mark, and returns the mark position (or `nil`) anyway.

`mark-marker` [Function]

This function returns the marker that represents the current buffer's mark. It is not a copy, it is the marker used internally. Therefore, changing this marker's position will directly affect the buffer's mark. Don't do that unless that is the effect you want.

```
(setq m (mark-marker))
     ⇒ #<marker at 3420 in markers.texi>
(set-marker m 100)
     ⇒ #<marker at 100 in markers.texi>
(mark-marker)
     ⇒ #<marker at 100 in markers.texi>
```

Like any marker, this marker can be set to point at any buffer you like. If you make it point at any buffer other than the one of which it is the mark, it will yield perfectly consistent, but rather odd, results. We recommend that you not do it!

`set-mark` *position* [Function]

This function sets the mark to *position*, and activates the mark. The old value of the mark is *not* pushed onto the mark ring.

Please note: Use this function only if you want the user to see that the mark has moved, and you want the previous mark position to be lost. Normally, when a new mark is set, the old one should go on the `mark-ring`. For this reason, most applications should use `push-mark` and `pop-mark`, not `set-mark`.

Novice Emacs Lisp programmers often try to use the mark for the wrong purposes. The mark saves a location for the user's convenience. An editing command should not alter the mark unless altering the mark is part of the user-level functionality of the command. (And, in that case, this effect should be documented.) To remember a location for internal use in the Lisp program, store it in a Lisp variable. For example:

```
(let ((beg (point)))
  (forward-line 1)
  (delete-region beg (point))).
```

`push-mark` **&optional** *position nomsg activate* [Function]

This function sets the current buffer's mark to *position*, and pushes a copy of the previous mark onto `mark-ring`. If *position* is `nil`, then the value of point is used.

The function `push-mark` normally *does not* activate the mark. To do that, specify `t` for the argument *activate*.

A '`Mark set`' message is displayed unless *nomsg* is non-`nil`.

`pop-mark` [Function]

This function pops off the top element of `mark-ring` and makes that mark become the buffer's actual mark. This does not move point in the buffer, and it does nothing if `mark-ring` is empty. It deactivates the mark.

`transient-mark-mode` [User Option]

This variable, if non-`nil`, enables Transient Mark mode. In Transient Mark mode, every buffer-modifying primitive sets `deactivate-mark`. As a consequence, most commands that modify the buffer also deactivate the mark.

When Transient Mark mode is enabled and the mark is active, many commands that normally apply to the text near point instead apply to the region. Such commands should use the function `use-region-p` to test whether they should operate on the region. See Section 31.8 [The Region], page 120.

Lisp programs can set `transient-mark-mode` to non-`nil`, non-`t` values to enable Transient Mark mode temporarily. If the value is `lambda`, Transient Mark mode is automatically turned off after any action, such as buffer modification, that would normally deactivate the mark. If the value is `(only . oldval)`, then `transient-mark-mode` is set to the value *oldval* after any subsequent command that moves point and is not shift-translated (see Section 21.8.1 [Key Sequence Input], page 342, vol. 1), or after any other action that would normally deactivate the mark.

`mark-even-if-inactive` [User Option]
If this is non-`nil`, Lisp programs and the Emacs user can use the mark even when it is inactive. This option affects the behavior of Transient Mark mode. When the option is non-`nil`, deactivation of the mark turns off region highlighting, but commands that use the mark behave as if the mark were still active.

`deactivate-mark` [Variable]
If an editor command sets this variable non-`nil`, then the editor command loop deactivates the mark after the command returns (if Transient Mark mode is enabled). All the primitives that change the buffer set `deactivate-mark`, to deactivate the mark when the command is finished.

To write Lisp code that modifies the buffer without causing deactivation of the mark at the end of the command, bind `deactivate-mark` to `nil` around the code that does the modification. For example:

```
(let (deactivate-mark)
  (insert " "))
```

`deactivate-mark` **&optional** *force* [Function]
If Transient Mark mode is enabled or *force* is non-`nil`, this function deactivates the mark and runs the normal hook `deactivate-mark-hook`. Otherwise, it does nothing.

`mark-active` [Variable]
The mark is active when this variable is non-`nil`. This variable is always buffer-local in each buffer. Do *not* use the value of this variable to decide whether a command that normally operates on text near point should operate on the region instead. Use the function `use-region-p` for that (see Section 31.8 [The Region], page 120).

`activate-mark-hook` [Variable]
`deactivate-mark-hook` [Variable]
These normal hooks are run, respectively, when the mark becomes active and when it becomes inactive. The hook `activate-mark-hook` is also run at the end of the command loop if the mark is active and it is possible that the region may have changed.

`handle-shift-selection` [Function]
This function implements the "shift-selection" behavior of point-motion commands. See Section "Shift Selection" in *The GNU Emacs Manual*. It is called automatically by the Emacs command loop whenever a command with a '`^`' character in its `interactive` spec is invoked, before the command itself is executed (see Section 21.2.2 [Interactive Codes], page 318, vol. 1).

If `shift-select-mode` is non-`nil` and the current command was invoked via shift translation (see Section 21.8.1 [Key Sequence Input], page 342, vol. 1), this function sets the mark and temporarily activates the region, unless the region was already temporarily activated in this way. Otherwise, if the region has been activated temporarily, it deactivates the mark and restores the variable `transient-mark-mode` to its earlier value.

`mark-ring` [Variable]

The value of this buffer-local variable is the list of saved former marks of the current buffer, most recent first.

```
mark-ring
⇒ (#<marker at 11050 in markers.texi>
    #<marker at 10832 in markers.texi>
    ...)
```

`mark-ring-max` [User Option]

The value of this variable is the maximum size of `mark-ring`. If more marks than this are pushed onto the `mark-ring`, `push-mark` discards an old mark when it adds a new one.

31.8 The Region

The text between point and the mark is known as *the region*. Various functions operate on text delimited by point and the mark, but only those functions specifically related to the region itself are described here.

The next two functions signal an error if the mark does not point anywhere. If Transient Mark mode is enabled and `mark-even-if-inactive` is `nil`, they also signal an error if the mark is inactive.

`region-beginning` [Function]

This function returns the position of the beginning of the region (as an integer). This is the position of either point or the mark, whichever is smaller.

`region-end` [Function]

This function returns the position of the end of the region (as an integer). This is the position of either point or the mark, whichever is larger.

Instead of using `region-beginning` and `region-end`, a command designed to operate on a region should normally use `interactive` with the '`r`' specification to find the beginning and end of the region. This lets other Lisp programs specify the bounds explicitly as arguments. See Section 21.2.2 [Interactive Codes], page 318, vol. 1.

`use-region-p` [Function]

This function returns `t` if Transient Mark mode is enabled, the mark is active, and there is a valid region in the buffer. This function is intended to be used by commands that operate on the region, instead of on text near point, when the mark is active.

A region is valid if it has a non-zero size, or if the user option `use-empty-active-region` is non-`nil` (by default, it is `nil`). The function `region-active-p` is similar to `use-region-p`, but considers all regions as valid. In most cases, you should not

use `region-active-p`, since if the region is empty it is often more appropriate to operate on point.

32 Text

This chapter describes the functions that deal with the text in a buffer. Most examine, insert, or delete text in the current buffer, often operating at point or on text adjacent to point. Many are interactive. All the functions that change the text provide for undoing the changes (see Section 32.9 [Undo], page 137).

Many text-related functions operate on a region of text defined by two buffer positions passed in arguments named *start* and *end*. These arguments should be either markers (see Chapter 31 [Markers], page 112) or numeric character positions (see Chapter 30 [Positions], page 99). The order of these arguments does not matter; it is all right for *start* to be the end of the region and *end* the beginning. For example, `(delete-region 1 10)` and `(delete-region 10 1)` are equivalent. An `args-out-of-range` error is signaled if either *start* or *end* is outside the accessible portion of the buffer. In an interactive call, point and the mark are used for these arguments.

Throughout this chapter, "text" refers to the characters in the buffer, together with their properties (when relevant). Keep in mind that point is always between two characters, and the cursor appears on the character after point.

32.1 Examining Text Near Point

Many functions are provided to look at the characters around point. Several simple functions are described here. See also `looking-at` in Section 34.4 [Regexp Search], page 221.

In the following four functions, "beginning" or "end" of buffer refers to the beginning or end of the accessible portion.

`char-after` **&optional** *position* [Function]

This function returns the character in the current buffer at (i.e., immediately after) position *position*. If *position* is out of range for this purpose, either before the beginning of the buffer, or at or beyond the end, then the value is `nil`. The default for *position* is point.

In the following example, assume that the first character in the buffer is '`@`':

```
(string (char-after 1))
     ⇒ "@"
```

`char-before` **&optional** *position* [Function]

This function returns the character in the current buffer immediately before position *position*. If *position* is out of range for this purpose, either at or before the beginning of the buffer, or beyond the end, then the value is `nil`. The default for *position* is point.

`following-char` [Function]

This function returns the character following point in the current buffer. This is similar to `(char-after (point))`. However, if point is at the end of the buffer, then `following-char` returns 0.

Remember that point is always between characters, and the cursor normally appears over the character following point. Therefore, the character returned by `following-char` is the character the cursor is over.

In this example, point is between the '`a`' and the '`c`'.

```
---------- Buffer: foo ----------
Gentlemen may cry ‘‘Pea⋆ce! Peace!,’’
but there is no peace.
---------- Buffer: foo ----------

(string (preceding-char))
     ⇒ "a"
(string (following-char))
     ⇒ "c"
```

`preceding-char` [Function]
: This function returns the character preceding point in the current buffer. See above, under `following-char`, for an example. If point is at the beginning of the buffer, `preceding-char` returns 0.

`bobp` [Function]
: This function returns `t` if point is at the beginning of the buffer. If narrowing is in effect, this means the beginning of the accessible portion of the text. See also `point-min` in Section 30.1 [Point], page 99.

`eobp` [Function]
: This function returns `t` if point is at the end of the buffer. If narrowing is in effect, this means the end of accessible portion of the text. See also `point-max` in See Section 30.1 [Point], page 99.

`bolp` [Function]
: This function returns `t` if point is at the beginning of a line. See Section 30.2.4 [Text Lines], page 102. The beginning of the buffer (or of its accessible portion) always counts as the beginning of a line.

`eolp` [Function]
: This function returns `t` if point is at the end of a line. The end of the buffer (or of its accessible portion) is always considered the end of a line.

32.2 Examining Buffer Contents

This section describes functions that allow a Lisp program to convert any portion of the text in the buffer into a string.

`buffer-substring` *start end* [Function]
: This function returns a string containing a copy of the text of the region defined by positions *start* and *end* in the current buffer. If the arguments are not positions in the accessible portion of the buffer, `buffer-substring` signals an `args-out-of-range` error.

  Here's an example which assumes Font-Lock mode is not enabled:

```
---------- Buffer: foo ----------
This is the contents of buffer foo

---------- Buffer: foo ----------
```

```
(buffer-substring 1 10)
     ⇒ "This is t"
(buffer-substring (point-max) 10)
     ⇒ "he contents of buffer foo\n"
```

If the text being copied has any text properties, these are copied into the string along with the characters they belong to. See Section 32.19 [Text Properties], page 156. However, overlays (see Section 38.9 [Overlays], page 315) in the buffer and their properties are ignored, not copied.

For example, if Font-Lock mode is enabled, you might get results like these:

```
(buffer-substring 1 10)
     ⇒ #("This is t" 0 1 (fontified t) 1 9 (fontified t))
```

`buffer-substring-no-properties` *start end* [Function]

This is like `buffer-substring`, except that it does not copy text properties, just the characters themselves. See Section 32.19 [Text Properties], page 156.

`buffer-string` [Function]

This function returns the contents of the entire accessible portion of the current buffer, as a string.

`filter-buffer-substring` *start end* **&optional** *delete* [Function]

This function passes the buffer text between *start* and *end* through the filter functions specified by the wrapper hook `filter-buffer-substring-functions`, and returns the result. The obsolete variable `buffer-substring-filters` is also consulted. If both of these variables are `nil`, the value is the unaltered text from the buffer, i.e. what `buffer-substring` would return.

If *delete* is non-`nil`, this function deletes the text between *start* and *end* after copying it, like `delete-and-extract-region`.

Lisp code should use this function instead of `buffer-substring`, `buffer-substring-no-properties`, or `delete-and-extract-region` when copying into user-accessible data structures such as the kill-ring, X clipboard, and registers. Major and minor modes can add functions to `filter-buffer-substring-functions` to alter such text as it is copied out of the buffer.

`filter-buffer-substring-functions` [Variable]

This variable is a wrapper hook (see Section 23.1.1 [Running Hooks], page 396, vol. 1), whose members should be functions that accept four arguments: *fun*, *start*, *end*, and *delete*. *fun* is a function that takes three arguments (*start*, *end*, and *delete*), and returns a string. In both cases, the *start*, *end*, and *delete* arguments are the same as those of `filter-buffer-substring`.

The first hook function is passed a *fun* that is equivalent to the default operation of `filter-buffer-substring`, i.e. it returns the buffer-substring between *start* and *end* (processed by any `buffer-substring-filters`) and optionally deletes the original text from the buffer. In most cases, the hook function will call *fun* once, and then do its own processing of the result. The next hook function receives a *fun* equivalent to this, and so on. The actual return value is the result of all the hook functions acting in sequence.

`buffer-substring-filters` [Variable]
: This variable is obsoleted by `filter-buffer-substring-functions`, but is still supported for backward compatibility. Its value should should be a list of functions which accept a single string argument and return another string. `filter-buffer-substring` passes the buffer substring to the first function in this list, and the return value of each function is passed to the next function. The return value of the last function is passed to `filter-buffer-substring-functions`.

`current-word` **&optional** *strict really-word* [Function]
: This function returns the symbol (or word) at or near point, as a string. The return value includes no text properties.

  If the optional argument *really-word* is non-`nil`, it finds a word; otherwise, it finds a symbol (which includes both word characters and symbol constituent characters).

  If the optional argument *strict* is non-`nil`, then point must be in or next to the symbol or word—if no symbol or word is there, the function returns `nil`. Otherwise, a nearby symbol or word on the same line is acceptable.

`thing-at-point` *thing* [Function]
: Return the *thing* around or next to point, as a string.

  The argument *thing* is a symbol which specifies a kind of syntactic entity. Possibilities include `symbol`, `list`, `sexp`, `defun`, `filename`, `url`, `word`, `sentence`, `whitespace`, `line`, `page`, and others.

```
---------- Buffer: foo ----------
Gentlemen may cry ``Pea*ce! Peace!,''
but there is no peace.
---------- Buffer: foo ----------

(thing-at-point 'word)
     ⇒ "Peace"
(thing-at-point 'line)
     ⇒ "Gentlemen may cry ``Peace! Peace!,''\n"
(thing-at-point 'whitespace)
     ⇒ nil
```

32.3 Comparing Text

This function lets you compare portions of the text in a buffer, without copying them into strings first.

`compare-buffer-substrings` *buffer1 start1 end1 buffer2 start2 end2* [Function]
: This function lets you compare two substrings of the same buffer or two different buffers. The first three arguments specify one substring, giving a buffer (or a buffer name) and two positions within the buffer. The last three arguments specify the other substring in the same way. You can use `nil` for *buffer1*, *buffer2*, or both to stand for the current buffer.

  The value is negative if the first substring is less, positive if the first is greater, and zero if they are equal. The absolute value of the result is one plus the index of the first differing characters within the substrings.

This function ignores case when comparing characters if `case-fold-search` is non-`nil`. It always ignores text properties.

Suppose the current buffer contains the text '`foobarbar haha!rara!`'; then in this example the two substrings are '`rbar `' and '`rara!`'. The value is 2 because the first substring is greater at the second character.

```
(compare-buffer-substrings nil 6 11 nil 16 21)
     ⇒ 2
```

32.4 Inserting Text

Insertion means adding new text to a buffer. The inserted text goes at point—between the character before point and the character after point. Some insertion functions leave point before the inserted text, while other functions leave it after. We call the former insertion *after point* and the latter insertion *before point*.

Insertion moves markers located at positions after the insertion point, so that they stay with the surrounding text (see Chapter 31 [Markers], page 112). When a marker points at the place of insertion, insertion may or may not relocate the marker, depending on the marker's insertion type (see Section 31.5 [Marker Insertion Types], page 116). Certain special functions such as `insert-before-markers` relocate all such markers to point after the inserted text, regardless of the markers' insertion type.

Insertion functions signal an error if the current buffer is read-only or if they insert within read-only text.

These functions copy text characters from strings and buffers along with their properties. The inserted characters have exactly the same properties as the characters they were copied from. By contrast, characters specified as separate arguments, not part of a string or buffer, inherit their text properties from the neighboring text.

The insertion functions convert text from unibyte to multibyte in order to insert in a multibyte buffer, and vice versa—if the text comes from a string or from a buffer. However, they do not convert unibyte character codes 128 through 255 to multibyte characters, not even if the current buffer is a multibyte buffer. See Section 33.2 [Converting Representations], page 183.

`insert` **&rest** *args* [Function]

This function inserts the strings and/or characters *args* into the current buffer, at point, moving point forward. In other words, it inserts the text before point. An error is signaled unless all *args* are either strings or characters. The value is `nil`.

`insert-before-markers` **&rest** *args* [Function]

This function inserts the strings and/or characters *args* into the current buffer, at point, moving point forward. An error is signaled unless all *args* are either strings or characters. The value is `nil`.

This function is unlike the other insertion functions in that it relocates markers initially pointing at the insertion point, to point after the inserted text. If an overlay begins at the insertion point, the inserted text falls outside the overlay; if a nonempty overlay ends at the insertion point, the inserted text falls inside that overlay.

`insert-char` *character count* **&optional** *inherit* [Function]
This function inserts *count* instances of *character* into the current buffer before point. The argument *count* should be an integer, and *character* must be a character. The value is `nil`.

This function does not convert unibyte character codes 128 through 255 to multibyte characters, not even if the current buffer is a multibyte buffer. See Section 33.2 [Converting Representations], page 183.

If *inherit* is non-`nil`, then the inserted characters inherit sticky text properties from the two characters before and after the insertion point. See Section 32.19.6 [Sticky Properties], page 167.

`insert-buffer-substring` *from-buffer-or-name* **&optional** *start end* [Function]
This function inserts a portion of buffer *from-buffer-or-name* (which must already exist) into the current buffer before point. The text inserted is the region between *start* and *end*. (These arguments default to the beginning and end of the accessible portion of that buffer.) This function returns `nil`.

In this example, the form is executed with buffer '`bar`' as the current buffer. We assume that buffer '`bar`' is initially empty.

```
---------- Buffer: foo ----------
We hold these truths to be self-evident, that all
---------- Buffer: foo ----------

(insert-buffer-substring "foo" 1 20)
     ⇒ nil

---------- Buffer: bar ----------
We hold these truth*
---------- Buffer: bar ----------
```

`insert-buffer-substring-no-properties` *from-buffer-or-name* [Function]
&optional *start end*
This is like `insert-buffer-substring` except that it does not copy any text properties.

See Section 32.19.6 [Sticky Properties], page 167, for other insertion functions that inherit text properties from the nearby text in addition to inserting it. Whitespace inserted by indentation functions also inherits text properties.

32.5 User-Level Insertion Commands

This section describes higher-level commands for inserting text, commands intended primarily for the user but useful also in Lisp programs.

`insert-buffer` *from-buffer-or-name* [Command]
This command inserts the entire accessible contents of *from-buffer-or-name* (which must exist) into the current buffer after point. It leaves the mark after the inserted text. The value is `nil`.

`self-insert-command` *count* [Command]
: This command inserts the last character typed; it does so *count* times, before point, and returns `nil`. Most printing characters are bound to this command. In routine use, `self-insert-command` is the most frequently called function in Emacs, but programs rarely use it except to install it on a keymap.

  In an interactive call, *count* is the numeric prefix argument.

  Self-insertion translates the input character through `translation-table-for-input`. See Section 33.8 [Translation of Characters], page 191.

  This command calls `auto-fill-function` whenever that is non-`nil` and the character inserted is in the table `auto-fill-chars` (see Section 32.14 [Auto Filling], page 146).

  This command performs abbrev expansion if Abbrev mode is enabled and the inserted character does not have word-constituent syntax. (See Chapter 36 [Abbrevs], page 250, and Section 35.2.1 [Syntax Class Table], page 235.) It is also responsible for calling `blink-paren-function` when the inserted character has close parenthesis syntax (see Section 38.19 [Blinking], page 375).

  The final thing this command does is to run the hook `post-self-insert-hook`. You could use this to automatically reindent text as it is typed, for example.

  Do not try substituting your own definition of `self-insert-command` for the standard one. The editor command loop handles this function specially.

`newline` **&optional** *number-of-newlines* [Command]
: This command inserts newlines into the current buffer before point. If *number-of-newlines* is supplied, that many newline characters are inserted.

  This function calls `auto-fill-function` if the current column number is greater than the value of `fill-column` and *number-of-newlines* is `nil`. Typically what `auto-fill-function` does is insert a newline; thus, the overall result in this case is to insert two newlines at different places: one at point, and another earlier in the line. `newline` does not auto-fill if *number-of-newlines* is non-`nil`.

  This command indents to the left margin if that is not zero. See Section 32.12 [Margins], page 143.

  The value returned is `nil`. In an interactive call, *count* is the numeric prefix argument.

`overwrite-mode` [Variable]
: This variable controls whether overwrite mode is in effect. The value should be `overwrite-mode-textual`, `overwrite-mode-binary`, or `nil`. `overwrite-mode-textual` specifies textual overwrite mode (treats newlines and tabs specially), and `overwrite-mode-binary` specifies binary overwrite mode (treats newlines and tabs like any other characters).

32.6 Deleting Text

Deletion means removing part of the text in a buffer, without saving it in the kill ring (see Section 32.8 [The Kill Ring], page 132). Deleted text can't be yanked, but can be reinserted using the undo mechanism (see Section 32.9 [Undo], page 137). Some deletion functions do save text in the kill ring in some special cases.

All of the deletion functions operate on the current buffer.

`erase-buffer` [Command]

This function deletes the entire text of the current buffer (*not* just the accessible portion), leaving it empty. If the buffer is read-only, it signals a `buffer-read-only` error; if some of the text in it is read-only, it signals a `text-read-only` error. Otherwise, it deletes the text without asking for any confirmation. It returns `nil`.

Normally, deleting a large amount of text from a buffer inhibits further auto-saving of that buffer "because it has shrunk". However, `erase-buffer` does not do this, the idea being that the future text is not really related to the former text, and its size should not be compared with that of the former text.

`delete-region` *start end* [Command]

This command deletes the text between positions *start* and *end* in the current buffer, and returns `nil`. If point was inside the deleted region, its value afterward is *start*. Otherwise, point relocates with the surrounding text, as markers do.

`delete-and-extract-region` *start end* [Function]

This function deletes the text between positions *start* and *end* in the current buffer, and returns a string containing the text just deleted.

If point was inside the deleted region, its value afterward is *start*. Otherwise, point relocates with the surrounding text, as markers do.

`delete-char` *count* **&optional** *killp* [Command]

This command deletes *count* characters directly after point, or before point if *count* is negative. If *killp* is non-`nil`, then it saves the deleted characters in the kill ring.

In an interactive call, *count* is the numeric prefix argument, and *killp* is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

`delete-backward-char` *count* **&optional** *killp* [Command]

This command deletes *count* characters directly before point, or after point if *count* is negative. If *killp* is non-`nil`, then it saves the deleted characters in the kill ring.

In an interactive call, *count* is the numeric prefix argument, and *killp* is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

`backward-delete-char-untabify` *count* **&optional** *killp* [Command]

This command deletes *count* characters backward, changing tabs into spaces. When the next character to be deleted is a tab, it is first replaced with the proper number of spaces to preserve alignment and then one of those spaces is deleted instead of the tab. If *killp* is non-`nil`, then the command saves the deleted characters in the kill ring.

Conversion of tabs to spaces happens only if *count* is positive. If it is negative, exactly $-$*count* characters after point are deleted.

In an interactive call, *count* is the numeric prefix argument, and *killp* is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

`backward-delete-char-untabify-method` [User Option]

This option specifies how `backward-delete-char-untabify` should deal with whitespace. Possible values include `untabify`, the default, meaning convert a tab to many spaces and delete one; `hungry`, meaning delete all tabs and spaces before point with one command; `all` meaning delete all tabs, spaces and newlines before point, and `nil`, meaning do nothing special for whitespace characters.

32.7 User-Level Deletion Commands

This section describes higher-level commands for deleting text, commands intended primarily for the user but useful also in Lisp programs.

`delete-horizontal-space` **&optional** *backward-only* [Command]

This function deletes all spaces and tabs around point. It returns `nil`.

If *backward-only* is non-`nil`, the function deletes spaces and tabs before point, but not after point.

In the following examples, we call `delete-horizontal-space` four times, once on each line, with point between the second and third characters on the line each time.

```
---------- Buffer: foo ----------
I ⋆thought
I ⋆     thought
We⋆ thought
Yo⋆u thought
---------- Buffer: foo ----------

(delete-horizontal-space)   ; Four times.
     ⇒ nil

---------- Buffer: foo ----------
Ithought
Ithought
Wethought
You thought
---------- Buffer: foo ----------
```

`delete-indentation` **&optional** *join-following-p* [Command]

This function joins the line point is on to the previous line, deleting any whitespace at the join and in some cases replacing it with one space. If *join-following-p* is non-`nil`, `delete-indentation` joins this line to the following line instead. The function returns `nil`.

If there is a fill prefix, and the second of the lines being joined starts with the prefix, then `delete-indentation` deletes the fill prefix before joining the lines. See Section 32.12 [Margins], page 143.

In the example below, point is located on the line starting '`events`', and it makes no difference if there are trailing spaces in the preceding line.

```
---------- Buffer: foo ----------
When in the course of human
*    events, it becomes necessary
---------- Buffer: foo ----------

(delete-indentation)
     ⇒ nil

---------- Buffer: foo ----------
When in the course of human* events, it becomes necessary
---------- Buffer: foo ----------
```

After the lines are joined, the function `fixup-whitespace` is responsible for deciding whether to leave a space at the junction.

`fixup-whitespace` [Command]

This function replaces all the horizontal whitespace surrounding point with either one space or no space, according to the context. It returns `nil`.

At the beginning or end of a line, the appropriate amount of space is none. Before a character with close parenthesis syntax, or after a character with open parenthesis or expression-prefix syntax, no space is also appropriate. Otherwise, one space is appropriate. See Section 35.2.1 [Syntax Class Table], page 235.

In the example below, `fixup-whitespace` is called the first time with point before the word '`spaces`' in the first line. For the second invocation, point is directly after the '`(`'.

```
---------- Buffer: foo ----------
This has too many     *spaces
This has too many spaces at the start of (*   this list)
---------- Buffer: foo ----------

(fixup-whitespace)
     ⇒ nil
(fixup-whitespace)
     ⇒ nil

---------- Buffer: foo ----------
This has too many spaces
This has too many spaces at the start of (this list)
---------- Buffer: foo ----------
```

`just-one-space` **&optional** *n* [Command]

This command replaces any spaces and tabs around point with a single space, or *n* spaces if *n* is specified. It returns `nil`.

`delete-blank-lines` [Command]

This function deletes blank lines surrounding point. If point is on a blank line with one or more blank lines before or after it, then all but one of them are deleted. If

point is on an isolated blank line, then it is deleted. If point is on a nonblank line, the command deletes all blank lines immediately following it.

A blank line is defined as a line containing only tabs and spaces.

`delete-blank-lines` returns `nil`.

32.8 The Kill Ring

Kill functions delete text like the deletion functions, but save it so that the user can reinsert it by *yanking*. Most of these functions have '`kill-`' in their name. By contrast, the functions whose names start with '`delete-`' normally do not save text for yanking (though they can still be undone); these are "deletion" functions.

Most of the kill commands are primarily for interactive use, and are not described here. What we do describe are the functions provided for use in writing such commands. You can use these functions to write commands for killing text. When you need to delete text for internal purposes within a Lisp function, you should normally use deletion functions, so as not to disturb the kill ring contents. See Section 32.6 [Deletion], page 128.

Killed text is saved for later yanking in the *kill ring*. This is a list that holds a number of recent kills, not just the last text kill. We call this a "ring" because yanking treats it as having elements in a cyclic order. The list is kept in the variable `kill-ring`, and can be operated on with the usual functions for lists; there are also specialized functions, described in this section, that treat it as a ring.

Some people think this use of the word "kill" is unfortunate, since it refers to operations that specifically *do not* destroy the entities "killed". This is in sharp contrast to ordinary life, in which death is permanent and "killed" entities do not come back to life. Therefore, other metaphors have been proposed. For example, the term "cut ring" makes sense to people who, in pre-computer days, used scissors and paste to cut up and rearrange manuscripts. However, it would be difficult to change the terminology now.

32.8.1 Kill Ring Concepts

The kill ring records killed text as strings in a list, most recent first. A short kill ring, for example, might look like this:

```
("some text" "a different piece of text" "even older text")
```

When the list reaches `kill-ring-max` entries in length, adding a new entry automatically deletes the last entry.

When kill commands are interwoven with other commands, each kill command makes a new entry in the kill ring. Multiple kill commands in succession build up a single kill ring entry, which would be yanked as a unit; the second and subsequent consecutive kill commands add text to the entry made by the first one.

For yanking, one entry in the kill ring is designated the "front" of the ring. Some yank commands "rotate" the ring by designating a different element as the "front". But this virtual rotation doesn't change the list itself—the most recent entry always comes first in the list.

32.8.2 Functions for Killing

`kill-region` is the usual subroutine for killing text. Any command that calls this function is a "kill command" (and should probably have '`kill`' in its name). `kill-region` puts the

newly killed text in a new element at the beginning of the kill ring or adds it to the most recent element. It determines automatically (using `last-command`) whether the previous command was a kill command, and if so appends the killed text to the most recent entry.

`kill-region` *start end* [Command]
: This function kills the text in the region defined by *start* and *end*. The text is deleted but saved in the kill ring, along with its text properties. The value is always `nil`.

  In an interactive call, *start* and *end* are point and the mark.

  If the buffer or text is read-only, `kill-region` modifies the kill ring just the same, then signals an error without modifying the buffer. This is convenient because it lets the user use a series of kill commands to copy text from a read-only buffer into the kill ring.

`kill-read-only-ok` [User Option]
: If this option is non-`nil`, `kill-region` does not signal an error if the buffer or text is read-only. Instead, it simply returns, updating the kill ring but not changing the buffer.

`copy-region-as-kill` *start end* [Command]
: This command saves the region defined by *start* and *end* on the kill ring (including text properties), but does not delete the text from the buffer. It returns `nil`.

  The command does not set `this-command` to `kill-region`, so a subsequent kill command does not append to the same kill ring entry.

  In Lisp programs, it is better to use `kill-new` or `kill-append` instead of this command. See Section 32.8.5 [Low-Level Kill Ring], page 135.

32.8.3 Yanking

Yanking means inserting text from the kill ring, but it does not insert the text blindly. Yank commands and some other commands use `insert-for-yank` to perform special processing on the text that they copy into the buffer.

`insert-for-yank` *string* [Function]
: This function normally works like `insert` except that it doesn't insert the text properties (see Section 32.19 [Text Properties], page 156) in the list variable `yank-excluded-properties`. However, if any part of *string* has a non-`nil` `yank-handler` text property, that property can do various special processing on that part of the text being inserted.

`insert-buffer-substring-as-yank` *buf* **&optional** *start end* [Function]
: This function resembles `insert-buffer-substring` except that it doesn't insert the text properties in the `yank-excluded-properties` list.

You can put a `yank-handler` text property on all or part of the text to control how it will be inserted if it is yanked. The `insert-for-yank` function looks for that property. The property value must be a list of one to four elements, with the following format (where elements after the first may be omitted):

```
(function param noexclude undo)
```

Here is what the elements do:

function When *function* is present and non-`nil`, it is called instead of `insert` to insert the string. *function* takes one argument—the string to insert.

param If *param* is present and non-`nil`, it replaces *string* (or the part of *string* being processed) as the object passed to *function* (or `insert`); for example, if *function* is `yank-rectangle`, *param* should be a list of strings to insert as a rectangle.

noexclude If *noexclude* is present and non-`nil`, the normal removal of the yank-excluded-properties is not performed; instead *function* is responsible for removing those properties. This may be necessary if *function* adjusts point before or after inserting the object.

undo If *undo* is present and non-`nil`, it is a function that will be called by `yank-pop` to undo the insertion of the current object. It is called with two arguments, the start and end of the current region. *function* can set `yank-undo-function` to override the *undo* value.

`yank-excluded-properties` [User Option]
Yanking discards certain text properties from the yanked text, as described above. The value of this variable is the list of properties to discard. Its default value contains properties that might lead to annoying results, such as causing the text to respond to the mouse or specifying key bindings.

32.8.4 Functions for Yanking

This section describes higher-level commands for yanking, which are intended primarily for the user but useful also in Lisp programs. Both `yank` and `yank-pop` honor the `yank-excluded-properties` variable and `yank-handler` text property (see Section 32.8.3 [Yanking], page 133).

`yank` **&optional** *arg* [Command]
This command inserts before point the text at the front of the kill ring. It sets the mark at the beginning of that text, using `push-mark` (see Section 31.7 [The Mark], page 117), and puts point at the end.

If *arg* is a non-`nil` list (which occurs interactively when the user types `C-u` with no digits), then `yank` inserts the text as described above, but puts point before the yanked text and sets the mark after it.

If *arg* is a number, then `yank` inserts the *arg*th most recently killed text—the *arg*th element of the kill ring list, counted cyclically from the front, which is considered the first element for this purpose.

`yank` does not alter the contents of the kill ring, unless it used text provided by another program, in which case it pushes that text onto the kill ring. However if *arg* is an integer different from one, it rotates the kill ring to place the yanked string at the front.

`yank` returns `nil`.

`yank-pop` **&optional** *arg* [Command]
This command replaces the just-yanked entry from the kill ring with a different entry from the kill ring.

This is allowed only immediately after a `yank` or another `yank-pop`. At such a time, the region contains text that was just inserted by yanking. `yank-pop` deletes that text and inserts in its place a different piece of killed text. It does not add the deleted text to the kill ring, since it is already in the kill ring somewhere. It does however rotate the kill ring to place the newly yanked string at the front.

If *arg* is `nil`, then the replacement text is the previous element of the kill ring. If *arg* is numeric, the replacement is the *arg*th previous kill. If *arg* is negative, a more recent kill is the replacement.

The sequence of kills in the kill ring wraps around, so that after the oldest one comes the newest one, and before the newest one goes the oldest.

The return value is always `nil`.

`yank-undo-function` [Variable]

If this variable is non-`nil`, the function `yank-pop` uses its value instead of `delete-region` to delete the text inserted by the previous `yank` or `yank-pop` command. The value must be a function of two arguments, the start and end of the current region.

The function `insert-for-yank` automatically sets this variable according to the *undo* element of the `yank-handler` text property, if there is one.

32.8.5 Low-Level Kill Ring

These functions and variables provide access to the kill ring at a lower level, but are still convenient for use in Lisp programs, because they take care of interaction with window system selections (see Section 29.18 [Window System Selections], page 91).

`current-kill` *n* **&optional** *do-not-move* [Function]

The function `current-kill` rotates the yanking pointer, which designates the "front" of the kill ring, by *n* places (from newer kills to older ones), and returns the text at that place in the ring.

If the optional second argument *do-not-move* is non-`nil`, then `current-kill` doesn't alter the yanking pointer; it just returns the *n*th kill, counting from the current yanking pointer.

If *n* is zero, indicating a request for the latest kill, `current-kill` calls the value of `interprogram-paste-function` (documented below) before consulting the kill ring. If that value is a function and calling it returns a string or a list of several string, `current-kill` pushes the strings onto the kill ring and returns the first string. It also sets the yanking pointer to point to the kill-ring entry of the first string returned by `interprogram-paste-function`, regardless of the value of *do-not-move*. Otherwise, `current-kill` does not treat a zero value for *n* specially: it returns the entry pointed at by the yanking pointer and does not move the yanking pointer.

`kill-new` *string* **&optional** *replace* [Function]

This function pushes the text *string* onto the kill ring and makes the yanking pointer point to it. It discards the oldest entry if appropriate. It also invokes the value of `interprogram-cut-function` (see below).

If *replace* is non-`nil`, then `kill-new` replaces the first element of the kill ring with *string*, rather than pushing *string* onto the kill ring.

`kill-append` *string before-p* [Function]
This function appends the text *string* to the first entry in the kill ring and makes the yanking pointer point to the combined entry. Normally *string* goes at the end of the entry, but if *before-p* is non-`nil`, it goes at the beginning. This function also invokes the value of `interprogram-cut-function` (see below).

`interprogram-paste-function` [Variable]
This variable provides a way of transferring killed text from other programs, when you are using a window system. Its value should be `nil` or a function of no arguments.

If the value is a function, `current-kill` calls it to get the "most recent kill". If the function returns a non-`nil` value, then that value is used as the "most recent kill". If it returns `nil`, then the front of the kill ring is used.

To facilitate support for window systems that support multiple selections, this function may also return a list of strings. In that case, the first string is used as the "most recent kill", and all the other strings are pushed onto the kill ring, for easy access by `yank-pop`.

The normal use of this function is to get the window system's clipboard as the most recent kill, even if the selection belongs to another application. See Section 29.18 [Window System Selections], page 91. However, if the clipboard contents come from the current Emacs session, this function should return `nil`.

`interprogram-cut-function` [Variable]
This variable provides a way of communicating killed text to other programs, when you are using a window system. Its value should be `nil` or a function of one required argument.

If the value is a function, `kill-new` and `kill-append` call it with the new first element of the kill ring as the argument.

The normal use of this function is to put newly killed text in the window system's clipboard. See Section 29.18 [Window System Selections], page 91.

32.8.6 Internals of the Kill Ring

The variable `kill-ring` holds the kill ring contents, in the form of a list of strings. The most recent kill is always at the front of the list.

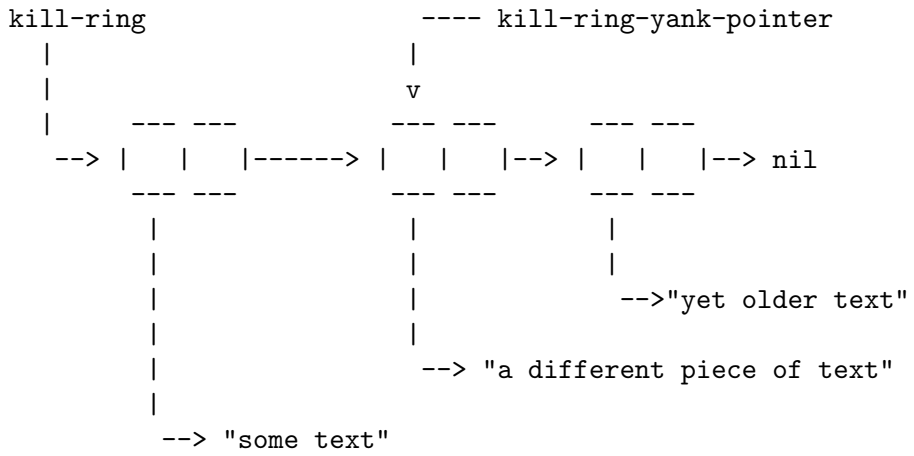
The `kill-ring-yank-pointer` variable points to a link in the kill ring list, whose CAR is the text to yank next. We say it identifies the "front" of the ring. Moving `kill-ring-yank-pointer` to a different link is called *rotating the kill ring*. We call the kill ring a "ring" because the functions that move the yank pointer wrap around from the end of the list to the beginning, or vice-versa. Rotation of the kill ring is virtual; it does not change the value of `kill-ring`.

Both `kill-ring` and `kill-ring-yank-pointer` are Lisp variables whose values are normally lists. The word "pointer" in the name of the `kill-ring-yank-pointer` indicates that the variable's purpose is to identify one element of the list for use by the next yank command.

The value of `kill-ring-yank-pointer` is always `eq` to one of the links in the kill ring list. The element it identifies is the CAR of that link. Kill commands, which change the kill

ring, also set this variable to the value of `kill-ring`. The effect is to rotate the ring so that the newly killed text is at the front.

Here is a diagram that shows the variable `kill-ring-yank-pointer` pointing to the second entry in the kill ring `("some text" "a different piece of text" "yet older text")`.



This state of affairs might occur after *C-y* (`yank`) immediately followed by *M-y* (`yank-pop`).

`kill-ring` [Variable]
: This variable holds the list of killed text sequences, most recently killed first.

`kill-ring-yank-pointer` [Variable]
: This variable's value indicates which element of the kill ring is at the "front" of the ring for yanking. More precisely, the value is a tail of the value of `kill-ring`, and its CAR is the kill string that *C-y* should yank.

`kill-ring-max` [User Option]
: The value of this variable is the maximum length to which the kill ring can grow, before elements are thrown away at the end. The default value for `kill-ring-max` is 60.

32.9 Undo

Most buffers have an *undo list*, which records all changes made to the buffer's text so that they can be undone. (The buffers that don't have one are usually special-purpose buffers for which Emacs assumes that undoing is not useful. In particular, any buffer whose name begins with a space has its undo recording off by default; see Section 27.3 [Buffer Names], page 4.) All the primitives that modify the text in the buffer automatically add elements to the front of the undo list, which is in the variable `buffer-undo-list`.

`buffer-undo-list` [Variable]
: This buffer-local variable's value is the undo list of the current buffer. A value of `t` disables the recording of undo information.

Here are the kinds of elements an undo list can have:

position
: This kind of element records a previous value of point; undoing this element moves point to *position*. Ordinary cursor motion does not make any sort of

undo record, but deletion operations use these entries to record where point was before the command.

`(`*beg* `.` *end*`)`
: This kind of element indicates how to delete text that was inserted. Upon insertion, the text occupied the range *beg*–*end* in the buffer.

`(`*text* `.` *position*`)`
: This kind of element indicates how to reinsert text that was deleted. The deleted text itself is the string *text*. The place to reinsert it is `(abs` *position*`)`. If *position* is positive, point was at the beginning of the deleted text, otherwise it was at the end.

`(t` *high* `.` *low*`)`
: This kind of element indicates that an unmodified buffer became modified. The elements *high* and *low* are two integers, each recording 16 bits of the visited file's modification time as of when it was previously visited or saved. `primitive-undo` uses those values to determine whether to mark the buffer as unmodified once again; it does so only if the file's modification time matches those numbers.

`(nil` *property value beg* `.` *end*`)`
: This kind of element records a change in a text property. Here's how you might undo the change:

```
(put-text-property beg end property value)
```

`(`*marker* `.` *adjustment*`)`
: This kind of element records the fact that the marker *marker* was relocated due to deletion of surrounding text, and that it moved *adjustment* character positions. Undoing this element moves *marker* − *adjustment* characters.

`(apply` *funname* `.` *args*`)`
: This is an extensible undo item, which is undone by calling *funname* with arguments *args*.

`(apply` *delta beg end funname* `.` *args*`)`
: This is an extensible undo item, which records a change limited to the range *beg* to *end*, which increased the size of the buffer by *delta*. It is undone by calling *funname* with arguments *args*.

  This kind of element enables undo limited to a region to determine whether the element pertains to that region.

`nil`
: This element is a boundary. The elements between two boundaries are called a *change group*; normally, each change group corresponds to one keyboard command, and undo commands normally undo an entire group as a unit.

`undo-boundary` [Function]

This function places a boundary element in the undo list. The undo command stops at such a boundary, and successive undo commands undo to earlier and earlier boundaries. This function returns `nil`.

The editor command loop automatically calls `undo-boundary` just before executing each key sequence, so that each undo normally undoes the effects of one command.

As an exception, the command `self-insert-command`, which produces self-inserting input characters (see Section 32.5 [Commands for Insertion], page 127), may remove the boundary inserted by the command loop: a boundary is accepted for the first such character, the next 19 consecutive self-inserting input characters do not have boundaries, and then the 20th does; and so on as long as the self-inserting characters continue. Hence, sequences of consecutive character insertions can be undone as a group.

All buffer modifications add a boundary whenever the previous undoable change was made in some other buffer. This is to ensure that each command makes a boundary in each buffer where it makes changes.

Calling this function explicitly is useful for splitting the effects of a command into more than one unit. For example, `query-replace` calls `undo-boundary` after each replacement, so that the user can undo individual replacements one by one.

`undo-in-progress` [Variable]

This variable is normally `nil`, but the undo commands bind it to `t`. This is so that various kinds of change hooks can tell when they're being called for the sake of undoing.

`primitive-undo` *count list* [Function]

This is the basic function for undoing elements of an undo list. It undoes the first *count* elements of *list*, returning the rest of *list*.

`primitive-undo` adds elements to the buffer's undo list when it changes the buffer. Undo commands avoid confusion by saving the undo list value at the beginning of a sequence of undo operations. Then the undo operations use and update the saved value. The new elements added by undoing are not part of this saved value, so they don't interfere with continuing to undo.

This function does not bind `undo-in-progress`.

32.10 Maintaining Undo Lists

This section describes how to enable and disable undo information for a given buffer. It also explains how the undo list is truncated automatically so it doesn't get too big.

Recording of undo information in a newly created buffer is normally enabled to start with; but if the buffer name starts with a space, the undo recording is initially disabled. You can explicitly enable or disable undo recording with the following two functions, or by setting `buffer-undo-list` yourself.

`buffer-enable-undo` **&optional** *buffer-or-name* [Command]

This command enables recording undo information for buffer *buffer-or-name*, so that subsequent changes can be undone. If no argument is supplied, then the current buffer is used. This function does nothing if undo recording is already enabled in the buffer. It returns `nil`.

In an interactive call, *buffer-or-name* is the current buffer. You cannot specify any other buffer.

`buffer-disable-undo` **&optional** *buffer-or-name* [Command]
: This function discards the undo list of *buffer-or-name*, and disables further recording of undo information. As a result, it is no longer possible to undo either previous changes or any subsequent changes. If the undo list of *buffer-or-name* is already disabled, this function has no effect.

  This function returns `nil`.

As editing continues, undo lists get longer and longer. To prevent them from using up all available memory space, garbage collection trims them back to size limits you can set. (For this purpose, the "size" of an undo list measures the cons cells that make up the list, plus the strings of deleted text.) Three variables control the range of acceptable sizes: `undo-limit`, `undo-strong-limit` and `undo-outer-limit`. In these variables, size is counted as the number of bytes occupied, which includes both saved text and other data.

`undo-limit` [User Option]
: This is the soft limit for the acceptable size of an undo list. The change group at which this size is exceeded is the last one kept.

`undo-strong-limit` [User Option]
: This is the upper limit for the acceptable size of an undo list. The change group at which this size is exceeded is discarded itself (along with all older change groups). There is one exception: the very latest change group is only discarded if it exceeds `undo-outer-limit`.

`undo-outer-limit` [User Option]
: If at garbage collection time the undo info for the current command exceeds this limit, Emacs discards the info and displays a warning. This is a last ditch limit to prevent memory overflow.

`undo-ask-before-discard` [User Option]
: If this variable is non-`nil`, when the undo info exceeds `undo-outer-limit`, Emacs asks in the echo area whether to discard the info. The default value is `nil`, which means to discard it automatically.

  This option is mainly intended for debugging. Garbage collection is inhibited while the question is asked, which means that Emacs might leak memory if the user waits too long before answering the question.

32.11 Filling

Filling means adjusting the lengths of lines (by moving the line breaks) so that they are nearly (but no greater than) a specified maximum width. Additionally, lines can be *justified*, which means inserting spaces to make the left and/or right margins line up precisely. The width is controlled by the variable `fill-column`. For ease of reading, lines should be no longer than 70 or so columns.

You can use Auto Fill mode (see Section 32.14 [Auto Filling], page 146) to fill text automatically as you insert it, but changes to existing text may leave it improperly filled. Then you must fill the text explicitly.

Most of the commands in this section return values that are not meaningful. All the functions that do filling take note of the current left margin, current right margin, and

current justification style (see Section 32.12 [Margins], page 143). If the current justification style is `none`, the filling functions don't actually do anything.

Several of the filling functions have an argument *justify*. If it is non-`nil`, that requests some kind of justification. It can be `left`, `right`, `full`, or `center`, to request a specific style of justification. If it is `t`, that means to use the current justification style for this part of the text (see `current-justification`, below). Any other value is treated as `full`.

When you call the filling functions interactively, using a prefix argument implies the value `full` for *justify*.

`fill-paragraph` **&optional** *justify region* [Command]
: This command fills the paragraph at or after point. If *justify* is non-`nil`, each line is justified as well. It uses the ordinary paragraph motion commands to find paragraph boundaries. See Section "Paragraphs" in *The GNU Emacs Manual*.

  When *region* is non-`nil`, then if Transient Mark mode is enabled and the mark is active, this command calls `fill-region` to fill all the paragraphs in the region, instead of filling only the current paragraph. When this command is called interactively, *region* is `t`.

`fill-region` *start end* **&optional** *justify nosqueeze to-eop* [Command]
: This command fills each of the paragraphs in the region from *start* to *end*. It justifies as well if *justify* is non-`nil`.

  If *nosqueeze* is non-`nil`, that means to leave whitespace other than line breaks untouched. If *to-eop* is non-`nil`, that means to keep filling to the end of the paragraph—or the next hard newline, if `use-hard-newlines` is enabled (see below).

  The variable `paragraph-separate` controls how to distinguish paragraphs. See Section 34.8 [Standard Regexps], page 233.

`fill-individual-paragraphs` *start end* **&optional** *justify citation-regexp* [Command]
: This command fills each paragraph in the region according to its individual fill prefix. Thus, if the lines of a paragraph were indented with spaces, the filled paragraph will remain indented in the same fashion.

  The first two arguments, *start* and *end*, are the beginning and end of the region to be filled. The third and fourth arguments, *justify* and *citation-regexp*, are optional. If *justify* is non-`nil`, the paragraphs are justified as well as filled. If *citation-regexp* is non-`nil`, it means the function is operating on a mail message and therefore should not fill the header lines. If *citation-regexp* is a string, it is used as a regular expression; if it matches the beginning of a line, that line is treated as a citation marker.

  Ordinarily, `fill-individual-paragraphs` regards each change in indentation as starting a new paragraph. If `fill-individual-varying-indent` is non-`nil`, then only separator lines separate paragraphs. That mode can handle indented paragraphs with additional indentation on the first line.

`fill-individual-varying-indent` [User Option]
: This variable alters the action of `fill-individual-paragraphs` as described above.

`fill-region-as-paragraph` *start end* **&optional** *justify nosqueeze squeeze-after* [Command]

This command considers a region of text as a single paragraph and fills it. If the region was made up of many paragraphs, the blank lines between paragraphs are removed. This function justifies as well as filling when *justify* is non-`nil`.

If *nosqueeze* is non-`nil`, that means to leave whitespace other than line breaks untouched. If *squeeze-after* is non-`nil`, it specifies a position in the region, and means don't canonicalize spaces before that position.

In Adaptive Fill mode, this command calls `fill-context-prefix` to choose a fill prefix by default. See Section 32.13 [Adaptive Fill], page 144.

`justify-current-line` **&optional** *how eop nosqueeze* [Command]

This command inserts spaces between the words of the current line so that the line ends exactly at `fill-column`. It returns `nil`.

The argument *how*, if non-`nil` specifies explicitly the style of justification. It can be `left`, `right`, `full`, `center`, or `none`. If it is `t`, that means to do follow specified justification style (see `current-justification`, below). `nil` means to do full justification.

If *eop* is non-`nil`, that means do only left-justification if `current-justification` specifies full justification. This is used for the last line of a paragraph; even if the paragraph as a whole is fully justified, the last line should not be.

If *nosqueeze* is non-`nil`, that means do not change interior whitespace.

`default-justification` [User Option]

This variable's value specifies the style of justification to use for text that doesn't specify a style with a text property. The possible values are `left`, `right`, `full`, `center`, or `none`. The default value is `left`.

`current-justification` [Function]

This function returns the proper justification style to use for filling the text around point.

This returns the value of the `justification` text property at point, or the variable *default-justification* if there is no such text property. However, it returns `nil` rather than `none` to mean "don't justify".

`sentence-end-double-space` [User Option]

If this variable is non-`nil`, a period followed by just one space does not count as the end of a sentence, and the filling functions avoid breaking the line at such a place.

`sentence-end-without-period` [User Option]

If this variable is non-`nil`, a sentence can end without a period. This is used for languages like Thai, where sentences end with a double space but without a period.

`sentence-end-without-space` [User Option]

If this variable is non-`nil`, it should be a string of characters that can end a sentence without following spaces.

`fill-paragraph-function` [Variable]
This variable provides a way to override the filling of paragraphs. If its value is non-`nil`, `fill-paragraph` calls this function to do the work. If the function returns a non-`nil` value, `fill-paragraph` assumes the job is done, and immediately returns that value.

The usual use of this feature is to fill comments in programming language modes. If the function needs to fill a paragraph in the usual way, it can do so as follows:

```
(let ((fill-paragraph-function nil))
  (fill-paragraph arg))
```

`fill-forward-paragraph-function` [Variable]
This variable provides a way to override how the filling functions, such as `fill-region` and `fill-paragraph`, move forward to the next paragraph. Its value should be a function, which is called with a single argument *n*, the number of paragraphs to move, and should return the difference between *n* and the number of paragraphs actually moved. The default value of this variable is `forward-paragraph`. See Section "Paragraphs" in *The GNU Emacs Manual*.

`use-hard-newlines` [Variable]
If this variable is non-`nil`, the filling functions do not delete newlines that have the `hard` text property. These "hard newlines" act as paragraph separators.

32.12 Margins for Filling

`fill-prefix` [User Option]
This buffer-local variable, if non-`nil`, specifies a string of text that appears at the beginning of normal text lines and should be disregarded when filling them. Any line that fails to start with the fill prefix is considered the start of a paragraph; so is any line that starts with the fill prefix followed by additional whitespace. Lines that start with the fill prefix but no additional whitespace are ordinary text lines that can be filled together. The resulting filled lines also start with the fill prefix.

The fill prefix follows the left margin whitespace, if any.

`fill-column` [User Option]
This buffer-local variable specifies the maximum width of filled lines. Its value should be an integer, which is a number of columns. All the filling, justification, and centering commands are affected by this variable, including Auto Fill mode (see Section 32.14 [Auto Filling], page 146).

As a practical matter, if you are writing text for other people to read, you should set `fill-column` to no more than 70. Otherwise the line will be too long for people to read comfortably, and this can make the text seem clumsy.

The default value for `fill-column` is 70.

`set-left-margin` *from to margin* [Command]
This sets the `left-margin` property on the text from *from* to *to* to the value *margin*. If Auto Fill mode is enabled, this command also refills the region to fit the new margin.

`set-right-margin` *from to margin* [Command]
: This sets the `right-margin` property on the text from *from* to *to* to the value *margin*. If Auto Fill mode is enabled, this command also refills the region to fit the new margin.

`current-left-margin` [Function]
: This function returns the proper left margin value to use for filling the text around point. The value is the sum of the `left-margin` property of the character at the start of the current line (or zero if none), and the value of the variable `left-margin`.

`current-fill-column` [Function]
: This function returns the proper fill column value to use for filling the text around point. The value is the value of the `fill-column` variable, minus the value of the `right-margin` property of the character after point.

`move-to-left-margin` **&optional** *n force* [Command]
: This function moves point to the left margin of the current line. The column moved to is determined by calling the function `current-left-margin`. If the argument *n* is non-`nil`, `move-to-left-margin` moves forward $n-1$ lines first.

  If *force* is non-`nil`, that says to fix the line's indentation if that doesn't match the left margin value.

`delete-to-left-margin` **&optional** *from to* [Function]
: This function removes left margin indentation from the text between *from* and *to*. The amount of indentation to delete is determined by calling `current-left-margin`. In no case does this function delete non-whitespace. If *from* and *to* are omitted, they default to the whole buffer.

`indent-to-left-margin` [Function]
: This function adjusts the indentation at the beginning of the current line to the value specified by the variable `left-margin`. (That may involve either inserting or deleting whitespace.) This function is value of `indent-line-function` in Paragraph-Indent Text mode.

`left-margin` [User Option]
: This variable specifies the base left margin column. In Fundamental mode, `C-j` indents to this column. This variable automatically becomes buffer-local when set in any fashion.

`fill-nobreak-predicate` [User Option]
: This variable gives major modes a way to specify not to break a line at certain places. Its value should be a list of functions. Whenever filling considers breaking the line at a certain place in the buffer, it calls each of these functions with no arguments and with point located at that place. If any of the functions returns non-`nil`, then the line won't be broken there.

32.13 Adaptive Fill Mode

When *Adaptive Fill Mode* is enabled, Emacs determines the fill prefix automatically from the text in each paragraph being filled rather than using a predetermined value. During

filling, this fill prefix gets inserted at the start of the second and subsequent lines of the paragraph as described in Section 32.11 [Filling], page 140, and in Section 32.14 [Auto Filling], page 146.

`adaptive-fill-mode` [User Option]
Adaptive Fill mode is enabled when this variable is non-`nil`. It is `t` by default.

`fill-context-prefix` *from to* [Function]
This function implements the heart of Adaptive Fill mode; it chooses a fill prefix based on the text between *from* and *to*, typically the start and end of a paragraph. It does this by looking at the first two lines of the paragraph, based on the variables described below.

Usually, this function returns the fill prefix, a string. However, before doing this, the function makes a final check (not specially mentioned in the following) that a line starting with this prefix wouldn't look like the start of a paragraph. Should this happen, the function signals the anomaly by returning `nil` instead.

In detail, `fill-context-prefix` does this:

1. It takes a candidate for the fill prefix from the first line—it tries first the function in `adaptive-fill-function` (if any), then the regular expression `adaptive-fill-regexp` (see below). The first non-`nil` result of these, or the empty string if they're both `nil`, becomes the first line's candidate.
2. If the paragraph has as yet only one line, the function tests the validity of the prefix candidate just found. The function then returns the candidate if it's valid, or a string of spaces otherwise. (see the description of `adaptive-fill-first-line-regexp` below).
3. When the paragraph already has two lines, the function next looks for a prefix candidate on the second line, in just the same way it did for the first line. If it doesn't find one, it returns `nil`.
4. The function now compares the two candidate prefixes heuristically: if the non-whitespace characters in the line 2 candidate occur in the same order in the line 1 candidate, the function returns the line 2 candidate. Otherwise, it returns the largest initial substring which is common to both candidates (which might be the empty string).

`adaptive-fill-regexp` [User Option]
Adaptive Fill mode matches this regular expression against the text starting after the left margin whitespace (if any) on a line; the characters it matches are that line's candidate for the fill prefix.

The default value matches whitespace with certain punctuation characters intermingled.

`adaptive-fill-first-line-regexp` [User Option]
Used only in one-line paragraphs, this regular expression acts as an additional check of the validity of the one available candidate fill prefix: the candidate must match this regular expression, or match `comment-start-skip`. If it doesn't, `fill-context-prefix` replaces the candidate with a string of spaces "of the same width" as it.

The default value of this variable is `"\\‘[ \t]*\\’"`, which matches only a string of whitespace. The effect of this default is to force the fill prefixes found in one-line paragraphs always to be pure whitespace.

`adaptive-fill-function` [User Option]
You can specify more complex ways of choosing a fill prefix automatically by setting this variable to a function. The function is called with point after the left margin (if any) of a line, and it must preserve point. It should return either “that line’s” fill prefix or `nil`, meaning it has failed to determine a prefix.

32.14 Auto Filling

Auto Fill mode is a minor mode that fills lines automatically as text is inserted. This section describes the hook used by Auto Fill mode. For a description of functions that you can call explicitly to fill and justify existing text, see Section 32.11 [Filling], page 140.

Auto Fill mode also enables the functions that change the margins and justification style to refill portions of the text. See Section 32.12 [Margins], page 143.

`auto-fill-function` [Variable]
The value of this buffer-local variable should be a function (of no arguments) to be called after self-inserting a character from the table `auto-fill-chars`. It may be `nil`, in which case nothing special is done in that case.

The value of `auto-fill-function` is `do-auto-fill` when Auto-Fill mode is enabled. That is a function whose sole purpose is to implement the usual strategy for breaking a line.

`normal-auto-fill-function` [Variable]
This variable specifies the function to use for `auto-fill-function`, if and when Auto Fill is turned on. Major modes can set buffer-local values for this variable to alter how Auto Fill works.

`auto-fill-chars` [Variable]
A char table of characters which invoke `auto-fill-function` when self-inserted—space and newline in most language environments. They have an entry `t` in the table.

32.15 Sorting Text

The sorting functions described in this section all rearrange text in a buffer. This is in contrast to the function `sort`, which rearranges the order of the elements of a list (see Section 5.6.3 [Rearrangement], page 76, vol. 1). The values returned by these functions are not meaningful.

`sort-subr` *reverse nextrecfun endrecfun* **&optional** *startkeyfun endkeyfun predicate* [Function]
This function is the general text-sorting routine that subdivides a buffer into records and then sorts them. Most of the commands in this section use this function.

To understand how `sort-subr` works, consider the whole accessible portion of the buffer as being divided into disjoint pieces called *sort records*. The records may or

may not be contiguous, but they must not overlap. A portion of each sort record (perhaps all of it) is designated as the sort key. Sorting rearranges the records in order by their sort keys.

Usually, the records are rearranged in order of ascending sort key. If the first argument to the `sort-subr` function, *reverse*, is non-`nil`, the sort records are rearranged in order of descending sort key.

The next four arguments to `sort-subr` are functions that are called to move point across a sort record. They are called many times from within `sort-subr`.

1. *nextrecfun* is called with point at the end of a record. This function moves point to the start of the next record. The first record is assumed to start at the position of point when `sort-subr` is called. Therefore, you should usually move point to the beginning of the buffer before calling `sort-subr`.

   This function can indicate there are no more sort records by leaving point at the end of the buffer.

2. *endrecfun* is called with point within a record. It moves point to the end of the record.

3. *startkeyfun* is called to move point from the start of a record to the start of the sort key. This argument is optional; if it is omitted, the whole record is the sort key. If supplied, the function should either return a non-`nil` value to be used as the sort key, or return `nil` to indicate that the sort key is in the buffer starting at point. In the latter case, *endkeyfun* is called to find the end of the sort key.

4. *endkeyfun* is called to move point from the start of the sort key to the end of the sort key. This argument is optional. If *startkeyfun* returns `nil` and this argument is omitted (or `nil`), then the sort key extends to the end of the record. There is no need for *endkeyfun* if *startkeyfun* returns a non-`nil` value.

The argument *predicate* is the function to use to compare keys. If keys are numbers, it defaults to `<`; otherwise it defaults to `string<`.

As an example of `sort-subr`, here is the complete function definition for `sort-lines`:

```
;; Note that the first two lines of doc string
;; are effectively one line when viewed by a user.
(defun sort-lines (reverse beg end)
  "Sort lines in region alphabetically;\
 argument means descending order.
Called from a program, there are three arguments:
REVERSE (non-nil means reverse order),\
 BEG and END (region to sort).
The variable ‘sort-fold-case’ determines\
 whether alphabetic case affects
the sort order."
```

```
(interactive "P\nr")
(save-excursion
  (save-restriction
    (narrow-to-region beg end)
    (goto-char (point-min))
    (let ((inhibit-field-text-motion t))
      (sort-subr reverse 'forward-line 'end-of-line)))))
```

Here `forward-line` moves point to the start of the next record, and `end-of-line` moves point to the end of record. We do not pass the arguments *startkeyfun* and *endkeyfun*, because the entire record is used as the sort key.

The `sort-paragraphs` function is very much the same, except that its `sort-subr` call looks like this:

```
(sort-subr reverse
           (function
             (lambda ()
               (while (and (not (eobp))
                           (looking-at paragraph-separate))
                 (forward-line 1))))
           'forward-paragraph)
```

Markers pointing into any sort records are left with no useful position after `sort-subr` returns.

`sort-fold-case` [User Option]

If this variable is non-`nil`, `sort-subr` and the other buffer sorting functions ignore case when comparing strings.

`sort-regexp-fields` *reverse record-regexp key-regexp start end* [Command]

This command sorts the region between *start* and *end* alphabetically as specified by *record-regexp* and *key-regexp*. If *reverse* is a negative integer, then sorting is in reverse order.

Alphabetical sorting means that two sort keys are compared by comparing the first characters of each, the second characters of each, and so on. If a mismatch is found, it means that the sort keys are unequal; the sort key whose character is less at the point of first mismatch is the lesser sort key. The individual characters are compared according to their numerical character codes in the Emacs character set.

The value of the *record-regexp* argument specifies how to divide the buffer into sort records. At the end of each record, a search is done for this regular expression, and the text that matches it is taken as the next record. For example, the regular expression '`^.+$`', which matches lines with at least one character besides a newline, would make each such line into a sort record. See Section 34.3 [Regular Expressions], page 211, for a description of the syntax and meaning of regular expressions.

The value of the *key-regexp* argument specifies what part of each record is the sort key. The *key-regexp* could match the whole record, or only a part. In the latter case, the rest of the record has no effect on the sorted order of records, but it is carried along when the record moves to its new position.

The *key-regexp* argument can refer to the text matched by a subexpression of *record-regexp*, or it can be a regular expression on its own.

If *key-regexp* is:

'`\digit`' then the text matched by the *digit*th '`\(...\)`' parenthesis grouping in *record-regexp* is the sort key.

'`\&`' then the whole record is the sort key.

a regular expression
then `sort-regexp-fields` searches for a match for the regular expression within the record. If such a match is found, it is the sort key. If there is no match for *key-regexp* within a record then that record is ignored, which means its position in the buffer is not changed. (The other records may move around it.)

For example, if you plan to sort all the lines in the region by the first word on each line starting with the letter '`f`', you should set *record-regexp* to '`^.*$`' and set *key-regexp* to '`\<f\w*\>`'. The resulting expression looks like this:

```
(sort-regexp-fields nil "^.*$" "\\<f\\w*\\>"
                    (region-beginning)
                    (region-end))
```

If you call `sort-regexp-fields` interactively, it prompts for *record-regexp* and *key-regexp* in the minibuffer.

`sort-lines` *reverse start end* [Command]
This command alphabetically sorts lines in the region between *start* and *end*. If *reverse* is non-`nil`, the sort is in reverse order.

`sort-paragraphs` *reverse start end* [Command]
This command alphabetically sorts paragraphs in the region between *start* and *end*. If *reverse* is non-`nil`, the sort is in reverse order.

`sort-pages` *reverse start end* [Command]
This command alphabetically sorts pages in the region between *start* and *end*. If *reverse* is non-`nil`, the sort is in reverse order.

`sort-fields` *field start end* [Command]
This command sorts lines in the region between *start* and *end*, comparing them alphabetically by the *field*th field of each line. Fields are separated by whitespace and numbered starting from 1. If *field* is negative, sorting is by the $-$*field*th field from the end of the line. This command is useful for sorting tables.

`sort-numeric-fields` *field start end* [Command]
This command sorts lines in the region between *start* and *end*, comparing them numerically by the *field*th field of each line. Fields are separated by whitespace and numbered starting from 1. The specified field must contain a number in each line of the region. Numbers starting with 0 are treated as octal, and numbers starting with '`0x`' are treated as hexadecimal.

If *field* is negative, sorting is by the $-$*field*th field from the end of the line. This command is useful for sorting tables.

`sort-numeric-base` [User Option]
This variable specifies the default radix for `sort-numeric-fields` to parse numbers.

`sort-columns` *reverse* **&optional** *beg end* [Command]
This command sorts the lines in the region between *beg* and *end*, comparing them alphabetically by a certain range of columns. The column positions of *beg* and *end* bound the range of columns to sort on.

If *reverse* is non-`nil`, the sort is in reverse order.

One unusual thing about this command is that the entire line containing position *beg*, and the entire line containing position *end*, are included in the region sorted.

Note that `sort-columns` rejects text that contains tabs, because tabs could be split across the specified columns. Use *`M-x untabify`* to convert tabs to spaces before sorting.

When possible, this command actually works by calling the `sort` utility program.

32.16 Counting Columns

The column functions convert between a character position (counting characters from the beginning of the buffer) and a column position (counting screen characters from the beginning of a line).

These functions count each character according to the number of columns it occupies on the screen. This means control characters count as occupying 2 or 4 columns, depending upon the value of `ctl-arrow`, and tabs count as occupying a number of columns that depends on the value of `tab-width` and on the column where the tab begins. See Section 38.20.1 [Usual Display], page 376.

Column number computations ignore the width of the window and the amount of horizontal scrolling. Consequently, a column value can be arbitrarily high. The first (or leftmost) column is numbered 0. They also ignore overlays and text properties, aside from invisibility.

`current-column` [Function]
This function returns the horizontal position of point, measured in columns, counting from 0 at the left margin. The column position is the sum of the widths of all the displayed representations of the characters between the start of the current line and point.

For an example of using `current-column`, see the description of `count-lines` in Section 30.2.4 [Text Lines], page 102.

`move-to-column` *column* **&optional** *force* [Command]
This function moves point to *column* in the current line. The calculation of *column* takes into account the widths of the displayed representations of the characters between the start of the line and point.

When called interactively, *column* is the value of prefix numeric argument. If *column* is not an integer, an error is signaled.

If column *column* is beyond the end of the line, point moves to the end of the line. If *column* is negative, point moves to the beginning of the line.

If it is impossible to move to column *column* because that is in the middle of a multi-column character such as a tab, point moves to the end of that character. However, if *force* is non-`nil`, and *column* is in the middle of a tab, then `move-to-column` converts the tab into spaces so that it can move precisely to column *column*. Other multi-column characters can cause anomalies despite *force*, since there is no way to split them.

The argument *force* also has an effect if the line isn't long enough to reach column *column*; if it is `t`, that means to add whitespace at the end of the line to reach that column.

The return value is the column number actually moved to.

32.17 Indentation

The indentation functions are used to examine, move to, and change whitespace that is at the beginning of a line. Some of the functions can also change whitespace elsewhere on a line. Columns and indentation count from zero at the left margin.

32.17.1 Indentation Primitives

This section describes the primitive functions used to count and insert indentation. The functions in the following sections use these primitives. See Section 38.10 [Width], page 323, for related functions.

`current-indentation` [Function]
: This function returns the indentation of the current line, which is the horizontal position of the first nonblank character. If the contents are entirely blank, then this is the horizontal position of the end of the line.

`indent-to` *column* **&optional** *minimum* [Command]
: This function indents from point with tabs and spaces until *column* is reached. If *minimum* is specified and non-`nil`, then at least that many spaces are inserted even if this requires going beyond *column*. Otherwise the function does nothing if point is already beyond *column*. The value is the column at which the inserted indentation ends.

  The inserted whitespace characters inherit text properties from the surrounding text (usually, from the preceding text only). See Section 32.19.6 [Sticky Properties], page 167.

`indent-tabs-mode` [User Option]
: If this variable is non-`nil`, indentation functions can insert tabs as well as spaces. Otherwise, they insert only spaces. Setting this variable automatically makes it buffer-local in the current buffer.

32.17.2 Indentation Controlled by Major Mode

An important function of each major mode is to customize the `TAB` key to indent properly for the language being edited. This section describes the mechanism of the `TAB` key and how to control it. The functions in this section return unpredictable values.

`indent-for-tab-command` **&optional** *rigid* [Command]
This is the command bound to `TAB` in most editing modes. Its usual action is to indent the current line, but it can alternatively insert a tab character or indent a region.

Here is what it does:

- First, it checks whether Transient Mark mode is enabled and the region is active. If so, it called `indent-region` to indent all the text in the region (see Section 32.17.3 [Region Indent], page 153).
- Otherwise, if the indentation function in `indent-line-function` is `indent-to-left-margin` (a trivial command that inserts a tab character), or if the variable `tab-always-indent` specifies that a tab character ought to be inserted (see below), then it inserts a tab character.
- Otherwise, it indents the current line; this is done by calling the function in `indent-line-function`. If the line is already indented, and the value of `tab-always-indent` is `complete` (see below), it tries completing the text at point.

If *rigid* is non-`nil` (interactively, with a prefix argument), then after this command indents a line or inserts a tab, it also rigidly indents the entire balanced expression which starts at the beginning of the current line, in order to reflect the new indentation. This argument is ignored if the command indents the region.

`indent-line-function` [Variable]
This variable's value is the function to be used by `indent-for-tab-command`, and various other indentation commands, to indent the current line. It is usually assigned by the major mode; for instance, Lisp mode sets it to `lisp-indent-line`, C mode sets it to `c-indent-line`, and so on. The default value is `indent-relative`. See Section 23.7 [Auto-Indentation], page 440, vol. 1.

`indent-according-to-mode` [Command]
This command calls the function in `indent-line-function` to indent the current line in a way appropriate for the current major mode.

`newline-and-indent` [Command]
This function inserts a newline, then indents the new line (the one following the newline just inserted) according to the major mode. It does indentation by calling `indent-according-to-mode`.

`reindent-then-newline-and-indent` [Command]
This command reindents the current line, inserts a newline at point, and then indents the new line (the one following the newline just inserted). It does indentation on both lines by calling `indent-according-to-mode`.

`tab-always-indent` [User Option]
This variable can be used to customize the behavior of the `TAB` (`indent-for-tab-command`) command. If the value is `t` (the default), the command normally just indents the current line. If the value is `nil`, the command indents the current line only if point is at the left margin or in the line's indentation; otherwise, it inserts a tab character. If the value is `complete`, the command first tries to indent the current

line, and if the line was already indented, it calls `completion-at-point` to complete the text at point (see Section 20.6.8 [Completion in Buffers], page 306, vol. 1).

32.17.3 Indenting an Entire Region

This section describes commands that indent all the lines in the region. They return unpredictable values.

`indent-region` *start end* **&optional** *to-column* [Command]

This command indents each nonblank line starting between *start* (inclusive) and *end* (exclusive). If *to-column* is `nil`, `indent-region` indents each nonblank line by calling the current mode's indentation function, the value of `indent-line-function`.

If *to-column* is non-`nil`, it should be an integer specifying the number of columns of indentation; then this function gives each line exactly that much indentation, by either adding or deleting whitespace.

If there is a fill prefix, `indent-region` indents each line by making it start with the fill prefix.

`indent-region-function` [Variable]

The value of this variable is a function that can be used by `indent-region` as a short cut. It should take two arguments, the start and end of the region. You should design the function so that it will produce the same results as indenting the lines of the region one by one, but presumably faster.

If the value is `nil`, there is no short cut, and `indent-region` actually works line by line.

A short-cut function is useful in modes such as C mode and Lisp mode, where the `indent-line-function` must scan from the beginning of the function definition: applying it to each line would be quadratic in time. The short cut can update the scan information as it moves through the lines indenting them; this takes linear time. In a mode where indenting a line individually is fast, there is no need for a short cut.

`indent-region` with a non-`nil` argument *to-column* has a different meaning and does not use this variable.

`indent-rigidly` *start end count* [Command]

This command indents all lines starting between *start* (inclusive) and *end* (exclusive) sideways by *count* columns. This "preserves the shape" of the affected region, moving it as a rigid unit. Consequently, this command is useful not only for indenting regions of unindented text, but also for indenting regions of formatted code.

For example, if *count* is 3, this command adds 3 columns of indentation to each of the lines beginning in the region specified.

In Mail mode, `C-c C-y` (`mail-yank-original`) uses `indent-rigidly` to indent the text copied from the message being replied to.

`indent-code-rigidly` *start end columns* **&optional** *nochange-regexp* [Command]

This is like `indent-rigidly`, except that it doesn't alter lines that start within strings or comments.

In addition, it doesn't alter a line if *nochange-regexp* matches at the beginning of the line (if *nochange-regexp* is non-`nil`).

32.17.4 Indentation Relative to Previous Lines

This section describes two commands that indent the current line based on the contents of previous lines.

`indent-relative` **&optional** *unindented-ok* [Command]

This command inserts whitespace at point, extending to the same column as the next *indent point* of the previous nonblank line. An indent point is a non-whitespace character following whitespace. The next indent point is the first one at a column greater than the current column of point. For example, if point is underneath and to the left of the first non-blank character of a line of text, it moves to that column by inserting whitespace.

If the previous nonblank line has no next indent point (i.e., none at a great enough column position), `indent-relative` either does nothing (if *unindented-ok* is non-`nil`) or calls `tab-to-tab-stop`. Thus, if point is underneath and to the right of the last column of a short line of text, this command ordinarily moves point to the next tab stop by inserting whitespace.

The return value of `indent-relative` is unpredictable.

In the following example, point is at the beginning of the second line:

```
                This line is indented twelve spaces.
    ⋆The quick brown fox jumped.
```

Evaluation of the expression `(indent-relative nil)` produces the following:

```
                This line is indented twelve spaces.
                ⋆The quick brown fox jumped.
```

In this next example, point is between the '`m`' and '`p`' of '`jumped`':

```
                This line is indented twelve spaces.
    The quick brown fox jum⋆ped.
```

Evaluation of the expression `(indent-relative nil)` produces the following:

```
                This line is indented twelve spaces.
    The quick brown fox jum  ⋆ped.
```

`indent-relative-maybe` [Command]

This command indents the current line like the previous nonblank line, by calling `indent-relative` with `t` as the *unindented-ok* argument. The return value is unpredictable.

If the previous nonblank line has no indent points beyond the current column, this command does nothing.

32.17.5 Adjustable "Tab Stops"

This section explains the mechanism for user-specified "tab stops" and the mechanisms that use and set them. The name "tab stops" is used because the feature is similar to that of the tab stops on a typewriter. The feature works by inserting an appropriate number of spaces and tab characters to reach the next tab stop column; it does not affect the display of tab characters in the buffer (see Section 38.20.1 [Usual Display], page 376). Note that the `TAB` character as input uses this tab stop feature only in a few major modes, such as Text mode. See Section "Tab Stops" in *The GNU Emacs Manual*.

`tab-to-tab-stop` [Command]
: This command inserts spaces or tabs before point, up to the next tab stop column defined by `tab-stop-list`. It searches the list for an element greater than the current column number, and uses that element as the column to indent to. It does nothing if no such element is found.

`tab-stop-list` [User Option]
: This variable is the list of tab stop columns used by `tab-to-tab-stops`. The elements should be integers in increasing order. The tab stop columns need not be evenly spaced.

  Use *`M-x edit-tab-stops`* to edit the location of tab stops interactively.

32.17.6 Indentation-Based Motion Commands

These commands, primarily for interactive use, act based on the indentation in the text.

`back-to-indentation` [Command]
: This command moves point to the first non-whitespace character in the current line (which is the line in which point is located). It returns `nil`.

`backward-to-indentation` **&optional** *arg* [Command]
: This command moves point backward *arg* lines and then to the first nonblank character on that line. It returns `nil`. If *arg* is omitted or `nil`, it defaults to 1.

`forward-to-indentation` **&optional** *arg* [Command]
: This command moves point forward *arg* lines and then to the first nonblank character on that line. It returns `nil`. If *arg* is omitted or `nil`, it defaults to 1.

32.18 Case Changes

The case change commands described here work on text in the current buffer. See Section 4.8 [Case Conversion], page 59, vol. 1, for case conversion functions that work on strings and characters. See Section 4.9 [Case Tables], page 61, vol. 1, for how to customize which characters are upper or lower case and how to convert them.

`capitalize-region` *start end* [Command]
: This function capitalizes all words in the region defined by *start* and *end*. To capitalize means to convert each word's first character to upper case and convert the rest of each word to lower case. The function returns `nil`.

  If one end of the region is in the middle of a word, the part of the word within the region is treated as an entire word.

  When `capitalize-region` is called interactively, *start* and *end* are point and the mark, with the smallest first.

```
---------- Buffer: foo ----------
This is the contents of the 5th foo.
---------- Buffer: foo ----------
```

```
(capitalize-region 1 44)
⇒ nil

---------- Buffer: foo ----------
This Is The Contents Of The 5th Foo.
---------- Buffer: foo ----------
```

`downcase-region` *start end* [Command]
This function converts all of the letters in the region defined by *start* and *end* to lower case. The function returns `nil`.

When `downcase-region` is called interactively, *start* and *end* are point and the mark, with the smallest first.

`upcase-region` *start end* [Command]
This function converts all of the letters in the region defined by *start* and *end* to upper case. The function returns `nil`.

When `upcase-region` is called interactively, *start* and *end* are point and the mark, with the smallest first.

`capitalize-word` *count* [Command]
This function capitalizes *count* words after point, moving point over as it does. To capitalize means to convert each word's first character to upper case and convert the rest of each word to lower case. If *count* is negative, the function capitalizes the −*count* previous words but does not move point. The value is `nil`.

If point is in the middle of a word, the part of the word before point is ignored when moving forward. The rest is treated as an entire word.

When `capitalize-word` is called interactively, *count* is set to the numeric prefix argument.

`downcase-word` *count* [Command]
This function converts the *count* words after point to all lower case, moving point over as it does. If *count* is negative, it converts the −*count* previous words but does not move point. The value is `nil`.

When `downcase-word` is called interactively, *count* is set to the numeric prefix argument.

`upcase-word` *count* [Command]
This function converts the *count* words after point to all upper case, moving point over as it does. If *count* is negative, it converts the −*count* previous words but does not move point. The value is `nil`.

When `upcase-word` is called interactively, *count* is set to the numeric prefix argument.

32.19 Text Properties

Each character position in a buffer or a string can have a *text property list*, much like the property list of a symbol (see Section 8.4 [Property Lists], page 106, vol. 1). The properties belong to a particular character at a particular place, such as, the letter '`T`' at the beginning

of this sentence or the first 'o' in 'foo'—if the same character occurs in two different places, the two occurrences in general have different properties.

Each property has a name and a value. Both of these can be any Lisp object, but the name is normally a symbol. Typically each property name symbol is used for a particular purpose; for instance, the text property `face` specifies the faces for displaying the character (see Section 32.19.4 [Special Properties], page 162). The usual way to access the property list is to specify a name and ask what value corresponds to it.

If a character has a `category` property, we call it the *property category* of the character. It should be a symbol. The properties of the symbol serve as defaults for the properties of the character.

Copying text between strings and buffers preserves the properties along with the characters; this includes such diverse functions as `substring`, `insert`, and `buffer-substring`.

32.19.1 Examining Text Properties

The simplest way to examine text properties is to ask for the value of a particular property of a particular character. For that, use `get-text-property`. Use `text-properties-at` to get the entire property list of a character. See Section 32.19.3 [Property Search], page 160, for functions to examine the properties of a number of characters at once.

These functions handle both strings and buffers. Keep in mind that positions in a string start from 0, whereas positions in a buffer start from 1.

`get-text-property` *pos prop* **&optional** *object* [Function]
: This function returns the value of the *prop* property of the character after position *pos* in *object* (a buffer or string). The argument *object* is optional and defaults to the current buffer.

  If there is no *prop* property strictly speaking, but the character has a property category that is a symbol, then `get-text-property` returns the *prop* property of that symbol.

`get-char-property` *position prop* **&optional** *object* [Function]
: This function is like `get-text-property`, except that it checks overlays first and then text properties. See Section 38.9 [Overlays], page 315.

  The argument *object* may be a string, a buffer, or a window. If it is a window, then the buffer displayed in that window is used for text properties and overlays, but only the overlays active for that window are considered. If *object* is a buffer, then overlays in that buffer are considered first, in order of decreasing priority, followed by the text properties. If *object* is a string, only text properties are considered, since strings never have overlays.

`get-char-property-and-overlay` *position prop* **&optional** *object* [Function]
: This is like `get-char-property`, but gives extra information about the overlay that the property value comes from.

  Its value is a cons cell whose CAR is the property value, the same value `get-char-property` would return with the same arguments. Its CDR is the overlay in which the property was found, or `nil`, if it was found as a text property or not found at all.

  If *position* is at the end of *object*, both the CAR and the CDR of the value are `nil`.

`char-property-alias-alist` [Variable]
This variable holds an alist which maps property names to a list of alternative property names. If a character does not specify a direct value for a property, the alternative property names are consulted in order; the first non-`nil` value is used. This variable takes precedence over `default-text-properties`, and `category` properties take precedence over this variable.

`text-properties-at` *position* **&optional** *object* [Function]
This function returns the entire property list of the character at *position* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

`default-text-properties` [Variable]
This variable holds a property list giving default values for text properties. Whenever a character does not specify a value for a property, neither directly, through a category symbol, or through `char-property-alias-alist`, the value stored in this list is used instead. Here is an example:

```
(setq default-text-properties '(foo 69)
      char-property-alias-alist nil)
;; Make sure character 1 has no properties of its own.
(set-text-properties 1 2 nil)
;; What we get, when we ask, is the default value.
(get-text-property 1 'foo)
     ⇒ 69
```

32.19.2 Changing Text Properties

The primitives for changing properties apply to a specified range of text in a buffer or string. The function `set-text-properties` (see end of section) sets the entire property list of the text in that range; more often, it is useful to add, change, or delete just certain properties specified by name.

Since text properties are considered part of the contents of the buffer (or string), and can affect how a buffer looks on the screen, any change in buffer text properties marks the buffer as modified. Buffer text property changes are undoable also (see Section 32.9 [Undo], page 137). Positions in a string start from 0, whereas positions in a buffer start from 1.

`put-text-property` *start end prop value* **&optional** *object* [Function]
This function sets the *prop* property to *value* for the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

`add-text-properties` *start end props* **&optional** *object* [Function]
This function adds or overrides text properties for the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

The argument *props* specifies which properties to add. It should have the form of a property list (see Section 8.4 [Property Lists], page 106, vol. 1): a list whose elements include the property names followed alternately by the corresponding values.

The return value is `t` if the function actually changed some property's value; `nil` otherwise (if *props* is `nil` or its values agree with those in the text).

For example, here is how to set the `comment` and `face` properties of a range of text:

```
(add-text-properties start end
                     '(comment t face highlight))
```

`remove-text-properties` *start end props* **&optional** *object* [Function]
This function deletes specified text properties from the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

The argument *props* specifies which properties to delete. It should have the form of a property list (see Section 8.4 [Property Lists], page 106, vol. 1): a list whose elements are property names alternating with corresponding values. But only the names matter—the values that accompany them are ignored. For example, here's how to remove the `face` property.

```
(remove-text-properties start end '(face nil))
```

The return value is `t` if the function actually changed some property's value; `nil` otherwise (if *props* is `nil` or if no character in the specified text had any of those properties).

To remove all text properties from certain text, use `set-text-properties` and specify `nil` for the new property list.

`remove-list-of-text-properties` *start end list-of-properties* [Function]
&optional *object*
Like `remove-text-properties` except that *list-of-properties* is a list of property names only, not an alternating list of property names and values.

`set-text-properties` *start end props* **&optional** *object* [Function]
This function completely replaces the text property list for the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

The argument *props* is the new property list. It should be a list whose elements are property names alternating with corresponding values.

After `set-text-properties` returns, all the characters in the specified range have identical properties.

If *props* is `nil`, the effect is to get rid of all properties from the specified range of text. Here's an example:

```
(set-text-properties start end nil)
```

Do not rely on the return value of this function.

The easiest way to make a string with text properties is with `propertize`:

`propertize` *string* **&rest** *properties* [Function]
This function returns a copy of *string* which has the text properties *properties*. These properties apply to all the characters in the string that is returned. Here is an example that constructs a string with a `face` property and a `mouse-face` property:

```
(propertize "foo" 'face 'italic
            'mouse-face 'bold-italic)
     ⇒ #("foo" 0 3 (mouse-face bold-italic face italic))
```

To put different properties on various parts of a string, you can construct each part with `propertize` and then combine them with `concat`:

```
(concat
 (propertize "foo" 'face 'italic
             'mouse-face 'bold-italic)
 " and "
 (propertize "bar" 'face 'italic
             'mouse-face 'bold-italic))
     ⇒ #("foo and bar"
                 0 3 (face italic mouse-face bold-italic)
                 3 8 nil
                 8 11 (face italic mouse-face bold-italic))
```

See Section 32.2 [Buffer Contents], page 123, for the function `buffer-substring-no-properties`, which copies text from the buffer but does not copy its properties.

32.19.3 Text Property Search Functions

In typical use of text properties, most of the time several or many consecutive characters have the same value for a property. Rather than writing your programs to examine characters one by one, it is much faster to process chunks of text that have the same property value.

Here are functions you can use to do this. They use `eq` for comparing property values. In all cases, *object* defaults to the current buffer.

For good performance, it's very important to use the *limit* argument to these functions, especially the ones that search for a single property—otherwise, they may spend a long time scanning to the end of the buffer, if the property you are interested in does not change.

These functions do not move point; instead, they return a position (or `nil`). Remember that a position is always between two characters; the position returned by these functions is between two characters with different properties.

`next-property-change` *pos* **&optional** *object limit* [Function]

The function scans the text forward from position *pos* in the string or buffer *object* until it finds a change in some text property, then returns the position of the change. In other words, it returns the position of the first character beyond *pos* whose properties are not identical to those of the character just after *pos*.

If *limit* is non-`nil`, then the scan ends at position *limit*. If there is no property change before that point, this function returns *limit*.

The value is `nil` if the properties remain unchanged all the way to the end of *object* and *limit* is `nil`. If the value is non-`nil`, it is a position greater than or equal to *pos*. The value equals *pos* only when *limit* equals *pos*.

Here is an example of how to scan the buffer by chunks of text within which all properties are constant:

```
(while (not (eobp))
  (let ((plist (text-properties-at (point)))
        (next-change
         (or (next-property-change (point) (current-buffer))
             (point-max))))
    Process text from point to next-change...
    (goto-char next-change)))
```

`previous-property-change` *pos* **&optional** *object limit* [Function]
This is like `next-property-change`, but scans back from *pos* instead of forward. If the value is non-`nil`, it is a position less than or equal to *pos*; it equals *pos* only if *limit* equals *pos*.

`next-single-property-change` *pos prop* **&optional** *object limit* [Function]
The function scans text for a change in the *prop* property, then returns the position of the change. The scan goes forward from position *pos* in the string or buffer *object*. In other words, this function returns the position of the first character beyond *pos* whose *prop* property differs from that of the character just after *pos*.

If *limit* is non-`nil`, then the scan ends at position *limit*. If there is no property change before that point, `next-single-property-change` returns *limit*.

The value is `nil` if the property remains unchanged all the way to the end of *object* and *limit* is `nil`. If the value is non-`nil`, it is a position greater than or equal to *pos*; it equals *pos* only if *limit* equals *pos*.

`previous-single-property-change` *pos prop* **&optional** *object limit* [Function]
This is like `next-single-property-change`, but scans back from *pos* instead of forward. If the value is non-`nil`, it is a position less than or equal to *pos*; it equals *pos* only if *limit* equals *pos*.

`next-char-property-change` *pos* **&optional** *limit* [Function]
This is like `next-property-change` except that it considers overlay properties as well as text properties, and if no change is found before the end of the buffer, it returns the maximum buffer position rather than `nil` (in this sense, it resembles the corresponding overlay function `next-overlay-change`, rather than `next-property-change`). There is no *object* operand because this function operates only on the current buffer. It returns the next address at which either kind of property changes.

`previous-char-property-change` *pos* **&optional** *limit* [Function]
This is like `next-char-property-change`, but scans back from *pos* instead of forward, and returns the minimum buffer position if no change is found.

`next-single-char-property-change` *pos prop* **&optional** *object limit* [Function]
This is like `next-single-property-change` except that it considers overlay properties as well as text properties, and if no change is found before the end of the *object*, it returns the maximum valid position in *object* rather than `nil`. Unlike `next-char-property-change`, this function *does* have an *object* operand; if *object* is not a buffer, only text-properties are considered.

`previous-single-char-property-change` *pos prop* **&optional** *object limit* [Function]
This is like `next-single-char-property-change`, but scans back from *pos* instead of forward, and returns the minimum valid position in *object* if no change is found.

`text-property-any` *start end prop value* **&optional** *object* [Function]
This function returns non-`nil` if at least one character between *start* and *end* has a property *prop* whose value is *value*. More precisely, it returns the position of the first such character. Otherwise, it returns `nil`.

The optional fifth argument, *object*, specifies the string or buffer to scan. Positions are relative to *object*. The default for *object* is the current buffer.

`text-property-not-all` *start end prop value* **&optional** *object* [Function]
This function returns non-`nil` if at least one character between *start* and *end* does not have a property *prop* with value *value*. More precisely, it returns the position of the first such character. Otherwise, it returns `nil`.

The optional fifth argument, *object*, specifies the string or buffer to scan. Positions are relative to *object*. The default for *object* is the current buffer.

32.19.4 Properties with Special Meanings

Here is a table of text property names that have special built-in meanings. The following sections list a few additional special property names that control filling and property inheritance. All other names have no standard meaning, and you can use them as you like.

Note: the properties `composition`, `display`, `invisible` and `intangible` can also cause point to move to an acceptable place, after each Emacs command. See Section 21.6 [Adjusting Point], page 326, vol. 1.

`category` If a character has a `category` property, we call it the *property category* of the character. It should be a symbol. The properties of this symbol serve as defaults for the properties of the character.

`face` The `face` property controls the appearance of the character, such as its font and color. See Section 38.12 [Faces], page 325. The value of the property can be the following:

- A face name (a symbol or string).
- A property list of face attributes. This has the form (*keyword value* ...), where each *keyword* is a face attribute name and *value* is a meaningful value for that attribute. With this feature, you do not need to create a face each time you want to specify a particular attribute for certain text. See Section 38.12.2 [Face Attributes], page 327.
- A list, where each element uses one of the two forms listed above.

Font Lock mode (see Section 23.6 [Font Lock Mode], page 429, vol. 1) works in most buffers by dynamically updating the `face` property of characters based on the context.

`font-lock-face`
This property specifies a value for the `face` property that Font Lock mode should apply to the underlying text. It is one of the fontification methods used by Font Lock mode, and is useful for special modes that implement their own highlighting. See Section 23.6.6 [Precalculated Fontification], page 436, vol. 1. When Font Lock mode is disabled, `font-lock-face` has no effect.

`mouse-face`
This property is used instead of `face` when the mouse is on or near the character. For this purpose, "near" means that all text between the character and where the mouse is have the same `mouse-face` property value.

Emacs ignores all face attributes from the `mouse-face` property that alter the text size (e.g. `:height`, `:weight`, and `:slant`). Those attributes are always the same as for the unhighlighted text.

`fontified`

This property says whether the text is ready for display. If `nil`, Emacs's redisplay routine calls the functions in `fontification-functions` (see Section 38.12.7 [Auto Faces], page 336) to prepare this part of the buffer before it is displayed. It is used internally by the "just in time" font locking code.

`display` This property activates various features that change the way text is displayed. For example, it can make text appear taller or shorter, higher or lower, wider or narrow, or replaced with an image. See Section 38.15 [Display Property], page 350.

`help-echo`

If text has a string as its `help-echo` property, then when you move the mouse onto that text, Emacs displays that string in the echo area, or in the tooltip window (see Section "Tooltips" in *The GNU Emacs Manual*).

If the value of the `help-echo` property is a function, that function is called with three arguments, *window*, *object* and *pos* and should return a help string or `nil` for none. The first argument, *window* is the window in which the help was found. The second, *object*, is the buffer, overlay or string which had the `help-echo` property. The *pos* argument is as follows:

- If *object* is a buffer, *pos* is the position in the buffer.
- If *object* is an overlay, that overlay has a `help-echo` property, and *pos* is the position in the overlay's buffer.
- If *object* is a string (an overlay string or a string displayed with the `display` property), *pos* is the position in that string.

If the value of the `help-echo` property is neither a function nor a string, it is evaluated to obtain a help string.

You can alter the way help text is displayed by setting the variable `show-help-function` (see [Help display], page 167).

This feature is used in the mode line and for other active text.

`keymap` The `keymap` property specifies an additional keymap for commands. When this keymap applies, it is used for key lookup before the minor mode keymaps and before the buffer's local map. See Section 22.7 [Active Keymaps], page 367, vol. 1. If the property value is a symbol, the symbol's function definition is used as the keymap.

The property's value for the character before point applies if it is non-`nil` and rear-sticky, and the property's value for the character after point applies if it is non-`nil` and front-sticky. (For mouse clicks, the position of the click is used instead of the position of point.)

`local-map`

This property works like `keymap` except that it specifies a keymap to use *instead of* the buffer's local map. For most purposes (perhaps all purposes), it is better to use the `keymap` property.

`syntax-table`
: The `syntax-table` property overrides what the syntax table says about this particular character. See Section 35.4 [Syntax Properties], page 240.

`read-only`
: If a character has the property `read-only`, then modifying that character is not allowed. Any command that would do so gets an error, `text-read-only`. If the property value is a string, that string is used as the error message.

  Insertion next to a read-only character is an error if inserting ordinary text there would inherit the `read-only` property due to stickiness. Thus, you can control permission to insert next to read-only text by controlling the stickiness. See Section 32.19.6 [Sticky Properties], page 167.

  Since changing properties counts as modifying the buffer, it is not possible to remove a `read-only` property unless you know the special trick: bind `inhibit-read-only` to a non-`nil` value and then remove the property. See Section 27.7 [Read Only Buffers], page 9.

`invisible`
: A non-`nil` `invisible` property can make a character invisible on the screen. See Section 38.6 [Invisible Text], page 309, for details.

`intangible`
: If a group of consecutive characters have equal and non-`nil` `intangible` properties, then you cannot place point between them. If you try to move point forward into the group, point actually moves to the end of the group. If you try to move point backward into the group, point actually moves to the start of the group.

  If consecutive characters have unequal non-`nil` `intangible` properties, they belong to separate groups; each group is separately treated as described above.

  When the variable `inhibit-point-motion-hooks` is non-`nil`, the `intangible` property is ignored.

  Beware: this property operates at a very low level, and affects a lot of code in unexpected ways. So use it with extreme caution. A common misuse is to put an intangible property on invisible text, which is actually unnecessary since the command loop will move point outside of the invisible text at the end of each command anyway. See Section 21.6 [Adjusting Point], page 326, vol. 1.

`field`
: Consecutive characters with the same `field` property constitute a *field*. Some motion functions including `forward-word` and `beginning-of-line` stop moving at a field boundary. See Section 32.19.9 [Fields], page 172.

`cursor`
: Normally, the cursor is displayed at the beginning or the end of any overlay and text property strings present at the current buffer position. You can place the cursor on any desired character of these strings by giving that character a non-`nil` `cursor` text property. In addition, if the value of the `cursor` property is an integer number, it specifies the number of buffer's character positions, starting with the position where the overlay or the `display` property begins, for which the cursor should be displayed on that character. Specifically, if the value of the `cursor` property of a character is the number n, the cursor will be displayed on

this character for any buffer position in the range `[`*ovpos*`..`*ovpos*`+`*n*`)`, where *ovpos* is the overlay's starting position given by `overlay-start` (see Section 38.9.1 [Managing Overlays], page 316), or the position where the `display` text property begins in the buffer.

In other words, the string character with the `cursor` property of any non-`nil` value is the character where to display the cursor. The value of the property says for which buffer positions to display the cursor there. If the value is an integer number *n*, the cursor is displayed there when point is anywhere between the beginning of the overlay or `display` property and *n* positions after that. If the value is anything else and non-`nil`, the cursor is displayed there only when point is at the beginning of the `display` property or at `overlay-start`.

When the buffer has many overlay strings (e.g., see Section 38.9.2 [Overlay Properties], page 318) or `display` properties that are strings, it is a good idea to use the `cursor` property on these strings to cue the Emacs display about the places where to put the cursor while traversing these strings. This directly communicates to the display engine where the Lisp program wants to put the cursor, or where the user would expect the cursor.

`pointer` This specifies a specific pointer shape when the mouse pointer is over this text or image. See Section 29.17 [Pointer Shape], page 90, for possible pointer shapes.

`line-spacing`
A newline can have a `line-spacing` text or overlay property that controls the height of the display line ending with that newline. The property value overrides the default frame line spacing and the buffer local `line-spacing` variable. See Section 38.11 [Line Height], page 324.

`line-height`
A newline can have a `line-height` text or overlay property that controls the total height of the display line ending in that newline. See Section 38.11 [Line Height], page 324.

`wrap-prefix`
If text has a `wrap-prefix` property, the prefix it defines will be added at display time to the beginning of every continuation line due to text wrapping (so if lines are truncated, the wrap-prefix is never used). It may be a string or an image (see Section 38.15.4 [Other Display Specs], page 353), or a stretch of whitespace such as specified by the `:width` or `:align-to` display properties (see Section 38.15.2 [Specified Space], page 351).

A wrap-prefix may also be specified for an entire buffer using the `wrap-prefix` buffer-local variable (however, a `wrap-prefix` text-property takes precedence over the value of the `wrap-prefix` variable). See Section 38.3 [Truncation], page 300.

`line-prefix`
If text has a `line-prefix` property, the prefix it defines will be added at display time to the beginning of every non-continuation line. It may be a string or an image (see Section 38.15.4 [Other Display Specs], page 353), or a stretch of whitespace such as specified by the `:width` or `:align-to` display properties (see Section 38.15.2 [Specified Space], page 351).

A line-prefix may also be specified for an entire buffer using the `line-prefix` buffer-local variable (however, a `line-prefix` text-property takes precedence over the value of the `line-prefix` variable). See Section 38.3 [Truncation], page 300.

`modification-hooks`
: If a character has the property `modification-hooks`, then its value should be a list of functions; modifying that character calls all of those functions before the actual modification. Each function receives two arguments: the beginning and end of the part of the buffer being modified. Note that if a particular modification hook function appears on several characters being modified by a single primitive, you can't predict how many times the function will be called. Furthermore, insertion will not modify any existing character, so this hook will only be run when removing some characters, replacing them with others, or changing their text-properties.

  If these functions modify the buffer, they should bind `inhibit-modification-hooks` to `t` around doing so, to avoid confusing the internal mechanism that calls these hooks.

  Overlays also support the `modification-hooks` property, but the details are somewhat different (see Section 38.9.2 [Overlay Properties], page 318).

`insert-in-front-hooks`
`insert-behind-hooks`
: The operation of inserting text in a buffer also calls the functions listed in the `insert-in-front-hooks` property of the following character and in the `insert-behind-hooks` property of the preceding character. These functions receive two arguments, the beginning and end of the inserted text. The functions are called *after* the actual insertion takes place.

  See also Section 32.27 [Change Hooks], page 180, for other hooks that are called when you change text in a buffer.

`point-entered`
`point-left`
: The special properties `point-entered` and `point-left` record hook functions that report motion of point. Each time point moves, Emacs compares these two property values:

  - the `point-left` property of the character after the old location, and
  - the `point-entered` property of the character after the new location.

  If these two values differ, each of them is called (if not `nil`) with two arguments: the old value of point, and the new one.

  The same comparison is made for the characters before the old and new locations. The result may be to execute two `point-left` functions (which may be the same function) and/or two `point-entered` functions (which may be the same function). In any case, all the `point-left` functions are called first, followed by all the `point-entered` functions.

It is possible to use `char-after` to examine characters at various buffer positions without moving point to those positions. Only an actual change in the value of point runs these hook functions.

The variable `inhibit-point-motion-hooks` can inhibit running the `point-left` and `point-entered` hooks, see [Inhibit point motion hooks], page 167.

`composition`
: This text property is used to display a sequence of characters as a single glyph composed from components. But the value of the property itself is completely internal to Emacs and should not be manipulated directly by, for instance, `put-text-property`.

`inhibit-point-motion-hooks` [Variable]
: When this variable is non-`nil`, `point-left` and `point-entered` hooks are not run, and the `intangible` property has no effect. Do not set this variable globally; bind it with `let`.

`show-help-function` [Variable]
: If this variable is non-`nil`, it specifies a function called to display help strings. These may be `help-echo` properties, menu help strings (see Section 22.17.1.1 [Simple Menu Items], page 384, vol. 1, see Section 22.17.1.2 [Extended Menu Items], page 385, vol. 1), or tool bar help strings (see Section 22.17.6 [Tool Bar], page 392, vol. 1). The specified function is called with one argument, the help string to display. Tooltip mode (see Section "Tooltips" in *The GNU Emacs Manual*) provides an example.

32.19.5 Formatted Text Properties

These text properties affect the behavior of the fill commands. They are used for representing formatted text. See Section 32.11 [Filling], page 140, and Section 32.12 [Margins], page 143.

`hard`
: If a newline character has this property, it is a "hard" newline. The fill commands do not alter hard newlines and do not move words across them. However, this property takes effect only if the `use-hard-newlines` minor mode is enabled. See Section "Hard and Soft Newlines" in *The GNU Emacs Manual*.

`right-margin`
: This property specifies an extra right margin for filling this part of the text.

`left-margin`
: This property specifies an extra left margin for filling this part of the text.

`justification`
: This property specifies the style of justification for filling this part of the text.

32.19.6 Stickiness of Text Properties

Self-inserting characters normally take on the same properties as the preceding character. This is called *inheritance* of properties.

A Lisp program can do insertion with inheritance or without, depending on the choice of insertion primitive. The ordinary text insertion functions, such as `insert`, do not inherit any properties. They insert text with precisely the properties of the string being inserted,

and no others. This is correct for programs that copy text from one context to another—for example, into or out of the kill ring. To insert with inheritance, use the special primitives described in this section. Self-inserting characters inherit properties because they work using these primitives.

When you do insertion with inheritance, *which* properties are inherited, and from where, depends on which properties are *sticky*. Insertion after a character inherits those of its properties that are *rear-sticky*. Insertion before a character inherits those of its properties that are *front-sticky*. When both sides offer different sticky values for the same property, the previous character's value takes precedence.

By default, a text property is rear-sticky but not front-sticky; thus, the default is to inherit all the properties of the preceding character, and nothing from the following character.

You can control the stickiness of various text properties with two specific text properties, `front-sticky` and `rear-nonsticky`, and with the variable `text-property-default-nonsticky`. You can use the variable to specify a different default for a given property. You can use those two text properties to make any specific properties sticky or nonsticky in any particular part of the text.

If a character's `front-sticky` property is `t`, then all its properties are front-sticky. If the `front-sticky` property is a list, then the sticky properties of the character are those whose names are in the list. For example, if a character has a `front-sticky` property whose value is `(face read-only)`, then insertion before the character can inherit its `face` property and its `read-only` property, but no others.

The `rear-nonsticky` property works the opposite way. Most properties are rear-sticky by default, so the `rear-nonsticky` property says which properties are *not* rear-sticky. If a character's `rear-nonsticky` property is `t`, then none of its properties are rear-sticky. If the `rear-nonsticky` property is a list, properties are rear-sticky *unless* their names are in the list.

`text-property-default-nonsticky` [Variable]
: This variable holds an alist which defines the default rear-stickiness of various text properties. Each element has the form `(`*`property`* `.` *`nonstickiness`*`)`, and it defines the stickiness of a particular text property, *property*.

  If *nonstickiness* is non-`nil`, this means that the property *property* is rear-nonsticky by default. Since all properties are front-nonsticky by default, this makes *property* nonsticky in both directions by default.

  The text properties `front-sticky` and `rear-nonsticky`, when used, take precedence over the default *nonstickiness* specified in `text-property-default-nonsticky`.

Here are the functions that insert text with inheritance of properties:

`insert-and-inherit` **&rest** *strings* [Function]
: Insert the strings *strings*, just like the function `insert`, but inherit any sticky properties from the adjoining text.

`insert-before-markers-and-inherit` **&rest** *strings* [Function]
: Insert the strings *strings*, just like the function `insert-before-markers`, but inherit any sticky properties from the adjoining text.

See Section 32.4 [Insertion], page 126, for the ordinary insertion functions which do not inherit.

32.19.7 Lazy Computation of Text Properties

Instead of computing text properties for all the text in the buffer, you can arrange to compute the text properties for parts of the text when and if something depends on them.

The primitive that extracts text from the buffer along with its properties is `buffer-substring`. Before examining the properties, this function runs the abnormal hook `buffer-access-fontify-functions`.

`buffer-access-fontify-functions` [Variable]

This variable holds a list of functions for computing text properties. Before `buffer-substring` copies the text and text properties for a portion of the buffer, it calls all the functions in this list. Each of the functions receives two arguments that specify the range of the buffer being accessed. (The buffer itself is always the current buffer.)

The function `buffer-substring-no-properties` does not call these functions, since it ignores text properties anyway.

In order to prevent the hook functions from being called more than once for the same part of the buffer, you can use the variable `buffer-access-fontified-property`.

`buffer-access-fontified-property` [Variable]

If this variable's value is non-`nil`, it is a symbol which is used as a text property name. A non-`nil` value for that text property means, "the other text properties for this character have already been computed".

If all the characters in the range specified for `buffer-substring` have a non-`nil` value for this property, `buffer-substring` does not call the `buffer-access-fontify-functions` functions. It assumes these characters already have the right text properties, and just copies the properties they already have.

The normal way to use this feature is that the `buffer-access-fontify-functions` functions add this property, as well as others, to the characters they operate on. That way, they avoid being called over and over for the same text.

32.19.8 Defining Clickable Text

Clickable text is text that can be clicked, with either the mouse or via a keyboard command, to produce some result. Many major modes use clickable text to implement textual hyperlinks, or *links* for short.

The easiest way to insert and manipulate links is to use the `button` package. See Section 38.17 [Buttons], page 366. In this section, we will explain how to manually set up clickable text in a buffer, using text properties. For simplicity, we will refer to the clickable text as a *link*.

Implementing a link involves three separate steps: (1) indicating clickability when the mouse moves over the link; (2) making `RET` or *`Mouse-2`* on that link do something; and (3) setting up a `follow-link` condition so that the link obeys `mouse-1-click-follows-link`.

To indicate clickability, add the `mouse-face` text property to the text of the link; then Emacs will highlight the link when the mouse moves over it. In addition, you should define

a tooltip or echo area message, using the `help-echo` text property. See Section 32.19.4 [Special Properties], page 162. For instance, here is how Dired indicates that file names are clickable:

```
(if (dired-move-to-filename)
    (add-text-properties
      (point)
      (save-excursion
        (dired-move-to-end-of-filename)
        (point))
      '(mouse-face highlight
        help-echo "mouse-2: visit this file in other window")))
```

To make the link clickable, bind `RET` and *`Mouse-2`* to commands that perform the desired action. Each command should check to see whether it was called on a link, and act accordingly. For instance, Dired's major mode keymap binds *`Mouse-2`* to the following command:

```
(defun dired-mouse-find-file-other-window (event)
  "In Dired, visit the file or directory name you click on."
  (interactive "e")
  (let ((window (posn-window (event-end event)))
        (pos (posn-point (event-end event)))
        file)
    (if (not (windowp window))
        (error "No file chosen"))
    (with-current-buffer (window-buffer window)
      (goto-char pos)
      (setq file (dired-get-file-for-visit)))
    (if (file-directory-p file)
        (or (and (cdr dired-subdir-alist)
                 (dired-goto-subdir file))
            (progn
              (select-window window)
              (dired-other-window file)))
      (select-window window)
      (find-file-other-window (file-name-sans-versions file t)))))
```

This command uses the functions `posn-window` and `posn-point` to determine where the click occurred, and `dired-get-file-for-visit` to determine which file to visit.

Instead of binding the mouse command in a major mode keymap, you can bind it within the link text, using the `keymap` text property (see Section 32.19.4 [Special Properties], page 162). For instance:

```
(let ((map (make-sparse-keymap)))
  (define-key map [mouse-2] 'operate-this-button)
  (put-text-property link-start link-end 'keymap map))
```

With this method, you can easily define different commands for different links. Furthermore, the global definition of `RET` and *`Mouse-2`* remain available for the rest of the text in the buffer.

The basic Emacs command for clicking on links is *`Mouse-2`*. However, for compatibility with other graphical applications, Emacs also recognizes *`Mouse-1`* clicks on links, provided the user clicks on the link quickly without moving the mouse. This behavior is controlled by the user option `mouse-1-click-follows-link`. See Section "Mouse References" in *The GNU Emacs Manual*.

To set up the link so that it obeys `mouse-1-click-follows-link`, you must either (1) apply a `follow-link` text or overlay property to the link text, or (2) bind the `follow-link` event to a keymap (which can be a major mode keymap or a local keymap specified via the `keymap` text property). The value of the `follow-link` property, or the binding for the `follow-link` event, acts as a "condition" for the link action. This condition tells Emacs two things: the circumstances under which a *`Mouse-1`* click should be regarded as occurring "inside" the link, and how to compute an "action code" that says what to translate the *`Mouse-1`* click into. The link action condition can be one of the following:

`mouse-face`
: If the condition is the symbol `mouse-face`, a position is inside a link if there is a non-`nil` `mouse-face` property at that position. The action code is always `t`.

  For example, here is how Info mode handles `Mouse-1`:

  ```
  (define-key Info-mode-map [follow-link] 'mouse-face)
  ```

a function
: If the condition is a function, *func*, then a position *pos* is inside a link if (*`func pos`*) evaluates to non-`nil`. The value returned by *func* serves as the action code.

  For example, here is how pcvs enables *`Mouse-1`* to follow links on file names only:

  ```
  (define-key map [follow-link]
    (lambda (pos)
      (eq (get-char-property pos 'face) 'cvs-filename-face)))
  ```

anything else
: If the condition value is anything else, then the position is inside a link and the condition itself is the action code. Clearly, you should specify this kind of condition only when applying the condition via a text or property overlay on the link text (so that it does not apply to the entire buffer).

The action code tells *`Mouse-1`* how to follow the link:

a string or vector
: If the action code is a string or vector, the *`Mouse-1`* event is translated into the first element of the string or vector; i.e., the action of the *`Mouse-1`* click is the local or global binding of that character or symbol. Thus, if the action code is `"foo"`, *`Mouse-1`* translates into *`f`*. If it is `[foo]`, *`Mouse-1`* translates into `foo`.

anything else
: For any other non-`nil` action code, the *`Mouse-1`* event is translated into a *`Mouse-2`* event at the same position.

To define *`Mouse-1`* to activate a button defined with `define-button-type`, give the button a `follow-link` property. The property value should be a link action condition, as described above. See Section 38.17 [Buttons], page 366. For example, here is how Help mode handles *`Mouse-1`*:

```
(define-button-type 'help-xref
  'follow-link t
  'action #'help-button-action)
```

To define *`Mouse-1`* on a widget defined with `define-widget`, give the widget a `:follow-link` property. The property value should be a link action condition, as described above.

For example, here is how the `link` widget specifies that a `Mouse-1` click shall be translated to `RET`:

```
(define-widget 'link 'item
  "An embedded link."
  :button-prefix 'widget-link-prefix
  :button-suffix 'widget-link-suffix
  :follow-link "\C-m"
  :help-echo "Follow the link."
  :format "%[%t%]")
```

`mouse-on-link-p` *pos* [Function]

This function returns non-`nil` if position *pos* in the current buffer is on a link. *pos* can also be a mouse event location, as returned by `event-start` (see Section 21.7.13 [Accessing Mouse], page 338, vol. 1).

32.19.9 Defining and Using Fields

A field is a range of consecutive characters in the buffer that are identified by having the same value (comparing with `eq`) of the `field` property (either a text-property or an overlay property). This section describes special functions that are available for operating on fields.

You specify a field with a buffer position, *pos*. We think of each field as containing a range of buffer positions, so the position you specify stands for the field containing that position.

When the characters before and after *pos* are part of the same field, there is no doubt which field contains *pos*: the one those characters both belong to. When *pos* is at a boundary between fields, which field it belongs to depends on the stickiness of the `field` properties of the two surrounding characters (see Section 32.19.6 [Sticky Properties], page 167). The field whose property would be inherited by text inserted at *pos* is the field that contains *pos*.

There is an anomalous case where newly inserted text at *pos* would not inherit the `field` property from either side. This happens if the previous character's `field` property is not rear-sticky, and the following character's `field` property is not front-sticky. In this case, *pos* belongs to neither the preceding field nor the following field; the field functions treat it as belonging to an empty field whose beginning and end are both at *pos*.

In all of these functions, if *pos* is omitted or `nil`, the value of point is used by default. If narrowing is in effect, then *pos* should fall within the accessible portion. See Section 30.4 [Narrowing], page 109.

`field-beginning` **&optional** *pos escape-from-edge limit* [Function]

This function returns the beginning of the field specified by *pos*.

If *pos* is at the beginning of its field, and *escape-from-edge* is non-`nil`, then the return value is always the beginning of the preceding field that *ends* at *pos*, regardless of the stickiness of the `field` properties around *pos*.

If *limit* is non-`nil`, it is a buffer position; if the beginning of the field is before *limit*, then *limit* will be returned instead.

`field-end` **&optional** *pos escape-from-edge limit* [Function]

This function returns the end of the field specified by *pos*.

If *pos* is at the end of its field, and *escape-from-edge* is non-`nil`, then the return value is always the end of the following field that *begins* at *pos*, regardless of the stickiness of the `field` properties around *pos*.

If *limit* is non-`nil`, it is a buffer position; if the end of the field is after *limit*, then *limit* will be returned instead.

`field-string` **&optional** *pos* [Function]
This function returns the contents of the field specified by *pos*, as a string.

`field-string-no-properties` **&optional** *pos* [Function]
This function returns the contents of the field specified by *pos*, as a string, discarding text properties.

`delete-field` **&optional** *pos* [Function]
This function deletes the text of the field specified by *pos*.

`constrain-to-field` *new-pos old-pos* **&optional** *escape-from-edge only-in-line inhibit-capture-property* [Function]
This function "constrains" *new-pos* to the field that *old-pos* belongs to—in other words, it returns the position closest to *new-pos* that is in the same field as *old-pos*.

If *new-pos* is `nil`, then `constrain-to-field` uses the value of point instead, and moves point to the resulting position in addition to returning that position.

If *old-pos* is at the boundary of two fields, then the acceptable final positions depend on the argument *escape-from-edge*. If *escape-from-edge* is `nil`, then *new-pos* must be in the field whose `field` property equals what new characters inserted at *old-pos* would inherit. (This depends on the stickiness of the `field` property for the characters before and after *old-pos*.) If *escape-from-edge* is non-`nil`, *new-pos* can be anywhere in the two adjacent fields. Additionally, if two fields are separated by another field with the special value `boundary`, then any point within this special field is also considered to be "on the boundary".

Commands like `C-a` with no argument, that normally move backward to a specific kind of location and stay there once there, probably should specify `nil` for *escape-from-edge*. Other motion commands that check fields should probably pass `t`.

If the optional argument *only-in-line* is non-`nil`, and constraining *new-pos* in the usual way would move it to a different line, *new-pos* is returned unconstrained. This used in commands that move by line, such as `next-line` and `beginning-of-line`, so that they respect field boundaries only in the case where they can still move to the right line.

If the optional argument *inhibit-capture-property* is non-`nil`, and *old-pos* has a non-`nil` property of that name, then any field boundaries are ignored.

You can cause `constrain-to-field` to ignore all field boundaries (and so never constrain anything) by binding the variable `inhibit-field-text-motion` to a non-`nil` value.

32.19.10 Why Text Properties are not Intervals

Some editors that support adding attributes to text in the buffer do so by letting the user specify "intervals" within the text, and adding the properties to the intervals. Those editors permit the user or the programmer to determine where individual intervals start and end. We deliberately provided a different sort of interface in Emacs Lisp to avoid certain paradoxical behavior associated with text modification.

If the actual subdivision into intervals is meaningful, that means you can distinguish between a buffer that is just one interval with a certain property, and a buffer containing the same text subdivided into two intervals, both of which have that property.

Suppose you take the buffer with just one interval and kill part of the text. The text remaining in the buffer is one interval, and the copy in the kill ring (and the undo list) becomes a separate interval. Then if you yank back the killed text, you get two intervals with the same properties. Thus, editing does not preserve the distinction between one interval and two.

Suppose we "fix" this problem by coalescing the two intervals when the text is inserted. That works fine if the buffer originally was a single interval. But suppose instead that we have two adjacent intervals with the same properties, and we kill the text of one interval and yank it back. The same interval-coalescence feature that rescues the other case causes trouble in this one: after yanking, we have just one interval. One again, editing does not preserve the distinction between one interval and two.

Insertion of text at the border between intervals also raises questions that have no satisfactory answer.

However, it is easy to arrange for editing to behave consistently for questions of the form, "What are the properties of this character?" So we have decided these are the only questions that make sense; we have not implemented asking questions about where intervals start or end.

In practice, you can usually use the text property search functions in place of explicit interval boundaries. You can think of them as finding the boundaries of intervals, assuming that intervals are always coalesced whenever possible. See Section 32.19.3 [Property Search], page 160.

Emacs also provides explicit intervals as a presentation feature; see Section 38.9 [Overlays], page 315.

32.20 Substituting for a Character Code

The following functions replace characters within a specified region based on their character codes.

`subst-char-in-region` *start end old-char new-char* **&optional** *noundo* [Function]

This function replaces all occurrences of the character *old-char* with the character *new-char* in the region of the current buffer defined by *start* and *end*.

If *noundo* is non-`nil`, then `subst-char-in-region` does not record the change for undo and does not mark the buffer as modified. This was useful for controlling the old selective display feature (see Section 38.7 [Selective Display], page 312).

`subst-char-in-region` does not move point and returns `nil`.

```
---------- Buffer: foo ----------
This is the contents of the buffer before.
---------- Buffer: foo ----------

(subst-char-in-region 1 20 ?i ?X)
     ⇒ nil

---------- Buffer: foo ----------
ThXs Xs the contents of the buffer before.
---------- Buffer: foo ----------
```

`translate-region` *start end table* [Command]

This function applies a translation table to the characters in the buffer between positions *start* and *end*.

The translation table *table* is a string or a char-table; `(aref table ochar)` gives the translated character corresponding to *ochar*. If *table* is a string, any characters with codes larger than the length of *table* are not altered by the translation.

The return value of `translate-region` is the number of characters that were actually changed by the translation. This does not count characters that were mapped into themselves in the translation table.

32.21 Registers

A register is a sort of variable used in Emacs editing that can hold a variety of different kinds of values. Each register is named by a single character. All ASCII characters and their meta variants (but with the exception of `C-g`) can be used to name registers. Thus, there are 255 possible registers. A register is designated in Emacs Lisp by the character that is its name.

`register-alist` [Variable]

This variable is an alist of elements of the form `(name . contents)`. Normally, there is one element for each Emacs register that has been used.

The object *name* is a character (an integer) identifying the register.

The *contents* of a register can have several possible types:

a number
: A number stands for itself. If `insert-register` finds a number in the register, it converts the number to decimal.

a marker
: A marker represents a buffer position to jump to.

a string
: A string is text saved in the register.

a rectangle
: A rectangle is represented by a list of strings.

`(window-configuration position)`
: This represents a window configuration to restore in one frame, and a position to jump to in the current buffer.

(*frame-configuration position*)
: This represents a frame configuration to restore, and a position to jump to in the current buffer.

(file *filename*)
: This represents a file to visit; jumping to this value visits file *filename*.

(file-query *filename position*)
: This represents a file to visit and a position in it; jumping to this value visits file *filename* and goes to buffer position *position*. Restoring this type of position asks the user for confirmation first.

The functions in this section return unpredictable values unless otherwise stated.

`get-register` *reg* [Function]
: This function returns the contents of the register *reg*, or `nil` if it has no contents.

`set-register` *reg value* [Function]
: This function sets the contents of register *reg* to *value*. A register can be set to any value, but the other register functions expect only certain data types. The return value is *value*.

`view-register` *reg* [Command]
: This command displays what is contained in register *reg*.

`insert-register` *reg* **&optional** *beforep* [Command]
: This command inserts contents of register *reg* into the current buffer.

  Normally, this command puts point before the inserted text, and the mark after it. However, if the optional second argument *beforep* is non-`nil`, it puts the mark before and point after. You can pass a non-`nil` second argument *beforep* to this function interactively by supplying any prefix argument.

  If the register contains a rectangle, then the rectangle is inserted with its upper left corner at point. This means that text is inserted in the current line and underneath it on successive lines.

  If the register contains something other than saved text (a string) or a rectangle (a list), currently useless things happen. This may be changed in the future.

32.22 Transposition of Text

This function can be used to transpose stretches of text:

`transpose-regions` *start1 end1 start2 end2* **&optional** *leave-markers* [Function]
: This function exchanges two nonoverlapping portions of the buffer. Arguments *start1* and *end1* specify the bounds of one portion and arguments *start2* and *end2* specify the bounds of the other portion.

  Normally, `transpose-regions` relocates markers with the transposed text; a marker previously positioned within one of the two transposed portions moves along with that portion, thus remaining between the same two characters in their new position. However, if *leave-markers* is non-`nil`, `transpose-regions` does not do this—it leaves all markers unrelocated.

32.23 Base 64 Encoding

Base 64 code is used in email to encode a sequence of 8-bit bytes as a longer sequence of ASCII graphic characters. It is defined in Internet RFC[1]2045. This section describes the functions for converting to and from this code.

`base64-encode-region` *beg end* **&optional** *no-line-break* [Command]
: This function converts the region from *beg* to *end* into base 64 code. It returns the length of the encoded text. An error is signaled if a character in the region is multibyte, i.e. in a multibyte buffer the region must contain only characters from the charsets `ascii`, `eight-bit-control` and `eight-bit-graphic`.

  Normally, this function inserts newline characters into the encoded text, to avoid overlong lines. However, if the optional argument *no-line-break* is non-`nil`, these newlines are not added, so the output is just one long line.

`base64-encode-string` *string* **&optional** *no-line-break* [Function]
: This function converts the string *string* into base 64 code. It returns a string containing the encoded text. As for `base64-encode-region`, an error is signaled if a character in the string is multibyte.

  Normally, this function inserts newline characters into the encoded text, to avoid overlong lines. However, if the optional argument *no-line-break* is non-`nil`, these newlines are not added, so the result string is just one long line.

`base64-decode-region` *beg end* [Command]
: This function converts the region from *beg* to *end* from base 64 code into the corresponding decoded text. It returns the length of the decoded text.

  The decoding functions ignore newline characters in the encoded text.

`base64-decode-string` *string* [Function]
: This function converts the string *string* from base 64 code into the corresponding decoded text. It returns a unibyte string containing the decoded text.

  The decoding functions ignore newline characters in the encoded text.

32.24 Checksum/Hash

Emacs has built-in support for computing *cryptographic hashes*. A cryptographic hash, or *checksum*, is a digital "fingerprint" of a piece of data (e.g. a block of text) which can be used to check that you have an unaltered copy of that data.

Emacs supports several common cryptographic hash algorithms: MD5, SHA-1, SHA-2, SHA-224, SHA-256, SHA-384 and SHA-512. MD5 is the oldest of these algorithms, and is commonly used in *message digests* to check the integrity of messages transmitted over a network. MD5 is not "collision resistant" (i.e. it is possible to deliberately design different pieces of data which have the same MD5 hash), so you should not used it for anything security-related. A similar theoretical weakness also exists in SHA-1. Therefore, for security-related applications you should use the other hash types, such as SHA-2.

[1] An RFC, an acronym for *Request for Comments*, is a numbered Internet informational document describing a standard. RFCs are usually written by technical experts acting on their own initiative, and are traditionally written in a pragmatic, experience-driven manner.

`secure-hash` *algorithm object* **&optional** *start end binary* [Function]
This function returns a hash for *object*. The argument *algorithm* is a symbol stating which hash to compute: one of `md5`, `sha1`, `sha224`, `sha256`, `sha384` or `sha512`. The argument *object* should be a buffer or a string.

The optional arguments *start* and *end* are character positions specifying the portion of *object* to compute the message digest for. If they are `nil` or omitted, the hash is computed for the whole of *object*.

If the argument *binary* is omitted or `nil`, the function returns the *text form* of the hash, as an ordinary Lisp string. If *binary* is non-`nil`, it returns the hash in *binary form*, as a sequence of bytes stored in a unibyte string.

This function does not compute the hash directly from the internal representation of *object*'s text (see Section 33.1 [Text Representations], page 182). Instead, it encodes the text using a coding system (see Section 33.9 [Coding Systems], page 193), and computes the hash from that encoded text. If *object* is a buffer, the coding system used is the one which would be chosen by default for writing the text into a file. If *object* is a string, the user's preferred coding system is used (see Section "Recognize Coding" in *GNU Emacs Manual*).

`md5` *object* **&optional** *start end coding-system noerror* [Function]
This function returns an MD5 hash. It is semi-obsolete, since for most purposes it is equivalent to calling `secure-hash` with `md5` as the *algorithm* argument. The *object*, *start* and *end* arguments have the same meanings as in `secure-hash`.

If *coding-system* is non-`nil`, it specifies a coding system to use to encode the text; if omitted or `nil`, the default coding system is used, like in `secure-hash`.

Normally, `md5` signals an error if the text can't be encoded using the specified or chosen coding system. However, if *noerror* is non-`nil`, it silently uses `raw-text` coding instead.

32.25 Parsing HTML and XML

When Emacs is compiled with libxml2 support, the following functions are available to parse HTML or XML text into Lisp object trees.

`libxml-parse-html-region` *start end* **&optional** *base-url* [Function]
This function parses the text between *start* and *end* as HTML, and returns a list representing the HTML *parse tree*. It attempts to handle "real world" HTML by robustly coping with syntax mistakes.

The optional argument *base-url*, if non-`nil`, should be a string specifying the base URL for relative URLs occurring in links.

In the parse tree, each HTML node is represented by a list in which the first element is a symbol representing the node name, the second element is an alist of node attributes, and the remaining elements are the subnodes.

The following example demonstrates this. Given this (malformed) HTML document:

```
<html><head></head><body width=101><div class=thing>Foo<div>Yes
```

A call to `libxml-parse-html-region` returns this:

```
(html ()
  (head ())
  (body ((width . "101"))
   (div ((class . "thing"))
    "Foo"
    (div ()
      "Yes"))))
```

`libxml-parse-xml-region` *start end* **&optional** *base-url* [Function]
This function is the same as `libxml-parse-html-region`, except that it parses the text as XML rather than HTML (so it is stricter about syntax).

32.26 Atomic Change Groups

In database terminology, an *atomic* change is an indivisible change—it can succeed entirely or it can fail entirely, but it cannot partly succeed. A Lisp program can make a series of changes to one or several buffers as an *atomic change group*, meaning that either the entire series of changes will be installed in their buffers or, in case of an error, none of them will be.

To do this for one buffer, the one already current, simply write a call to `atomic-change-group` around the code that makes the changes, like this:

```
(atomic-change-group
  (insert foo)
  (delete-region x y))
```

If an error (or other nonlocal exit) occurs inside the body of `atomic-change-group`, it unmakes all the changes in that buffer that were during the execution of the body. This kind of change group has no effect on any other buffers—any such changes remain.

If you need something more sophisticated, such as to make changes in various buffers constitute one atomic group, you must directly call lower-level functions that `atomic-change-group` uses.

`prepare-change-group` **&optional** *buffer* [Function]
This function sets up a change group for buffer *buffer*, which defaults to the current buffer. It returns a "handle" that represents the change group. You must use this handle to activate the change group and subsequently to finish it.

To use the change group, you must *activate* it. You must do this before making any changes in the text of *buffer*.

`activate-change-group` *handle* [Function]
This function activates the change group that *handle* designates.

After you activate the change group, any changes you make in that buffer become part of it. Once you have made all the desired changes in the buffer, you must *finish* the change group. There are two ways to do this: you can either accept (and finalize) all the changes, or cancel them all.

`accept-change-group` *handle* [Function]
This function accepts all the changes in the change group specified by *handle*, making them final.

`cancel-change-group` *handle* [Function]
: This function cancels and undoes all the changes in the change group specified by *handle*.

Your code should use `unwind-protect` to make sure the group is always finished. The call to `activate-change-group` should be inside the `unwind-protect`, in case the user types *C-g* just after it runs. (This is one reason why `prepare-change-group` and `activate-change-group` are separate functions, because normally you would call `prepare-change-group` before the start of that `unwind-protect`.) Once you finish the group, don't use the handle again—in particular, don't try to finish the same group twice.

To make a multibuffer change group, call `prepare-change-group` once for each buffer you want to cover, then use `nconc` to combine the returned values, like this:

```
(nconc (prepare-change-group buffer-1)
       (prepare-change-group buffer-2))
```

You can then activate the multibuffer change group with a single call to `activate-change-group`, and finish it with a single call to `accept-change-group` or `cancel-change-group`.

Nested use of several change groups for the same buffer works as you would expect. Non-nested use of change groups for the same buffer will get Emacs confused, so don't let it happen; the first change group you start for any given buffer should be the last one finished.

32.27 Change Hooks

These hook variables let you arrange to take notice of all changes in all buffers (or in a particular buffer, if you make them buffer-local). See also Section 32.19.4 [Special Properties], page 162, for how to detect changes to specific parts of the text.

The functions you use in these hooks should save and restore the match data if they do anything that uses regular expressions; otherwise, they will interfere in bizarre ways with the editing operations that call them.

`before-change-functions` [Variable]
: This variable holds a list of functions to call before any buffer modification. Each function gets two arguments, the beginning and end of the region that is about to change, represented as integers. The buffer that is about to change is always the current buffer.

`after-change-functions` [Variable]
: This variable holds a list of functions to call after any buffer modification. Each function receives three arguments: the beginning and end of the region just changed, and the length of the text that existed before the change. All three arguments are integers. The buffer has been changed is always the current buffer.

  The length of the old text is the difference between the buffer positions before and after that text as it was before the change. As for the changed text, its length is simply the difference between the first two arguments.

Output of messages into the '`*Messages*`' buffer does not call these functions.

`combine-after-change-calls` *body...* [Macro]

The macro executes *body* normally, but arranges to call the after-change functions just once for a series of several changes—if that seems safe.

If a program makes several text changes in the same area of the buffer, using the macro `combine-after-change-calls` around that part of the program can make it run considerably faster when after-change hooks are in use. When the after-change hooks are ultimately called, the arguments specify a portion of the buffer including all of the changes made within the `combine-after-change-calls` body.

Warning: You must not alter the values of `after-change-functions` within the body of a `combine-after-change-calls` form.

Warning: if the changes you combine occur in widely scattered parts of the buffer, this will still work, but it is not advisable, because it may lead to inefficient behavior for some change hook functions.

`first-change-hook` [Variable]

This variable is a normal hook that is run whenever a buffer is changed that was previously in the unmodified state.

`inhibit-modification-hooks` [Variable]

If this variable is non-`nil`, all of the change hooks are disabled; none of them run. This affects all the hook variables described above in this section, as well as the hooks attached to certain special text properties (see Section 32.19.4 [Special Properties], page 162) and overlay properties (see Section 38.9.2 [Overlay Properties], page 318).

Also, this variable is bound to non-`nil` while running those same hook variables, so that by default modifying the buffer from a modification hook does not cause other modification hooks to be run. If you do want modification hooks to be run in a particular piece of code that is itself run from a modification hook, then rebind locally `inhibit-modification-hooks` to `nil`.

33 Non-ASCII Characters

This chapter covers the special issues relating to characters and how they are stored in strings and buffers.

33.1 Text Representations

Emacs buffers and strings support a large repertoire of characters from many different scripts, allowing users to type and display text in almost any known written language.

To support this multitude of characters and scripts, Emacs closely follows the *Unicode Standard*. The Unicode Standard assigns a unique number, called a *codepoint*, to each and every character. The range of codepoints defined by Unicode, or the Unicode *codespace*, is `0..#x10FFFF` (in hexadecimal notation), inclusive. Emacs extends this range with codepoints in the range `#x110000..#x3FFFFF`, which it uses for representing characters that are not unified with Unicode and *raw 8-bit bytes* that cannot be interpreted as characters. Thus, a character codepoint in Emacs is a 22-bit integer number.

To conserve memory, Emacs does not hold fixed-length 22-bit numbers that are codepoints of text characters within buffers and strings. Rather, Emacs uses a variable-length internal representation of characters, that stores each character as a sequence of 1 to 5 8-bit bytes, depending on the magnitude of its codepoint[1]. For example, any ASCII character takes up only 1 byte, a Latin-1 character takes up 2 bytes, etc. We call this representation of text *multibyte*.

Outside Emacs, characters can be represented in many different encodings, such as ISO-8859-1, GB-2312, Big-5, etc. Emacs converts between these external encodings and its internal representation, as appropriate, when it reads text into a buffer or a string, or when it writes text to a disk file or passes it to some other process.

Occasionally, Emacs needs to hold and manipulate encoded text or binary non-text data in its buffers or strings. For example, when Emacs visits a file, it first reads the file's text verbatim into a buffer, and only then converts it to the internal representation. Before the conversion, the buffer holds encoded text.

Encoded text is not really text, as far as Emacs is concerned, but rather a sequence of raw 8-bit bytes. We call buffers and strings that hold encoded text *unibyte* buffers and strings, because Emacs treats them as a sequence of individual bytes. Usually, Emacs displays unibyte buffers and strings as octal codes such as `\237`. We recommend that you never use unibyte buffers and strings except for manipulating encoded text or binary non-text data.

In a buffer, the buffer-local value of the variable `enable-multibyte-characters` specifies the representation used. The representation for a string is determined and recorded in the string when the string is constructed.

`enable-multibyte-characters` [Variable]

This variable specifies the current buffer's text representation. If it is non-`nil`, the buffer contains multibyte text; otherwise, it contains unibyte encoded text or binary non-text data.

[1] This internal representation is based on one of the encodings defined by the Unicode Standard, called *UTF-8*, for representing any Unicode codepoint, but Emacs extends UTF-8 to represent the additional codepoints it uses for raw 8-bit bytes and characters not unified with Unicode.

You cannot set this variable directly; instead, use the function `set-buffer-multibyte` to change a buffer's representation.

`position-bytes` *position* [Function]
Buffer positions are measured in character units. This function returns the byte-position corresponding to buffer position *position* in the current buffer. This is 1 at the start of the buffer, and counts upward in bytes. If *position* is out of range, the value is `nil`.

`byte-to-position` *byte-position* [Function]
Return the buffer position, in character units, corresponding to given *byte-position* in the current buffer. If *byte-position* is out of range, the value is `nil`. In a multibyte buffer, an arbitrary value of *byte-position* can be not at character boundary, but inside a multibyte sequence representing a single character; in this case, this function returns the buffer position of the character whose multibyte sequence includes *byte-position*. In other words, the value does not change for all byte positions that belong to the same character.

`multibyte-string-p` *string* [Function]
Return `t` if *string* is a multibyte string, `nil` otherwise.

`string-bytes` *string* [Function]
This function returns the number of bytes in *string*. If *string* is a multibyte string, this can be greater than `(length` *string*`)`.

`unibyte-string` **&rest** *bytes* [Function]
This function concatenates all its argument *bytes* and makes the result a unibyte string.

33.2 Converting Text Representations

Emacs can convert unibyte text to multibyte; it can also convert multibyte text to unibyte, provided that the multibyte text contains only ASCII and 8-bit raw bytes. In general, these conversions happen when inserting text into a buffer, or when putting text from several strings together in one string. You can also explicitly convert a string's contents to either representation.

Emacs chooses the representation for a string based on the text from which it is constructed. The general rule is to convert unibyte text to multibyte text when combining it with other multibyte text, because the multibyte representation is more general and can hold whatever characters the unibyte text has.

When inserting text into a buffer, Emacs converts the text to the buffer's representation, as specified by `enable-multibyte-characters` in that buffer. In particular, when you insert multibyte text into a unibyte buffer, Emacs converts the text to unibyte, even though this conversion cannot in general preserve all the characters that might be in the multibyte text. The other natural alternative, to convert the buffer contents to multibyte, is not acceptable because the buffer's representation is a choice made by the user that cannot be overridden automatically.

Converting unibyte text to multibyte text leaves ASCII characters unchanged, and converts bytes with codes 128 through 255 to the multibyte representation of raw eight-bit bytes.

Converting multibyte text to unibyte converts all ASCII and eight-bit characters to their single-byte form, but loses information for non-ASCII characters by discarding all but the low 8 bits of each character's codepoint. Converting unibyte text to multibyte and back to unibyte reproduces the original unibyte text.

The next two functions either return the argument *string*, or a newly created string with no text properties.

`string-to-multibyte` *string* [Function]
: This function returns a multibyte string containing the same sequence of characters as *string*. If *string* is a multibyte string, it is returned unchanged. The function assumes that *string* includes only ASCII characters and raw 8-bit bytes; the latter are converted to their multibyte representation corresponding to the codepoints `#x3FFF80` through `#x3FFFFF`, inclusive (see Section 33.1 [Text Representations], page 182).

`string-to-unibyte` *string* [Function]
: This function returns a unibyte string containing the same sequence of characters as *string*. It signals an error if *string* contains a non-ASCII character. If *string* is a unibyte string, it is returned unchanged. Use this function for *string* arguments that contain only ASCII and eight-bit characters.

`byte-to-string` *byte* [Function]
: This function returns a unibyte string containing a single byte of character data, *character*. It signals an error if *character* is not an integer between 0 and 255.

`multibyte-char-to-unibyte` *char* [Function]
: This converts the multibyte character *char* to a unibyte character, and returns that character. If *char* is neither ASCII nor eight-bit, the function returns -1.

`unibyte-char-to-multibyte` *char* [Function]
: This convert the unibyte character *char* to a multibyte character, assuming *char* is either ASCII or raw 8-bit byte.

33.3 Selecting a Representation

Sometimes it is useful to examine an existing buffer or string as multibyte when it was unibyte, or vice versa.

`set-buffer-multibyte` *multibyte* [Function]
: Set the representation type of the current buffer. If *multibyte* is non-`nil`, the buffer becomes multibyte. If *multibyte* is `nil`, the buffer becomes unibyte.

  This function leaves the buffer contents unchanged when viewed as a sequence of bytes. As a consequence, it can change the contents viewed as characters; for instance, a sequence of three bytes which is treated as one character in multibyte representation will count as three characters in unibyte representation. Eight-bit characters representing raw bytes are an exception. They are represented by one byte in a unibyte buffer,

but when the buffer is set to multibyte, they are converted to two-byte sequences, and vice versa.

This function sets `enable-multibyte-characters` to record which representation is in use. It also adjusts various data in the buffer (including overlays, text properties and markers) so that they cover the same text as they did before.

You cannot use `set-buffer-multibyte` on an indirect buffer, because indirect buffers always inherit the representation of the base buffer.

`string-as-unibyte` *string* [Function]

If *string* is already a unibyte string, this function returns *string* itself. Otherwise, it returns a new string with the same bytes as *string*, but treating each byte as a separate character (so that the value may have more characters than *string*); as an exception, each eight-bit character representing a raw byte is converted into a single byte. The newly-created string contains no text properties.

`string-as-multibyte` *string* [Function]

If *string* is a multibyte string, this function returns *string* itself. Otherwise, it returns a new string with the same bytes as *string*, but treating each multibyte sequence as one character. This means that the value may have fewer characters than *string* has. If a byte sequence in *string* is invalid as a multibyte representation of a single character, each byte in the sequence is treated as a raw 8-bit byte. The newly-created string contains no text properties.

33.4 Character Codes

The unibyte and multibyte text representations use different character codes. The valid character codes for unibyte representation range from 0 to `#xFF` (255)—the values that can fit in one byte. The valid character codes for multibyte representation range from 0 to `#x3FFFFF`. In this code space, values 0 through `#x7F` (127) are for ASCII characters, and values `#x80` (128) through `#x3FFF7F` (4194175) are for non-ASCII characters.

Emacs character codes are a superset of the Unicode standard. Values 0 through `#x10FFFF` (1114111) correspond to Unicode characters of the same codepoint; values `#x110000` (1114112) through `#x3FFF7F` (4194175) represent characters that are not unified with Unicode; and values `#x3FFF80` (4194176) through `#x3FFFFF` (4194303) represent eight-bit raw bytes.

`characterp` *charcode* [Function]

This returns `t` if *charcode* is a valid character, and `nil` otherwise.

```
(characterp 65)
     ⇒ t
(characterp 4194303)
     ⇒ t
(characterp 4194304)
     ⇒ nil
```

`max-char` [Function]

This function returns the largest value that a valid character codepoint can have.

```
(characterp (max-char))
     ⇒ t
(characterp (1+ (max-char)))
     ⇒ nil
```

`get-byte` **&optional** *pos string* [Function]

This function returns the byte at character position *pos* in the current buffer. If the current buffer is unibyte, this is literally the byte at that position. If the buffer is multibyte, byte values of ASCII characters are the same as character codepoints, whereas eight-bit raw bytes are converted to their 8-bit codes. The function signals an error if the character at *pos* is non-ASCII.

The optional argument *string* means to get a byte value from that string instead of the current buffer.

33.5 Character Properties

A *character property* is a named attribute of a character that specifies how the character behaves and how it should be handled during text processing and display. Thus, character properties are an important part of specifying the character's semantics.

On the whole, Emacs follows the Unicode Standard in its implementation of character properties. In particular, Emacs supports the Unicode Character Property Model, and the Emacs character property database is derived from the Unicode Character Database (UCD). See the Character Properties chapter of the Unicode Standard, for a detailed description of Unicode character properties and their meaning. This section assumes you are already familiar with that chapter of the Unicode Standard, and want to apply that knowledge to Emacs Lisp programs.

In Emacs, each property has a name, which is a symbol, and a set of possible values, whose types depend on the property; if a character does not have a certain property, the value is `nil`. As a general rule, the names of character properties in Emacs are produced from the corresponding Unicode properties by downcasing them and replacing each '`_`' character with a dash '`-`'. For example, `Canonical_Combining_Class` becomes `canonical-combining-class`. However, sometimes we shorten the names to make their use easier.

Some codepoints are left *unassigned* by the UCD—they don't correspond to any character. The Unicode Standard defines default values of properties for such codepoints; they are mentioned below for each property.

Here is the full list of value types for all the character properties that Emacs knows about:

`name`
: Corresponds to the `Name` Unicode property. The value is a string consisting of upper-case Latin letters A to Z, digits, spaces, and hyphen '`-`' characters. For unassigned codepoints, the value is an empty string.

`general-category`
: Corresponds to the `General_Category` Unicode property. The value is a symbol whose name is a 2-letter abbreviation of the character's classification. For unassigned codepoints, the value is `Cn`.

`canonical-combining-class`
: Corresponds to the `Canonical_Combining_Class` Unicode property. The value is an integer number. For unassigned codepoints, the value is zero.

`bidi-class`
: Corresponds to the Unicode `Bidi_Class` property. The value is a symbol whose name is the Unicode *directional type* of the character. Emacs uses this property when it reorders bidirectional text for display (see Section 38.23 [Bidirectional Display], page 382). For unassigned codepoints, the value depends on the code blocks to which the codepoint belongs: most unassigned codepoints get the value of `L` (strong L), but some get values of `AL` (Arabic letter) or `R` (strong R).

`decomposition`
: Corresponds to the Unicode properties `Decomposition_Type` and `Decomposition_Value`. The value is a list, whose first element may be a symbol representing a compatibility formatting tag, such as `small`[2]; the other elements are characters that give the compatibility decomposition sequence of this character. For unassigned codepoints, the value is the character itself.

`decimal-digit-value`
: Corresponds to the Unicode `Numeric_Value` property for characters whose `Numeric_Type` is '`Digit`'. The value is an integer number. For unassigned codepoints, the value is `nil`, which means NaN, or "not-a-number".

`digit-value`
: Corresponds to the Unicode `Numeric_Value` property for characters whose `Numeric_Type` is '`Decimal`'. The value is an integer number. Examples of such characters include compatibility subscript and superscript digits, for which the value is the corresponding number. For unassigned codepoints, the value is `nil`, which means NaN.

`numeric-value`
: Corresponds to the Unicode `Numeric_Value` property for characters whose `Numeric_Type` is '`Numeric`'. The value of this property is an integer or a floating-point number. Examples of characters that have this property include fractions, subscripts, superscripts, Roman numerals, currency numerators, and encircled numbers. For example, the value of this property for the character `U+2155` (VULGAR FRACTION ONE FIFTH) is `0.2`. For unassigned codepoints, the value is `nil`, which means NaN.

`mirrored`
: Corresponds to the Unicode `Bidi_Mirrored` property. The value of this property is a symbol, either `Y` or `N`. For unassigned codepoints, the value is `N`.

`mirroring`
: Corresponds to the Unicode `Bidi_Mirroring_Glyph` property. The value of this property is a character whose glyph represents the mirror image of the character's glyph, or `nil` if there's no defined mirroring glyph. All the characters whose `mirrored` property is `N` have `nil` as their `mirroring` property; however,

[2] The Unicode specification writes these tag names inside '`<..>`' brackets, but the tag names in Emacs do not include the brackets; e.g. Unicode specifies '`<small>`' where Emacs uses '`small`'.

some characters whose `mirrored` property is `Y` also have `nil` for `mirroring`, because no appropriate characters exist with mirrored glyphs. Emacs uses this property to display mirror images of characters when appropriate (see Section 38.23 [Bidirectional Display], page 382). For unassigned codepoints, the value is `nil`.

`old-name` Corresponds to the Unicode `Unicode_1_Name` property. The value is a string. For unassigned codepoints, the value is an empty string.

`iso-10646-comment`
Corresponds to the Unicode `ISO_Comment` property. The value is a string. For unassigned codepoints, the value is an empty string.

`uppercase`
Corresponds to the Unicode `Simple_Uppercase_Mapping` property. The value of this property is a single character. For unassigned codepoints, the value is `nil`, which means the character itself.

`lowercase`
Corresponds to the Unicode `Simple_Lowercase_Mapping` property. The value of this property is a single character. For unassigned codepoints, the value is `nil`, which means the character itself.

`titlecase`
Corresponds to the Unicode `Simple_Titlecase_Mapping` property. *Title case* is a special form of a character used when the first character of a word needs to be capitalized. The value of this property is a single character. For unassigned codepoints, the value is `nil`, which means the character itself.

`get-char-code-property` *char propname* [Function]
This function returns the value of *char*'s *propname* property.

```
(get-char-code-property ?  'general-category)
     ⇒ Zs
(get-char-code-property ?1  'general-category)
     ⇒ Nd
;; subscript 4
(get-char-code-property ?\u2084 'digit-value)
     ⇒ 4
;; one fifth
(get-char-code-property ?\u2155 'numeric-value)
     ⇒ 0.2
;; Roman IV
(get-char-code-property ?\u2163 'numeric-value)
     ⇒ 4
```

`char-code-property-description` *prop value* [Function]
This function returns the description string of property *prop*'s *value*, or `nil` if *value* has no description.

```
(char-code-property-description 'general-category 'Zs)
     ⇒ "Separator, Space"
```

```
(char-code-property-description 'general-category 'Nd)
     ⇒ "Number, Decimal Digit"
(char-code-property-description 'numeric-value '1/5)
     ⇒ nil
```

`put-char-code-property` *char propname value* [Function]
This function stores *value* as the value of the property *propname* for the character *char*.

`unicode-category-table` [Variable]
The value of this variable is a char-table (see Section 6.6 [Char-Tables], page 92, vol. 1) that specifies, for each character, its Unicode `General_Category` property as a symbol.

`char-script-table` [Variable]
The value of this variable is a char-table that specifies, for each character, a symbol whose name is the script to which the character belongs, according to the Unicode Standard classification of the Unicode code space into script-specific blocks. This char-table has a single extra slot whose value is the list of all script symbols.

`char-width-table` [Variable]
The value of this variable is a char-table that specifies the width of each character in columns that it will occupy on the screen.

`printable-chars` [Variable]
The value of this variable is a char-table that specifies, for each character, whether it is printable or not. That is, if evaluating `(aref printable-chars char)` results in `t`, the character is printable, and if it results in `nil`, it is not.

33.6 Character Sets

An Emacs *character set*, or *charset*, is a set of characters in which each character is assigned a numeric code point. (The Unicode Standard calls this a *coded character set*.) Each Emacs charset has a name which is a symbol. A single character can belong to any number of different character sets, but it will generally have a different code point in each charset. Examples of character sets include `ascii`, `iso-8859-1`, `greek-iso8859-7`, and `windows-1255`. The code point assigned to a character in a charset is usually different from its code point used in Emacs buffers and strings.

Emacs defines several special character sets. The character set `unicode` includes all the characters whose Emacs code points are in the range `0..#x10FFFF`. The character set `emacs` includes all ASCII and non-ASCII characters. Finally, the `eight-bit` charset includes the 8-bit raw bytes; Emacs uses it to represent raw bytes encountered in text.

`charsetp` *object* [Function]
Returns `t` if *object* is a symbol that names a character set, `nil` otherwise.

`charset-list` [Variable]
The value is a list of all defined character set names.

`charset-priority-list` **&optional** *highestp* [Function]
: This functions returns a list of all defined character sets ordered by their priority. If *highestp* is non-`nil`, the function returns a single character set of the highest priority.

`set-charset-priority` **&rest** *charsets* [Function]
: This function makes *charsets* the highest priority character sets.

`char-charset` *character* **&optional** *restriction* [Function]
: This function returns the name of the character set of highest priority that *character* belongs to. ASCII characters are an exception: for them, this function always returns `ascii`.

  If *restriction* is non-`nil`, it should be a list of charsets to search. Alternatively, it can be a coding system, in which case the returned charset must be supported by that coding system (see Section 33.9 [Coding Systems], page 193).

`charset-plist` *charset* [Function]
: This function returns the property list of the character set *charset*. Although *charset* is a symbol, this is not the same as the property list of that symbol. Charset properties include important information about the charset, such as its documentation string, short name, etc.

`put-charset-property` *charset propname value* [Function]
: This function sets the *propname* property of *charset* to the given *value*.

`get-charset-property` *charset propname* [Function]
: This function returns the value of *charset*s property *propname*.

`list-charset-chars` *charset* [Command]
: This command displays a list of characters in the character set *charset*.

Emacs can convert between its internal representation of a character and the character's codepoint in a specific charset. The following two functions support these conversions.

`decode-char` *charset code-point* [Function]
: This function decodes a character that is assigned a *code-point* in *charset*, to the corresponding Emacs character, and returns it. If *charset* doesn't contain a character of that code point, the value is `nil`. If *code-point* doesn't fit in a Lisp integer (see Section 3.1 [Integer Basics], page 33, vol. 1), it can be specified as a cons cell `(high . low)`, where *low* are the lower 16 bits of the value and *high* are the high 16 bits.

`encode-char` *char charset* [Function]
: This function returns the code point assigned to the character *char* in *charset*. If the result does not fit in a Lisp integer, it is returned as a cons cell `(high . low)` that fits the second argument of `decode-char` above. If *charset* doesn't have a codepoint for *char*, the value is `nil`.

The following function comes in handy for applying a certain function to all or part of the characters in a charset:

`map-charset-chars` *function charset* **&optional** *arg from-code to-code* [Function]
Call *function* for characters in *charset*. *function* is called with two arguments. The first one is a cons cell `(from . to)`, where *from* and *to* indicate a range of characters contained in charset. The second argument passed to *function* is *arg*.

By default, the range of codepoints passed to *function* includes all the characters in *charset*, but optional arguments *from-code* and *to-code* limit that to the range of characters between these two codepoints of *charset*. If either of them is `nil`, it defaults to the first or last codepoint of *charset*, respectively.

33.7 Scanning for Character Sets

Sometimes it is useful to find out which character set a particular character belongs to. One use for this is in determining which coding systems (see Section 33.9 [Coding Systems], page 193) are capable of representing all of the text in question; another is to determine the font(s) for displaying that text.

`charset-after` **&optional** *pos* [Function]
This function returns the charset of highest priority containing the character at position *pos* in the current buffer. If *pos* is omitted or `nil`, it defaults to the current value of point. If *pos* is out of range, the value is `nil`.

`find-charset-region` *beg end* **&optional** *translation* [Function]
This function returns a list of the character sets of highest priority that contain characters in the current buffer between positions *beg* and *end*.

The optional argument *translation* specifies a translation table to use for scanning the text (see Section 33.8 [Translation of Characters], page 191). If it is non-`nil`, then each character in the region is translated through this table, and the value returned describes the translated characters instead of the characters actually in the buffer.

`find-charset-string` *string* **&optional** *translation* [Function]
This function returns a list of character sets of highest priority that contain characters in *string*. It is just like `find-charset-region`, except that it applies to the contents of *string* instead of part of the current buffer.

33.8 Translation of Characters

A *translation table* is a char-table (see Section 6.6 [Char-Tables], page 92, vol. 1) that specifies a mapping of characters into characters. These tables are used in encoding and decoding, and for other purposes. Some coding systems specify their own particular translation tables; there are also default translation tables which apply to all other coding systems.

A translation table has two extra slots. The first is either `nil` or a translation table that performs the reverse translation; the second is the maximum number of characters to look up for translating sequences of characters (see the description of `make-translation-table-from-alist` below).

`make-translation-table` **&rest** *translations* [Function]
This function returns a translation table based on the argument *translations*. Each element of *translations* should be a list of elements of the form `(from . to)`; this says to translate the character *from* into *to*.

The arguments and the forms in each argument are processed in order, and if a previous form already translates *to* to some other character, say *to-alt*, *from* is also translated to *to-alt*.

During decoding, the translation table's translations are applied to the characters that result from ordinary decoding. If a coding system has the property `:decode-translation-table`, that specifies the translation table to use, or a list of translation tables to apply in sequence. (This is a property of the coding system, as returned by `coding-system-get`, not a property of the symbol that is the coding system's name. See Section 33.9.1 [Basic Concepts of Coding Systems], page 193.) Finally, if `standard-translation-table-for-decode` is non-`nil`, the resulting characters are translated by that table.

During encoding, the translation table's translations are applied to the characters in the buffer, and the result of translation is actually encoded. If a coding system has property `:encode-translation-table`, that specifies the translation table to use, or a list of translation tables to apply in sequence. In addition, if the variable `standard-translation-table-for-encode` is non-`nil`, it specifies the translation table to use for translating the result.

`standard-translation-table-for-decode` [Variable]
This is the default translation table for decoding. If a coding systems specifies its own translation tables, the table that is the value of this variable, if non-`nil`, is applied after them.

`standard-translation-table-for-encode` [Variable]
This is the default translation table for encoding. If a coding systems specifies its own translation tables, the table that is the value of this variable, if non-`nil`, is applied after them.

`translation-table-for-input` [Variable]
Self-inserting characters are translated through this translation table before they are inserted. Search commands also translate their input through this table, so they can compare more reliably with what's in the buffer.

This variable automatically becomes buffer-local when set.

`make-translation-table-from-vector` *vec* [Function]
This function returns a translation table made from *vec* that is an array of 256 elements to map bytes (values 0 through #xFF) to characters. Elements may be `nil` for untranslated bytes. The returned table has a translation table for reverse mapping in the first extra slot, and the value `1` in the second extra slot.

This function provides an easy way to make a private coding system that maps each byte to a specific character. You can specify the returned table and the reverse translation table using the properties `:decode-translation-table` and `:encode-translation-table` respectively in the *props* argument to `define-coding-system`.

`make-translation-table-from-alist` *alist* [Function]
This function is similar to `make-translation-table` but returns a complex translation table rather than a simple one-to-one mapping. Each element of *alist* is of the form `(from . to)`, where *from* and *to* are either characters or vectors specifying

a sequence of characters. If *from* is a character, that character is translated to *to* (i.e. to a character or a character sequence). If *from* is a vector of characters, that sequence is translated to *to*. The returned table has a translation table for reverse mapping in the first extra slot, and the maximum length of all the *from* character sequences in the second extra slot.

33.9 Coding Systems

When Emacs reads or writes a file, and when Emacs sends text to a subprocess or receives text from a subprocess, it normally performs character code conversion and end-of-line conversion as specified by a particular *coding system*.

How to define a coding system is an arcane matter, and is not documented here.

33.9.1 Basic Concepts of Coding Systems

Character code conversion involves conversion between the internal representation of characters used inside Emacs and some other encoding. Emacs supports many different encodings, in that it can convert to and from them. For example, it can convert text to or from encodings such as Latin 1, Latin 2, Latin 3, Latin 4, Latin 5, and several variants of ISO 2022. In some cases, Emacs supports several alternative encodings for the same characters; for example, there are three coding systems for the Cyrillic (Russian) alphabet: ISO, Alternativnyj, and KOI8.

Every coding system specifies a particular set of character code conversions, but the coding system `undecided` is special: it leaves the choice unspecified, to be chosen heuristically for each file, based on the file's data.

In general, a coding system doesn't guarantee roundtrip identity: decoding a byte sequence using coding system, then encoding the resulting text in the same coding system, can produce a different byte sequence. But some coding systems do guarantee that the byte sequence will be the same as what you originally decoded. Here are a few examples:

iso-8859-1, utf-8, big5, shift_jis, euc-jp

Encoding buffer text and then decoding the result can also fail to reproduce the original text. For instance, if you encode a character with a coding system which does not support that character, the result is unpredictable, and thus decoding it using the same coding system may produce a different text. Currently, Emacs can't report errors that result from encoding unsupported characters.

End of line conversion handles three different conventions used on various systems for representing end of line in files. The Unix convention, used on GNU and Unix systems, is to use the linefeed character (also called newline). The DOS convention, used on MS-Windows and MS-DOS systems, is to use a carriage-return and a linefeed at the end of a line. The Mac convention is to use just carriage-return.

Base coding systems such as `latin-1` leave the end-of-line conversion unspecified, to be chosen based on the data. *Variant coding systems* such as `latin-1-unix`, `latin-1-dos` and `latin-1-mac` specify the end-of-line conversion explicitly as well. Most base coding systems have three corresponding variants whose names are formed by adding '`-unix`', '`-dos`' and '`-mac`'.

The coding system `raw-text` is special in that it prevents character code conversion, and causes the buffer visited with this coding system to be a unibyte buffer. For historical

reasons, you can save both unibyte and multibyte text with this coding system. When you use `raw-text` to encode multibyte text, it does perform one character code conversion: it converts eight-bit characters to their single-byte external representation. `raw-text` does not specify the end-of-line conversion, allowing that to be determined as usual by the data, and has the usual three variants which specify the end-of-line conversion.

`no-conversion` (and its alias `binary`) is equivalent to `raw-text-unix`: it specifies no conversion of either character codes or end-of-line.

The coding system `utf-8-emacs` specifies that the data is represented in the internal Emacs encoding (see Section 33.1 [Text Representations], page 182). This is like `raw-text` in that no code conversion happens, but different in that the result is multibyte data. The name `emacs-internal` is an alias for `utf-8-emacs`.

`coding-system-get` *coding-system property* [Function]

This function returns the specified property of the coding system *coding-system*. Most coding system properties exist for internal purposes, but one that you might find useful is `:mime-charset`. That property's value is the name used in MIME for the character coding which this coding system can read and write. Examples:

```
(coding-system-get 'iso-latin-1 :mime-charset)
     ⇒ iso-8859-1
(coding-system-get 'iso-2022-cn :mime-charset)
     ⇒ iso-2022-cn
(coding-system-get 'cyrillic-koi8 :mime-charset)
     ⇒ koi8-r
```

The value of the `:mime-charset` property is also defined as an alias for the coding system.

`coding-system-aliases` *coding-system* [Function]

This function returns the list of aliases of *coding-system*.

33.9.2 Encoding and I/O

The principal purpose of coding systems is for use in reading and writing files. The function `insert-file-contents` uses a coding system to decode the file data, and `write-region` uses one to encode the buffer contents.

You can specify the coding system to use either explicitly (see Section 33.9.6 [Specifying Coding Systems], page 202), or implicitly using a default mechanism (see Section 33.9.5 [Default Coding Systems], page 199). But these methods may not completely specify what to do. For example, they may choose a coding system such as `undefined` which leaves the character code conversion to be determined from the data. In these cases, the I/O operation finishes the job of choosing a coding system. Very often you will want to find out afterwards which coding system was chosen.

`buffer-file-coding-system` [Variable]

This buffer-local variable records the coding system used for saving the buffer and for writing part of the buffer with `write-region`. If the text to be written cannot be safely encoded using the coding system specified by this variable, these operations select an alternative encoding by calling the function `select-safe-coding-system` (see Section 33.9.4 [User-Chosen Coding Systems], page 198). If selecting a different

encoding requires to ask the user to specify a coding system, `buffer-file-coding-system` is updated to the newly selected coding system.

`buffer-file-coding-system` does *not* affect sending text to a subprocess.

`save-buffer-coding-system` [Variable]
This variable specifies the coding system for saving the buffer (by overriding `buffer-file-coding-system`). Note that it is not used for `write-region`.

When a command to save the buffer starts out to use `buffer-file-coding-system` (or `save-buffer-coding-system`), and that coding system cannot handle the actual text in the buffer, the command asks the user to choose another coding system (by calling `select-safe-coding-system`). After that happens, the command also updates `buffer-file-coding-system` to represent the coding system that the user specified.

`last-coding-system-used` [Variable]
I/O operations for files and subprocesses set this variable to the coding system name that was used. The explicit encoding and decoding functions (see Section 33.9.7 [Explicit Encoding], page 203) set it too.

Warning: Since receiving subprocess output sets this variable, it can change whenever Emacs waits; therefore, you should copy the value shortly after the function call that stores the value you are interested in.

The variable `selection-coding-system` specifies how to encode selections for the window system. See Section 29.18 [Window System Selections], page 91.

`file-name-coding-system` [Variable]
The variable `file-name-coding-system` specifies the coding system to use for encoding file names. Emacs encodes file names using that coding system for all file operations. If `file-name-coding-system` is `nil`, Emacs uses a default coding system determined by the selected language environment. In the default language environment, any non-ASCII characters in file names are not encoded specially; they appear in the file system using the internal Emacs representation.

Warning: if you change `file-name-coding-system` (or the language environment) in the middle of an Emacs session, problems can result if you have already visited files whose names were encoded using the earlier coding system and are handled differently under the new coding system. If you try to save one of these buffers under the visited file name, saving may use the wrong file name, or it may get an error. If such a problem happens, use `C-x C-w` to specify a new file name for that buffer.

33.9.3 Coding Systems in Lisp

Here are the Lisp facilities for working with coding systems:

`coding-system-list` **&optional** *base-only* [Function]
This function returns a list of all coding system names (symbols). If *base-only* is non-`nil`, the value includes only the base coding systems. Otherwise, it includes alias and variant coding systems as well.

`coding-system-p` *object* [Function]
This function returns `t` if *object* is a coding system name or `nil`.

`check-coding-system` *coding-system* [Function]
This function checks the validity of *coding-system*. If that is valid, it returns *coding-system*. If *coding-system* is `nil`, the function return `nil`. For any other values, it signals an error whose `error-symbol` is `coding-system-error` (see Section 10.5.3.1 [Signaling Errors], page 128, vol. 1).

`coding-system-eol-type` *coding-system* [Function]
This function returns the type of end-of-line (a.k.a. *eol*) conversion used by *coding-system*. If *coding-system* specifies a certain eol conversion, the return value is an integer 0, 1, or 2, standing for `unix`, `dos`, and `mac`, respectively. If *coding-system* doesn't specify eol conversion explicitly, the return value is a vector of coding systems, each one with one of the possible eol conversion types, like this:

```
(coding-system-eol-type 'latin-1)
     ⇒ [latin-1-unix latin-1-dos latin-1-mac]
```

If this function returns a vector, Emacs will decide, as part of the text encoding or decoding process, what eol conversion to use. For decoding, the end-of-line format of the text is auto-detected, and the eol conversion is set to match it (e.g., DOS-style CRLF format will imply `dos` eol conversion). For encoding, the eol conversion is taken from the appropriate default coding system (e.g., default value of `buffer-file-coding-system` for `buffer-file-coding-system`), or from the default eol conversion appropriate for the underlying platform.

`coding-system-change-eol-conversion` *coding-system eol-type* [Function]
This function returns a coding system which is like *coding-system* except for its eol conversion, which is specified by `eol-type`. *eol-type* should be `unix`, `dos`, `mac`, or `nil`. If it is `nil`, the returned coding system determines the end-of-line conversion from the data.

eol-type may also be 0, 1 or 2, standing for `unix`, `dos` and `mac`, respectively.

`coding-system-change-text-conversion` *eol-coding text-coding* [Function]
This function returns a coding system which uses the end-of-line conversion of *eol-coding*, and the text conversion of *text-coding*. If *text-coding* is `nil`, it returns `undecided`, or one of its variants according to *eol-coding*.

`find-coding-systems-region` *from to* [Function]
This function returns a list of coding systems that could be used to encode a text between *from* and *to*. All coding systems in the list can safely encode any multibyte characters in that portion of the text.

If the text contains no multibyte characters, the function returns the list `(undecided)`.

`find-coding-systems-string` *string* [Function]
This function returns a list of coding systems that could be used to encode the text of *string*. All coding systems in the list can safely encode any multibyte characters in *string*. If the text contains no multibyte characters, this returns the list `(undecided)`.

`find-coding-systems-for-charsets` *charsets* [Function]
: This function returns a list of coding systems that could be used to encode all the character sets in the list *charsets*.

`check-coding-systems-region` *start end coding-system-list* [Function]
: This function checks whether coding systems in the list `coding-system-list` can encode all the characters in the region between *start* and *end*. If all of the coding systems in the list can encode the specified text, the function returns `nil`. If some coding systems cannot encode some of the characters, the value is an alist, each element of which has the form `(`*`coding-system1 pos1 pos2 ...`*`)`, meaning that *coding-system1* cannot encode characters at buffer positions *pos1*, *pos2*, ...

  start may be a string, in which case *end* is ignored and the returned value references string indices instead of buffer positions.

`detect-coding-region` *start end* **&optional** *highest* [Function]
: This function chooses a plausible coding system for decoding the text from *start* to *end*. This text should be a byte sequence, i.e. unibyte text or multibyte text with only ASCII and eight-bit characters (see Section 33.9.7 [Explicit Encoding], page 203).

  Normally this function returns a list of coding systems that could handle decoding the text that was scanned. They are listed in order of decreasing priority. But if *highest* is non-`nil`, then the return value is just one coding system, the one that is highest in priority.

  If the region contains only ASCII characters except for such ISO-2022 control characters ISO-2022 as `ESC`, the value is `undecided` or `(undecided)`, or a variant specifying end-of-line conversion, if that can be deduced from the text.

  If the region contains null bytes, the value is `no-conversion`, even if the region contains text encoded in some coding system.

`detect-coding-string` *string* **&optional** *highest* [Function]
: This function is like `detect-coding-region` except that it operates on the contents of *string* instead of bytes in the buffer.

`inhibit-null-byte-detection` [Variable]
: If this variable has a non-`nil` value, null bytes are ignored when detecting the encoding of a region or a string. This allows to correctly detect the encoding of text that contains null bytes, such as Info files with Index nodes.

`inhibit-iso-escape-detection` [Variable]
: If this variable has a non-`nil` value, ISO-2022 escape sequences are ignored when detecting the encoding of a region or a string. The result is that no text is ever detected as encoded in some ISO-2022 encoding, and all escape sequences become visible in a buffer. **Warning:** *Use this variable with extreme caution, because many files in the Emacs distribution use ISO-2022 encoding.*

`coding-system-charset-list` *coding-system* [Function]
: This function returns the list of character sets (see Section 33.6 [Character Sets], page 189) supported by *coding-system*. Some coding systems that support too many character sets to list them all yield special values:

- If *coding-system* supports all the ISO-2022 charsets, the value is `iso-2022`.
- If *coding-system* supports all Emacs characters, the value is `(emacs)`.
- If *coding-system* supports all emacs-mule characters, the value is `emacs-mule`.
- If *coding-system* supports all Unicode characters, the value is `(unicode)`.

See [Process Information], page 269, in particular the description of the functions `process-coding-system` and `set-process-coding-system`, for how to examine or set the coding systems used for I/O to a subprocess.

33.9.4 User-Chosen Coding Systems

`select-safe-coding-system` *from to* **&optional** *default-coding-system accept-default-p file* [Function]

This function selects a coding system for encoding specified text, asking the user to choose if necessary. Normally the specified text is the text in the current buffer between *from* and *to*. If *from* is a string, the string specifies the text to encode, and *to* is ignored.

If the specified text includes raw bytes (see Section 33.1 [Text Representations], page 182), `select-safe-coding-system` suggests `raw-text` for its encoding.

If *default-coding-system* is non-`nil`, that is the first coding system to try; if that can handle the text, `select-safe-coding-system` returns that coding system. It can also be a list of coding systems; then the function tries each of them one by one. After trying all of them, it next tries the current buffer's value of `buffer-file-coding-system` (if it is not `undecided`), then the default value of `buffer-file-coding-system` and finally the user's most preferred coding system, which the user can set using the command `prefer-coding-system` (see Section "Recognizing Coding Systems" in *The GNU Emacs Manual*).

If one of those coding systems can safely encode all the specified text, `select-safe-coding-system` chooses it and returns it. Otherwise, it asks the user to choose from a list of coding systems which can encode all the text, and returns the user's choice.

default-coding-system can also be a list whose first element is t and whose other elements are coding systems. Then, if no coding system in the list can handle the text, `select-safe-coding-system` queries the user immediately, without trying any of the three alternatives described above.

The optional argument *accept-default-p*, if non-`nil`, should be a function to determine whether a coding system selected without user interaction is acceptable. `select-safe-coding-system` calls this function with one argument, the base coding system of the selected coding system. If *accept-default-p* returns `nil`, `select-safe-coding-system` rejects the silently selected coding system, and asks the user to select a coding system from a list of possible candidates.

If the variable `select-safe-coding-system-accept-default-p` is non-`nil`, it should be a function taking a single argument. It is used in place of *accept-default-p*, overriding any value supplied for this argument.

As a final step, before returning the chosen coding system, `select-safe-coding-system` checks whether that coding system is consistent with what would be selected

if the contents of the region were read from a file. (If not, this could lead to data corruption in a file subsequently re-visited and edited.) Normally, `select-safe-coding-system` uses `buffer-file-name` as the file for this purpose, but if *file* is non-`nil`, it uses that file instead (this can be relevant for `write-region` and similar functions). If it detects an apparent inconsistency, `select-safe-coding-system` queries the user before selecting the coding system.

Here are two functions you can use to let the user specify a coding system, with completion. See Section 20.6 [Completion], page 291, vol. 1.

`read-coding-system` *prompt* **&optional** *default* [Function]
This function reads a coding system using the minibuffer, prompting with string *prompt*, and returns the coding system name as a symbol. If the user enters null input, *default* specifies which coding system to return. It should be a symbol or a string.

`read-non-nil-coding-system` *prompt* [Function]
This function reads a coding system using the minibuffer, prompting with string *prompt*, and returns the coding system name as a symbol. If the user tries to enter null input, it asks the user to try again. See Section 33.9 [Coding Systems], page 193.

33.9.5 Default Coding Systems

This section describes variables that specify the default coding system for certain files or when running certain subprograms, and the function that I/O operations use to access them.

The idea of these variables is that you set them once and for all to the defaults you want, and then do not change them again. To specify a particular coding system for a particular operation in a Lisp program, don't change these variables; instead, override them using `coding-system-for-read` and `coding-system-for-write` (see Section 33.9.6 [Specifying Coding Systems], page 202).

`auto-coding-regexp-alist` [User Option]
This variable is an alist of text patterns and corresponding coding systems. Each element has the form `(`*`regexp`* `.` *`coding-system`*`)`; a file whose first few kilobytes match *regexp* is decoded with *coding-system* when its contents are read into a buffer. The settings in this alist take priority over `coding:` tags in the files and the contents of `file-coding-system-alist` (see below). The default value is set so that Emacs automatically recognizes mail files in Babyl format and reads them with no code conversions.

`file-coding-system-alist` [User Option]
This variable is an alist that specifies the coding systems to use for reading and writing particular files. Each element has the form `(`*`pattern`* `.` *`coding`*`)`, where *pattern* is a regular expression that matches certain file names. The element applies to file names that match *pattern*.

The CDR of the element, *coding*, should be either a coding system, a cons cell containing two coding systems, or a function name (a symbol with a function definition). If *coding* is a coding system, that coding system is used for both reading the file and

writing it. If *coding* is a cons cell containing two coding systems, its CAR specifies the coding system for decoding, and its CDR specifies the coding system for encoding.

If *coding* is a function name, the function should take one argument, a list of all arguments passed to `find-operation-coding-system`. It must return a coding system or a cons cell containing two coding systems. This value has the same meaning as described above.

If *coding* (or what returned by the above function) is `undecided`, the normal code-detection is performed.

`auto-coding-alist` [User Option]

This variable is an alist that specifies the coding systems to use for reading and writing particular files. Its form is like that of `file-coding-system-alist`, but, unlike the latter, this variable takes priority over any `coding:` tags in the file.

`process-coding-system-alist` [Variable]

This variable is an alist specifying which coding systems to use for a subprocess, depending on which program is running in the subprocess. It works like `file-coding-system-alist`, except that *pattern* is matched against the program name used to start the subprocess. The coding system or systems specified in this alist are used to initialize the coding systems used for I/O to the subprocess, but you can specify other coding systems later using `set-process-coding-system`.

Warning: Coding systems such as `undecided`, which determine the coding system from the data, do not work entirely reliably with asynchronous subprocess output. This is because Emacs handles asynchronous subprocess output in batches, as it arrives. If the coding system leaves the character code conversion unspecified, or leaves the end-of-line conversion unspecified, Emacs must try to detect the proper conversion from one batch at a time, and this does not always work.

Therefore, with an asynchronous subprocess, if at all possible, use a coding system which determines both the character code conversion and the end of line conversion—that is, one like `latin-1-unix`, rather than `undecided` or `latin-1`.

`network-coding-system-alist` [Variable]

This variable is an alist that specifies the coding system to use for network streams. It works much like `file-coding-system-alist`, with the difference that the *pattern* in an element may be either a port number or a regular expression. If it is a regular expression, it is matched against the network service name used to open the network stream.

`default-process-coding-system` [Variable]

This variable specifies the coding systems to use for subprocess (and network stream) input and output, when nothing else specifies what to do.

The value should be a cons cell of the form `(`*`input-coding`*` . `*`output-coding`*`)`. Here *input-coding* applies to input from the subprocess, and *output-coding* applies to output to it.

`auto-coding-functions` [User Option]

This variable holds a list of functions that try to determine a coding system for a file based on its undecoded contents.

Each function in this list should be written to look at text in the current buffer, but should not modify it in any way. The buffer will contain undecoded text of parts of the file. Each function should take one argument, *size*, which tells it how many characters to look at, starting from point. If the function succeeds in determining a coding system for the file, it should return that coding system. Otherwise, it should return `nil`.

If a file has a '`coding:`' tag, that takes precedence, so these functions won't be called.

`find-auto-coding` *filename size* [Function]

This function tries to determine a suitable coding system for *filename*. It examines the buffer visiting the named file, using the variables documented above in sequence, until it finds a match for one of the rules specified by these variables. It then returns a cons cell of the form `(`*`coding`*` . `*`source`*`)`, where *coding* is the coding system to use and *source* is a symbol, one of `auto-coding-alist`, `auto-coding-regexp-alist`, `:coding`, or `auto-coding-functions`, indicating which one supplied the matching rule. The value `:coding` means the coding system was specified by the `coding:` tag in the file (see Section "coding tag" in *The GNU Emacs Manual*). The order of looking for a matching rule is `auto-coding-alist` first, then `auto-coding-regexp-alist`, then the `coding:` tag, and lastly `auto-coding-functions`. If no matching rule was found, the function returns `nil`.

The second argument *size* is the size of text, in characters, following point. The function examines text only within *size* characters after point. Normally, the buffer should be positioned at the beginning when this function is called, because one of the places for the `coding:` tag is the first one or two lines of the file; in that case, *size* should be the size of the buffer.

`set-auto-coding` *filename size* [Function]

This function returns a suitable coding system for file *filename*. It uses `find-auto-coding` to find the coding system. If no coding system could be determined, the function returns `nil`. The meaning of the argument *size* is like in `find-auto-coding`.

`find-operation-coding-system` *operation* **&rest** *arguments* [Function]

This function returns the coding system to use (by default) for performing *operation* with *arguments*. The value has this form:

```
(decoding-system . encoding-system)
```

The first element, *decoding-system*, is the coding system to use for decoding (in case *operation* does decoding), and *encoding-system* is the coding system for encoding (in case *operation* does encoding).

The argument *operation* is a symbol; it should be one of `write-region`, `start-process`, `call-process`, `call-process-region`, `insert-file-contents`, or `open-network-stream`. These are the names of the Emacs I/O primitives that can do character code and eol conversion.

The remaining arguments should be the same arguments that might be given to the corresponding I/O primitive. Depending on the primitive, one of those arguments is selected as the *target*. For example, if *operation* does file I/O, whichever argument specifies the file name is the target. For subprocess primitives, the process name is the target. For `open-network-stream`, the target is the service name or port number.

Depending on *operation*, this function looks up the target in `file-coding-system-alist`, `process-coding-system-alist`, or `network-coding-system-alist`. If the target is found in the alist, `find-operation-coding-system` returns its association in the alist; otherwise it returns `nil`.

If *operation* is `insert-file-contents`, the argument corresponding to the target may be a cons cell of the form (*filename* . *buffer*)). In that case, *filename* is a file name to look up in `file-coding-system-alist`, and *buffer* is a buffer that contains the file's contents (not yet decoded). If `file-coding-system-alist` specifies a function to call for this file, and that function needs to examine the file's contents (as it usually does), it should examine the contents of *buffer* instead of reading the file.

33.9.6 Specifying a Coding System for One Operation

You can specify the coding system for a specific operation by binding the variables `coding-system-for-read` and/or `coding-system-for-write`.

`coding-system-for-read` [Variable]

If this variable is non-`nil`, it specifies the coding system to use for reading a file, or for input from a synchronous subprocess.

It also applies to any asynchronous subprocess or network stream, but in a different way: the value of `coding-system-for-read` when you start the subprocess or open the network stream specifies the input decoding method for that subprocess or network stream. It remains in use for that subprocess or network stream unless and until overridden.

The right way to use this variable is to bind it with `let` for a specific I/O operation. Its global value is normally `nil`, and you should not globally set it to any other value. Here is an example of the right way to use the variable:

```
;; Read the file with no character code conversion.
;; Assume crlf represents end-of-line.
(let ((coding-system-for-read 'emacs-mule-dos))
  (insert-file-contents filename))
```

When its value is non-`nil`, this variable takes precedence over all other methods of specifying a coding system to use for input, including `file-coding-system-alist`, `process-coding-system-alist` and `network-coding-system-alist`.

`coding-system-for-write` [Variable]

This works much like `coding-system-for-read`, except that it applies to output rather than input. It affects writing to files, as well as sending output to subprocesses and net connections.

When a single operation does both input and output, as do `call-process-region` and `start-process`, both `coding-system-for-read` and `coding-system-for-write` affect it.

`inhibit-eol-conversion` [User Option]

When this variable is non-`nil`, no end-of-line conversion is done, no matter which coding system is specified. This applies to all the Emacs I/O and subprocess primitives, and to the explicit encoding and decoding functions (see Section 33.9.7 [Explicit Encoding], page 203).

Sometimes, you need to prefer several coding systems for some operation, rather than fix a single one. Emacs lets you specify a priority order for using coding systems. This ordering affects the sorting of lists of coding systems returned by functions such as `find-coding-systems-region` (see Section 33.9.3 [Lisp and Coding Systems], page 195).

`coding-system-priority-list` **&optional** *highestp* [Function]
This function returns the list of coding systems in the order of their current priorities. Optional argument *highestp*, if non-`nil`, means return only the highest priority coding system.

`set-coding-system-priority` **&rest** *coding-systems* [Function]
This function puts *coding-systems* at the beginning of the priority list for coding systems, thus making their priority higher than all the rest.

`with-coding-priority` *coding-systems* **&rest** *body*... [Macro]
This macro execute *body*, like `progn` does (see Section 10.1 [Sequencing], page 120, vol. 1), with *coding-systems* at the front of the priority list for coding systems. *coding-systems* should be a list of coding systems to prefer during execution of *body*.

33.9.7 Explicit Encoding and Decoding

All the operations that transfer text in and out of Emacs have the ability to use a coding system to encode or decode the text. You can also explicitly encode and decode text using the functions in this section.

The result of encoding, and the input to decoding, are not ordinary text. They logically consist of a series of byte values; that is, a series of ASCII and eight-bit characters. In unibyte buffers and strings, these characters have codes in the range 0 through #xFF (255). In a multibyte buffer or string, eight-bit characters have character codes higher than #xFF (see Section 33.1 [Text Representations], page 182), but Emacs transparently converts them to their single-byte values when you encode or decode such text.

The usual way to read a file into a buffer as a sequence of bytes, so you can decode the contents explicitly, is with `insert-file-contents-literally` (see Section 25.3 [Reading from Files], page 467, vol. 1); alternatively, specify a non-`nil` *rawfile* argument when visiting a file with `find-file-noselect`. These methods result in a unibyte buffer.

The usual way to use the byte sequence that results from explicitly encoding text is to copy it to a file or process—for example, to write it with `write-region` (see Section 25.4 [Writing to Files], page 468, vol. 1), and suppress encoding by binding `coding-system-for-write` to `no-conversion`.

Here are the functions to perform explicit encoding or decoding. The encoding functions produce sequences of bytes; the decoding functions are meant to operate on sequences of bytes. All of these functions discard text properties. They also set `last-coding-system-used` to the precise coding system they used.

`encode-coding-region` *start end coding-system* **&optional** *destination* [Command]
This command encodes the text from *start* to *end* according to coding system *coding-system*. Normally, the encoded text replaces the original text in the buffer, but the optional argument *destination* can change that. If *destination* is a buffer, the encoded text is inserted in that buffer after point (point does not move); if it is `t`, the command returns the encoded text as a unibyte string without inserting it.

If encoded text is inserted in some buffer, this command returns the length of the encoded text.

The result of encoding is logically a sequence of bytes, but the buffer remains multibyte if it was multibyte before, and any 8-bit bytes are converted to their multibyte representation (see Section 33.1 [Text Representations], page 182).

Do *not* use `undecided` for *coding-system* when encoding text, since that may lead to unexpected results. Instead, use `select-safe-coding-system` (see Section 33.9.4 [User-Chosen Coding Systems], page 198) to suggest a suitable encoding, if there's no obvious pertinent value for *coding-system*.

`encode-coding-string` *string coding-system* **&optional** *nocopy buffer* [Function]
This function encodes the text in *string* according to coding system *coding-system*. It returns a new string containing the encoded text, except when *nocopy* is non-`nil`, in which case the function may return *string* itself if the encoding operation is trivial. The result of encoding is a unibyte string.

`decode-coding-region` *start end coding-system* **&optional** *destination* [Command]
This command decodes the text from *start* to *end* according to coding system *coding-system*. To make explicit decoding useful, the text before decoding ought to be a sequence of byte values, but both multibyte and unibyte buffers are acceptable (in the multibyte case, the raw byte values should be represented as eight-bit characters). Normally, the decoded text replaces the original text in the buffer, but the optional argument *destination* can change that. If *destination* is a buffer, the decoded text is inserted in that buffer after point (point does not move); if it is `t`, the command returns the decoded text as a multibyte string without inserting it.

If decoded text is inserted in some buffer, this command returns the length of the decoded text.

This command puts a `charset` text property on the decoded text. The value of the property states the character set used to decode the original text.

`decode-coding-string` *string coding-system* **&optional** *nocopy buffer* [Function]
This function decodes the text in *string* according to *coding-system*. It returns a new string containing the decoded text, except when *nocopy* is non-`nil`, in which case the function may return *string* itself if the decoding operation is trivial. To make explicit decoding useful, the contents of *string* ought to be a unibyte string with a sequence of byte values, but a multibyte string is also acceptable (assuming it contains 8-bit bytes in their multibyte form).

If optional argument *buffer* specifies a buffer, the decoded text is inserted in that buffer after point (point does not move). In this case, the return value is the length of the decoded text.

This function puts a `charset` text property on the decoded text. The value of the property states the character set used to decode the original text:

```
(decode-coding-string "Gr\374ss Gott" 'latin-1)
     ⇒ #("Grüss Gott" 0 9 (charset iso-8859-1))
```

`decode-coding-inserted-region` *from to filename* **&optional** *visit beg end replace* [Function]

This function decodes the text from *from* to *to* as if it were being read from file *filename* using `insert-file-contents` using the rest of the arguments provided.

The normal way to use this function is after reading text from a file without decoding, if you decide you would rather have decoded it. Instead of deleting the text and reading it again, this time with decoding, you can call this function.

33.9.8 Terminal I/O Encoding

Emacs can decode keyboard input using a coding system, and encode terminal output. This is useful for terminals that transmit or display text using a particular encoding such as Latin-1. Emacs does not set `last-coding-system-used` for encoding or decoding of terminal I/O.

`keyboard-coding-system` **&optional** *terminal* [Function]

This function returns the coding system that is in use for decoding keyboard input from *terminal*—or `nil` if no coding system is to be used for that terminal. If *terminal* is omitted or `nil`, it means the selected frame's terminal. See Section 29.2 [Multiple Terminals], page 67.

`set-keyboard-coding-system` *coding-system* **&optional** *terminal* [Command]

This command specifies *coding-system* as the coding system to use for decoding keyboard input from *terminal*. If *coding-system* is `nil`, that means do not decode keyboard input. If *terminal* is a frame, it means that frame's terminal; if it is `nil`, that means the currently selected frame's terminal. See Section 29.2 [Multiple Terminals], page 67.

`terminal-coding-system` **&optional** *terminal* [Function]

This function returns the coding system that is in use for encoding terminal output from *terminal*—or `nil` if the output is not encoded. If *terminal* is a frame, it means that frame's terminal; if it is `nil`, that means the currently selected frame's terminal.

`set-terminal-coding-system` *coding-system* **&optional** *terminal* [Command]

This command specifies *coding-system* as the coding system to use for encoding terminal output from *terminal*. If *coding-system* is `nil`, terminal output is not encoded. If *terminal* is a frame, it means that frame's terminal; if it is `nil`, that means the currently selected frame's terminal.

33.9.9 MS-DOS File Types

On MS-DOS and Microsoft Windows, Emacs guesses the appropriate end-of-line conversion for a file by looking at the file's name. This feature classifies files as *text files* and *binary files*. By "binary file" we mean a file of literal byte values that are not necessarily meant to be characters; Emacs does no end-of-line conversion and no character code conversion for them. On the other hand, the bytes in a text file are intended to represent characters; when you create a new file whose name implies that it is a text file, Emacs uses DOS end-of-line conversion.

`buffer-file-type` [Variable]

This variable, automatically buffer-local in each buffer, records the file type of the buffer's visited file. When a buffer does not specify a coding system with `buffer-file-coding-system`, this variable is used to determine which coding system to use when writing the contents of the buffer. It should be `nil` for text, `t` for binary. If it is `t`, the coding system is `no-conversion`. Otherwise, `undecided-dos` is used.

Normally this variable is set by visiting a file; it is set to `nil` if the file was visited without any actual conversion.

Its default value is used to decide how to handle files for which `file-name-buffer-file-type-alist` says nothing about the type: If the default value is non-`nil`, then these files are treated as binary: the coding system `no-conversion` is used. Otherwise, nothing special is done for them—the coding system is deduced solely from the file contents, in the usual Emacs fashion.

`file-name-buffer-file-type-alist` [User Option]

This variable holds an alist for recognizing text and binary files. Each element has the form (*regexp* . *type*), where *regexp* is matched against the file name, and *type* may be `nil` for text, `t` for binary, or a function to call to compute which. If it is a function, then it is called with a single argument (the file name) and should return `t` or `nil`.

When running on MS-DOS or MS-Windows, Emacs checks this alist to decide which coding system to use when reading a file. For a text file, `undecided-dos` is used. For a binary file, `no-conversion` is used.

If no element in this alist matches a given file name, then the default value of `buffer-file-type` says how to treat the file.

33.10 Input Methods

Input methods provide convenient ways of entering non-ASCII characters from the keyboard. Unlike coding systems, which translate non-ASCII characters to and from encodings meant to be read by programs, input methods provide human-friendly commands. (See Section "Input Methods" in *The GNU Emacs Manual*, for information on how users use input methods to enter text.) How to define input methods is not yet documented in this manual, but here we describe how to use them.

Each input method has a name, which is currently a string; in the future, symbols may also be usable as input method names.

`current-input-method` [Variable]

This variable holds the name of the input method now active in the current buffer. (It automatically becomes local in each buffer when set in any fashion.) It is `nil` if no input method is active in the buffer now.

`default-input-method` [User Option]

This variable holds the default input method for commands that choose an input method. Unlike `current-input-method`, this variable is normally global.

`set-input-method` *input-method* [Command]
: This command activates input method *input-method* for the current buffer. It also sets `default-input-method` to *input-method*. If *input-method* is `nil`, this command deactivates any input method for the current buffer.

`read-input-method-name` *prompt* **&optional** *default inhibit-null* [Function]
: This function reads an input method name with the minibuffer, prompting with *prompt*. If *default* is non-`nil`, that is returned by default, if the user enters empty input. However, if *inhibit-null* is non-`nil`, empty input signals an error.

  The returned value is a string.

`input-method-alist` [Variable]
: This variable defines all the supported input methods. Each element defines one input method, and should have the form:

```
(input-method language-env activate-func
 title description args...)
```

  Here *input-method* is the input method name, a string; *language-env* is another string, the name of the language environment this input method is recommended for. (That serves only for documentation purposes.)

  activate-func is a function to call to activate this method. The *args*, if any, are passed as arguments to *activate-func*. All told, the arguments to *activate-func* are *input-method* and the *args*.

  title is a string to display in the mode line while this method is active. *description* is a string describing this method and what it is good for.

The fundamental interface to input methods is through the variable `input-method-function`. See Section 21.8.2 [Reading One Event], page 344, vol. 1, and Section 21.8.4 [Invoking the Input Method], page 347, vol. 1.

33.11 Locales

POSIX defines a concept of "locales" which control which language to use in language-related features. These Emacs variables control how Emacs interacts with these features.

`locale-coding-system` [Variable]
: This variable specifies the coding system to use for decoding system error messages and—on X Window system only—keyboard input, for encoding the format argument to `format-time-string`, and for decoding the return value of `format-time-string`.

`system-messages-locale` [Variable]
: This variable specifies the locale to use for generating system error messages. Changing the locale can cause messages to come out in a different language or in a different orthography. If the variable is `nil`, the locale is specified by environment variables in the usual POSIX fashion.

`system-time-locale` [Variable]
: This variable specifies the locale to use for formatting time values. Changing the locale can cause messages to appear according to the conventions of a different language. If the variable is `nil`, the locale is specified by environment variables in the usual POSIX fashion.

`locale-info` *item* [Function]

This function returns locale data *item* for the current POSIX locale, if available. *item* should be one of these symbols:

`codeset` Return the character set as a string (locale item `CODESET`).

`days` Return a 7-element vector of day names (locale items `DAY_1` through `DAY_7`);

`months` Return a 12-element vector of month names (locale items `MON_1` through `MON_12`).

`paper` Return a list `(`*`width height`*`)` for the default paper size measured in millimeters (locale items `PAPER_WIDTH` and `PAPER_HEIGHT`).

If the system can't provide the requested information, or if *item* is not one of those symbols, the value is `nil`. All strings in the return value are decoded using `locale-coding-system`. See Section "Locales" in *The GNU Libc Manual*, for more information about locales and locale items.

34 Searching and Matching

GNU Emacs provides two ways to search through a buffer for specified text: exact string searches and regular expression searches. After a regular expression search, you can examine the *match data* to determine which text matched the whole regular expression or various portions of it.

The '`skip-chars...`' functions also perform a kind of searching. See Section 30.2.7 [Skipping Characters], page 107. To search for changes in character properties, see Section 32.19.3 [Property Search], page 160.

34.1 Searching for Strings

These are the primitive functions for searching through the text in a buffer. They are meant for use in programs, but you may call them interactively. If you do so, they prompt for the search string; the arguments *limit* and *noerror* are `nil`, and *repeat* is 1. For more details on interactive searching, see Section "Searching and Replacement" in *The GNU Emacs Manual*.

These search functions convert the search string to multibyte if the buffer is multibyte; they convert the search string to unibyte if the buffer is unibyte. See Section 33.1 [Text Representations], page 182.

`search-forward` *string* **&optional** *limit noerror repeat* [Command]

This function searches forward from point for an exact match for *string*. If successful, it sets point to the end of the occurrence found, and returns the new value of point. If no match is found, the value and side effects depend on *noerror* (see below).

In the following example, point is initially at the beginning of the line. Then `(search-forward "fox")` moves point after the last letter of '`fox`':

```
---------- Buffer: foo ----------
⋆The quick brown fox jumped over the lazy dog.
---------- Buffer: foo ----------

(search-forward "fox")
     ⇒ 20

---------- Buffer: foo ----------
The quick brown fox⋆ jumped over the lazy dog.
---------- Buffer: foo ----------
```

The argument *limit* specifies the bound to the search, and should be a position in the current buffer. No match extending after that position is accepted. If *limit* is omitted or `nil`, it defaults to the end of the accessible portion of the buffer.

What happens when the search fails depends on the value of *noerror*. If *noerror* is `nil`, a `search-failed` error is signaled. If *noerror* is `t`, `search-forward` returns `nil` and does nothing. If *noerror* is neither `nil` nor `t`, then `search-forward` moves point to the upper bound and returns `nil`.

The argument *noerror* only affects valid searches which fail to find a match. Invalid arguments cause errors regardless of *noerror*.

If *repeat* is a positive number *n*, it serves as a repeat count: the search is repeated *n* times, each time starting at the end of the previous time's match. If these successive searches succeed, the function succeeds, moving point and returning its new value. Otherwise the search fails, with results depending on the value of *noerror*, as described above. If *repeat* is a negative number -*n*, it serves as a repeat count of *n* for a search in the opposite (backward) direction.

`search-backward` *string* **&optional** *limit noerror repeat* [Command]
This function searches backward from point for *string*. It is like `search-forward`, except that it searches backwards rather than forwards. Backward searches leave point at the beginning of the match.

`word-search-forward` *string* **&optional** *limit noerror repeat* [Command]
This function searches forward from point for a "word" match for *string*. If it finds a match, it sets point to the end of the match found, and returns the new value of point.

Word matching regards *string* as a sequence of words, disregarding punctuation that separates them. It searches the buffer for the same sequence of words. Each word must be distinct in the buffer (searching for the word '`ball`' does not match the word '`balls`'), but the details of punctuation and spacing are ignored (searching for '`ball boy`' does match '`ball. Boy!`').

In this example, point is initially at the beginning of the buffer; the search leaves it between the '`y`' and the '`!`'.

```
---------- Buffer: foo ----------
⋆He said "Please!  Find
the ball boy!"
---------- Buffer: foo ----------

(word-search-forward "Please find the ball, boy.")
     ⇒ 36

---------- Buffer: foo ----------
He said "Please!  Find
the ball boy⋆!"
---------- Buffer: foo ----------
```

If *limit* is non-`nil`, it must be a position in the current buffer; it specifies the upper bound to the search. The match found must not extend after that position.

If *noerror* is `nil`, then `word-search-forward` signals an error if the search fails. If *noerror* is `t`, then it returns `nil` instead of signaling an error. If *noerror* is neither `nil` nor `t`, it moves point to *limit* (or the end of the accessible portion of the buffer) and returns `nil`.

If *repeat* is non-`nil`, then the search is repeated that many times. Point is positioned at the end of the last match.

Internal, `word-search-forward` and related functions use the function `word-search-regexp` to convert *string* to a regular expression that ignores punctuation.

`word-search-forward-lax` *string* **&optional** *limit noerror repeat* [Command]
: This command is identical to `word-search-forward`, except that the end of *string* need not match a word boundary, unless *string* ends in whitespace. For instance, searching for '`ball boy`' matches '`ball boyee`', but does not match '`aball boy`'.

`word-search-backward` *string* **&optional** *limit noerror repeat* [Command]
: This function searches backward from point for a word match to *string*. This function is just like `word-search-forward` except that it searches backward and normally leaves point at the beginning of the match.

`word-search-backward-lax` *string* **&optional** *limit noerror repeat* [Command]
: This command is identical to `word-search-backward`, except that the end of *string* need not match a word boundary, unless *string* ends in whitespace.

34.2 Searching and Case

By default, searches in Emacs ignore the case of the text they are searching through; if you specify searching for '`FOO`', then '`Foo`' or '`foo`' is also considered a match. This applies to regular expressions, too; thus, '`[aB]`' would match '`a`' or '`A`' or '`b`' or '`B`'.

If you do not want this feature, set the variable `case-fold-search` to `nil`. Then all letters must match exactly, including case. This is a buffer-local variable; altering the variable affects only the current buffer. (See Section 11.10.1 [Intro to Buffer-Local], page 150, vol. 1.) Alternatively, you may change the default value. In Lisp code, you will more typically use `let` to bind `case-fold-search` to the desired value.

Note that the user-level incremental search feature handles case distinctions differently. When the search string contains only lower case letters, the search ignores case, but when the search string contains one or more upper case letters, the search becomes case-sensitive. But this has nothing to do with the searching functions used in Lisp code. See Section "Incremental Search" in *The GNU Emacs Manual*.

`case-fold-search` [User Option]
: This buffer-local variable determines whether searches should ignore case. If the variable is `nil` they do not ignore case; otherwise (and by default) they do ignore case.

`case-replace` [User Option]
: This variable determines whether the higher-level replacement functions should preserve case. If the variable is `nil`, that means to use the replacement text verbatim. A non-`nil` value means to convert the case of the replacement text according to the text being replaced.

  This variable is used by passing it as an argument to the function `replace-match`. See Section 34.6.1 [Replacing Match], page 225.

34.3 Regular Expressions

A *regular expression*, or *regexp* for short, is a pattern that denotes a (possibly infinite) set of strings. Searching for matches for a regexp is a very powerful operation. This section explains how to write regexps; the following section says how to search for them.

For interactive development of regular expressions, you can use the *M-x re-builder* command. It provides a convenient interface for creating regular expressions, by giving immediate visual feedback in a separate buffer. As you edit the regexp, all its matches in the target buffer are highlighted. Each parenthesized sub-expression of the regexp is shown in a distinct face, which makes it easier to verify even very complex regexps.

34.3.1 Syntax of Regular Expressions

Regular expressions have a syntax in which a few characters are special constructs and the rest are *ordinary*. An ordinary character is a simple regular expression that matches that character and nothing else. The special characters are '`.`', '`*`', '`+`', '`?`', '`[`', '`^`', '`$`', and '`\`'; no new special characters will be defined in the future. The character '`]`' is special if it ends a character alternative (see later). The character '`-`' is special inside a character alternative. A '`[:`' and balancing '`:]`' enclose a character class inside a character alternative. Any other character appearing in a regular expression is ordinary, unless a '`\`' precedes it.

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

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

As a simple example, we can concatenate the regular expressions '`f`' and '`o`' to get the regular expression '`fo`', which matches only the string '`fo`'. Still trivial. To do something more powerful, you need to use one of the special regular expression constructs.

34.3.1.1 Special Characters in Regular Expressions

Here is a list of the characters that are special in a regular expression.

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

'`*`'
: is not a construct by itself; it is a postfix operator that means to match the preceding regular expression repetitively as many times as possible. Thus, '`o*`' matches any number of '`o`'s (including no '`o`'s).

  '`*`' always applies to the *smallest* possible preceding expression. Thus, '`fo*`' has a repeating '`o`', not a repeating '`fo`'. It matches '`f`', '`fo`', '`foo`', and so on.

  The matcher processes a '`*`' construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the '`*`'-modified construct in the hope that that will make it possible to match the rest of the pattern. For example, in matching '`ca*ar`' against the string '`caaar`', the '`a*`' first tries to match all three '`a`'s; but the rest of the pattern is '`ar`' and there is only '`r`' left to match, so this try fails. The next alternative is for '`a*`' to match only two '`a`'s. With this choice, the rest of the regexp matches successfully.

Warning: Nested repetition operators can run for an indefinitely long time, if they lead to ambiguous matching. For example, trying to match the regular expression '`\(x+y*\)*a`' against the string '`xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxz`' could take hours before it ultimately fails. Emacs must try each way of grouping the '`x`'s before concluding that none of them can work. Even worse, '`\(x*\)*`' can match the null string in infinitely many ways, so it causes an infinite loop. To avoid these problems, check nested repetitions carefully, to make sure that they do not cause combinatorial explosions in backtracking.

'`+`' is a postfix operator, similar to '`*`' except that it must match the preceding expression at least once. So, for example, '`ca+r`' matches the strings '`car`' and '`caaaar`' but not the string '`cr`', whereas '`ca*r`' matches all three strings.

'`?`' is a postfix operator, similar to '`*`' except that it must match the preceding expression either once or not at all. For example, '`ca?r`' matches '`car`' or '`cr`'; nothing else.

'`*?`', '`+?`', '`??`'

These are "non-greedy" variants of the operators '`*`', '`+`' and '`?`'. Where those operators match the largest possible substring (consistent with matching the entire containing expression), the non-greedy variants match the smallest possible substring (consistent with matching the entire containing expression).

For example, the regular expression '`c[ad]*a`' when applied to the string '`cdaaada`' matches the whole string; but the regular expression '`c[ad]*?a`', applied to that same string, matches just '`cda`'. (The smallest possible match here for '`[ad]*?`' that permits the whole expression to match is '`d`'.)

'`[ ... ]`' is a *character alternative*, which begins with '`[`' and is terminated by '`]`'. In the simplest case, the characters between the two brackets are what this character alternative can match.

Thus, '`[ad]`' matches either one '`a`' or one '`d`', and '`[ad]*`' matches any string composed of just '`a`'s and '`d`'s (including the empty string). It follows that '`c[ad]*r`' matches '`cr`', '`car`', '`cdr`', '`caddaar`', etc.

You can also include character ranges in a character alternative, by writing the starting and ending characters with a '`-`' between them. Thus, '`[a-z]`' matches any lower-case ASCII letter. Ranges may be intermixed freely with individual characters, as in '`[a-z$%.]`', which matches any lower case ASCII letter or '`$`', '`%`' or period.

If `case-fold-search` is non-`nil`, '`[a-z]`' also matches upper-case letters. Note that a range like '`[a-z]`' is not affected by the locale's collation sequence, it always represents a sequence in ASCII order.

Note also that the usual regexp special characters are not special inside a character alternative. A completely different set of characters is special inside character alternatives: '`]`', '`-`' and '`^`'.

To include a '`]`' in a character alternative, you must make it the first character. For example, '`[]a]`' matches '`]`' or '`a`'. To include a '`-`', write '`-`' as the first or last character of the character alternative, or put it after a range. Thus, '`[]-]`'

matches both ‘`]`’ and ‘`-`’. (As explained below, you cannot use ‘`\]`’ to include a ‘`]`’ inside a character alternative, since ‘`\`’ is not special there.)

To include ‘`^`’ in a character alternative, put it anywhere but at the beginning.

If a range starts with a unibyte character *c* and ends with a multibyte character *c2*, the range is divided into two parts: one spans the unibyte characters ‘`c..?\377`’, the other the multibyte characters ‘*c1*`..`*c2*’, where *c1* is the first character of the charset to which *c2* belongs.

A character alternative can also specify named character classes (see Section 34.3.1.2 [Char Classes], page 215). This is a POSIX feature. For example, ‘`[[:ascii:]]`’ matches any ASCII character. Using a character class is equivalent to mentioning each of the characters in that class; but the latter is not feasible in practice, since some classes include thousands of different characters.

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

‘`^`’ is not special in a character alternative unless it is the first character. The character following the ‘`^`’ is treated as if it were first (in other words, ‘`-`’ and ‘`]`’ are not special there).

A complemented character alternative can match a newline, unless newline is mentioned as one of the characters not to match. This is in contrast to the handling of regexps in programs such as `grep`.

You can specify named character classes, just like in character alternatives. For instance, ‘`[^[:ascii:]]`’ matches any non-ASCII character. See Section 34.3.1.2 [Char Classes], page 215.

‘`^`’ When matching a buffer, ‘`^`’ matches the empty string, but only at the beginning of a line in the text being matched (or the beginning of the accessible portion of the buffer). Otherwise it fails to match anything. Thus, ‘`^foo`’ matches a ‘`foo`’ that occurs at the beginning of a line.

When matching a string instead of a buffer, ‘`^`’ matches at the beginning of the string or after a newline character.

For historical compatibility reasons, ‘`^`’ can be used only at the beginning of the regular expression, or after ‘`\(`’, ‘`\(?:`’ or ‘`\|`’.

‘`$`’ is similar to ‘`^`’ but matches only at the end of a line (or the end of the accessible portion of the buffer). Thus, ‘`x+$`’ matches a string of one ‘`x`’ or more at the end of a line.

When matching a string instead of a buffer, ‘`$`’ matches at the end of the string or before a newline character.

For historical compatibility reasons, ‘`$`’ can be used only at the end of the regular expression, or before ‘`\)`’ or ‘`\|`’.

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

Because '\' quotes special characters, '\$' is a regular expression that matches only '$', and '\[' is a regular expression that matches only '[', and so on.

Note that '\' also has special meaning in the read syntax of Lisp strings (see Section 2.3.8 [String Type], page 18, vol. 1), and must be quoted with '\'. For example, the regular expression that matches the '\' character is '\\'. To write a Lisp string that contains the characters '\\', Lisp syntax requires you to quote each '\' with another '\'. Therefore, the read syntax for a regular expression matching '\' is "\\\\".

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

As a '\' is not special inside a character alternative, it can never remove the special meaning of '-' or ']'. So you should not quote these characters when they have no special meaning either. This would not clarify anything, since backslashes can legitimately precede these characters where they *have* special meaning, as in '[^\]' ("[^\\]" for Lisp string syntax), which matches any single character except a backslash.

In practice, most ']' that occur in regular expressions close a character alternative and hence are special. However, occasionally a regular expression may try to match a complex pattern of literal '[' and ']'. In such situations, it sometimes may be necessary to carefully parse the regexp from the start to determine which square brackets enclose a character alternative. For example, '[^][]' consists of the complemented character alternative '[^][]' (which matches any single character that is not a square bracket), followed by a literal ']'.

The exact rules are that at the beginning of a regexp, '[' is special and ']' not. This lasts until the first unquoted '[', after which we are in a character alternative; '[' is no longer special (except when it starts a character class) but ']' is special, unless it immediately follows the special '[' or that '[' followed by a '^'. This lasts until the next special ']' that does not end a character class. This ends the character alternative and restores the ordinary syntax of regular expressions; an unquoted '[' is special again and a ']' not.

34.3.1.2 Character Classes

Here is a table of the classes you can use in a character alternative, and what they mean:

'[:ascii:]'
: This matches any ASCII character (codes 0–127).

'[:alnum:]'
: This matches any letter or digit. (At present, for multibyte characters, it matches anything that has word syntax.)

'[:alpha:]'
: This matches any letter. (At present, for multibyte characters, it matches anything that has word syntax.)

'[:blank:]'
: This matches space and tab only.

‘`[:cntrl:]`’
: This matches any ASCII control character.

‘`[:digit:]`’
: This matches ‘`0`’ through ‘`9`’. Thus, ‘`[-+[:digit:]]`’ matches any digit, as well as ‘`+`’ and ‘`-`’.

‘`[:graph:]`’
: This matches graphic characters—everything except ASCII control characters, space, and the delete character.

‘`[:lower:]`’
: This matches any lower-case letter, as determined by the current case table (see Section 4.9 [Case Tables], page 61, vol. 1). If `case-fold-search` is non-`nil`, this also matches any upper-case letter.

‘`[:multibyte:]`’
: This matches any multibyte character (see Section 33.1 [Text Representations], page 182).

‘`[:nonascii:]`’
: This matches any non-ASCII character.

‘`[:print:]`’
: This matches printing characters—everything except ASCII control characters and the delete character.

‘`[:punct:]`’
: This matches any punctuation character. (At present, for multibyte characters, it matches anything that has non-word syntax.)

‘`[:space:]`’
: This matches any character that has whitespace syntax (see Section 35.2.1 [Syntax Class Table], page 235).

‘`[:unibyte:]`’
: This matches any unibyte character (see Section 33.1 [Text Representations], page 182).

‘`[:upper:]`’
: This matches any upper-case letter, as determined by the current case table (see Section 4.9 [Case Tables], page 61, vol. 1). If `case-fold-search` is non-`nil`, this also matches any lower-case letter.

‘`[:word:]`’
: This matches any character that has word syntax (see Section 35.2.1 [Syntax Class Table], page 235).

‘`[:xdigit:]`’
: This matches the hexadecimal digits: ‘`0`’ through ‘`9`’, ‘`a`’ through ‘`f`’ and ‘`A`’ through ‘`F`’.

34.3.1.3 Backslash Constructs in Regular Expressions

For the most part, '\' followed by any character matches only that character. However, there are several exceptions: certain two-character sequences starting with '\' that have special meanings. (The character after the '\' in such a sequence is always ordinary when used on its own.) Here is a table of the special '\' constructs.

'`\|`' specifies an alternative. Two regular expressions *a* and *b* with '`\|`' in between form an expression that matches anything that either *a* or *b* matches.

Thus, '`foo\|bar`' matches either '`foo`' or '`bar`' but no other string.

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

If you need full backtracking capability to handle multiple uses of '`\|`', use the POSIX regular expression functions (see Section 34.5 [POSIX Regexps], page 224).

'`\{`*m*`\}`' is a postfix operator that repeats the previous pattern exactly *m* times. Thus, '`x\{5\}`' matches the string '`xxxxx`' and nothing else. '`c[ad]\{3\}r`' matches string such as '`caaar`', '`cdddr`', '`cadar`', and so on.

'`\{`*m*`,`*n*`\}`' is a more general postfix operator that specifies repetition with a minimum of *m* repeats and a maximum of *n* repeats. If *m* is omitted, the minimum is 0; if *n* is omitted, there is no maximum.

For example, '`c[ad]\{1,2\}r`' matches the strings '`car`', '`cdr`', '`caar`', '`cadr`', '`cdar`', and '`cddr`', and nothing else.
'`\{0,1\}`' or '`\{,1\}`' is equivalent to '`?`'.
'`\{0,\}`' or '`\{,\}`' is equivalent to '`*`'.
'`\{1,\}`' is equivalent to '`+`'.

'`\( ... \)`'
is a grouping construct that serves three purposes:

1. To enclose a set of '`\|`' alternatives for other operations. Thus, the regular expression '`\(foo\|bar\)x`' matches either '`foox`' or '`barx`'.
2. To enclose a complicated expression for the postfix operators '`*`', '`+`' and '`?`' to operate on. Thus, '`ba\(na\)*`' matches '`ba`', '`bana`', '`banana`', '`bananana`', etc., with any number (zero or more) of '`na`' strings.
3. To record a matched substring for future reference with '`\`*digit*' (see below).

This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature that was assigned as a second meaning to the same '`\( ... \)`' construct because, in practice, there was usually no conflict between the two meanings. But occasionally there is a conflict, and that led to the introduction of shy groups.

'`\(?: ... \)`'
is the *shy group* construct. A shy group serves the first two purposes of an ordinary group (controlling the nesting of other operators), but it does not get a number, so you cannot refer back to its value with '`\`*digit*'. Shy groups are

particularly useful for mechanically-constructed regular expressions, because they can be added automatically without altering the numbering of ordinary, non-shy groups.

Shy groups are also called *non-capturing* or *unnumbered groups*.

'`\(?num: ... \)`'
is the *explicitly numbered group* construct. Normal groups get their number implicitly, based on their position, which can be inconvenient. This construct allows you to force a particular group number. There is no particular restriction on the numbering, e.g. you can have several groups with the same number in which case the last one to match (i.e. the rightmost match) will win. Implicitly numbered groups always get the smallest integer larger than the one of any previous group.

'`\digit`' matches the same text that matched the *digit*th occurrence of a grouping ('`\( ... \)`') construct.

In other words, after the end of a group, the matcher remembers the beginning and end of the text matched by that group. Later on in the regular expression you can use '`\`' followed by *digit* to match that same text, whatever it may have been.

The strings matching the first nine grouping constructs appearing in the entire regular expression passed to a search or matching function are assigned numbers 1 through 9 in the order that the open parentheses appear in the regular expression. So you can use '`\1`' through '`\9`' to refer to the text matched by the corresponding grouping constructs.

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

If a '`\( ... \)`' construct matches more than once (which can happen, for instance, if it is followed by '`*`'), only the last match is recorded.

If a particular grouping construct in the regular expression was never matched—for instance, if it appears inside of an alternative that wasn't used, or inside of a repetition that repeated zero times—then the corresponding '`\digit`' construct never matches anything. To use an artificial example, '`\(foo\(b*\)\|lose\)\2`' cannot match '`lose`': the second alternative inside the larger group matches it, but then '`\2`' is undefined and can't match anything. But it can match '`foobb`', because the first alternative matches '`foob`' and '`\2`' matches '`b`'.

'`\w`' matches any word-constituent character. The editor syntax table determines which characters these are. See Chapter 35 [Syntax Tables], page 234.

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

'`\scode`' matches any character whose syntax is *code*. Here *code* is a character that represents a syntax code: thus, '`w`' for word constituent, '`-`' for whitespace, '`(`' for open parenthesis, etc. To represent whitespace syntax, use either '`-`' or a space character. See Section 35.2.1 [Syntax Class Table], page 235, for a list of syntax codes and the characters that stand for them.

‘\S*code*’ matches any character whose syntax is not *code*.

‘\c*c*’ matches any character whose category is *c*. Here *c* is a character that represents a category: thus, ‘c’ for Chinese characters or ‘g’ for Greek characters in the standard category table. You can see the list of all the currently defined categories with *M-x describe-categories RET*. You can also define your own categories in addition to the standard ones using the `define-category` function (see Section 35.9 [Categories], page 247).

‘\C*c*’ matches any character whose category is not *c*.

The following regular expression constructs match the empty string—that is, they don't use up any characters—but whether they match depends on the context. For all, the beginning and end of the accessible portion of the buffer are treated as if they were the actual beginning and end of the buffer.

‘\‘’ matches the empty string, but only at the beginning of the buffer or string being matched against.

‘\’’ matches the empty string, but only at the end of the buffer or string being matched against.

‘\=’ matches the empty string, but only at point. (This construct is not defined when matching against a string.)

‘\b’ matches the empty string, but only at the beginning or end of a word. Thus, ‘\bfoo\b’ matches any occurrence of ‘foo’ as a separate word. ‘\bballs?\b’ matches ‘ball’ or ‘balls’ as a separate word.

‘\b’ matches at the beginning or end of the buffer (or string) regardless of what text appears next to it.

‘\B’ matches the empty string, but *not* at the beginning or end of a word, nor at the beginning or end of the buffer (or string).

‘\<’ matches the empty string, but only at the beginning of a word. ‘\<’ matches at the beginning of the buffer (or string) only if a word-constituent character follows.

‘\>’ matches the empty string, but only at the end of a word. ‘\>’ matches at the end of the buffer (or string) only if the contents end with a word-constituent character.

‘_<’ matches the empty string, but only at the beginning of a symbol. A symbol is a sequence of one or more word or symbol constituent characters. ‘_<’ matches at the beginning of the buffer (or string) only if a symbol-constituent character follows.

‘_>’ matches the empty string, but only at the end of a symbol. ‘_>’ matches at the end of the buffer (or string) only if the contents end with a symbol-constituent character.

Not every string is a valid regular expression. For example, a string that ends inside a character alternative without a terminating ‘]’ is invalid, and so is a string that ends with a single ‘\’. If an invalid regular expression is passed to any of the search functions, an `invalid-regexp` error is signaled.

34.3.2 Complex Regexp Example

Here is a complicated regexp which was formerly used by Emacs to recognize the end of a sentence together with any whitespace that follows. (Nowadays Emacs uses a similar but more complex default regexp constructed by the function `sentence-end`. See Section 34.8 [Standard Regexps], page 233.)

Below, we show first the regexp as a string in Lisp syntax (to distinguish spaces from tab characters), and then the result of evaluating it. The string constant begins and ends with a double-quote. '`\"`' stands for a double-quote as part of the string, '`\\`' for a backslash as part of the string, '`\t`' for a tab and '`\n`' for a newline.

```
"[.?!][]\"')}]*\\($\\| $\\|\t\\|  \\)[ \t\n]*"
     ⇒ "[.?!][]\"')}]*\\($\\| $\\|  \\|  \\)[
]*"
```

In the output, tab and newline appear as themselves.

This regular expression contains four parts in succession and can be deciphered as follows:

`[.?!]` The first part of the pattern is a character alternative that matches any one of three characters: period, question mark, and exclamation mark. The match must begin with one of these three characters. (This is one point where the new default regexp used by Emacs differs from the old. The new value also allows some non-ASCII characters that end a sentence without any following whitespace.)

`[]\"')}]*`
The second part of the pattern matches any closing braces and quotation marks, zero or more of them, that may follow the period, question mark or exclamation mark. The `\"` is Lisp syntax for a double-quote in a string. The '`*`' at the end indicates that the immediately preceding regular expression (a character alternative, in this case) may be repeated zero or more times.

`\\($\\| $\\|\t\\|  \\)`
The third part of the pattern matches the whitespace that follows the end of a sentence: the end of a line (optionally with a space), or a tab, or two spaces. The double backslashes mark the parentheses and vertical bars as regular expression syntax; the parentheses delimit a group and the vertical bars separate alternatives. The dollar sign is used to match the end of a line.

`[ \t\n]*` Finally, the last part of the pattern matches any additional whitespace beyond the minimum needed to end a sentence.

34.3.3 Regular Expression Functions

These functions operate on regular expressions.

`regexp-quote` *string* [Function]
This function returns a regular expression whose only exact match is *string*. Using this regular expression in `looking-at` will succeed only if the next characters in the buffer are *string*; using it in a search function will succeed if the text being searched contains *string*. See Section 34.4 [Regexp Search], page 221.

This allows you to request an exact string match or search when calling a function that wants a regular expression.

```
(regexp-quote "^The cat$")
     ⇒ "\\^The cat\\$"
```

One use of `regexp-quote` is to combine an exact string match with context described as a regular expression. For example, this searches for the string that is the value of *string*, surrounded by whitespace:

```
(re-search-forward
 (concat "\\s-" (regexp-quote string) "\\s-"))
```

`regexp-opt` *strings* **&optional** *paren* [Function]

This function returns an efficient regular expression that will match any of the strings in the list *strings*. This is useful when you need to make matching or searching as fast as possible—for example, for Font Lock mode[1].

If the optional argument *paren* is non-`nil`, then the returned regular expression is always enclosed by at least one parentheses-grouping construct. If *paren* is `words`, then that construct is additionally surrounded by '`\<`' and '`\>`'; alternatively, if *paren* is `symbols`, then that construct is additionally surrounded by '`\_<`' and '`\_>`' (`symbols` is often appropriate when matching programming-language keywords and the like).

This simplified definition of `regexp-opt` produces a regular expression which is equivalent to the actual value (but not as efficient):

```
(defun regexp-opt (strings &optional paren)
  (let ((open-paren (if paren "\\(" ""))
        (close-paren (if paren "\\)" "")))
    (concat open-paren
            (mapconcat 'regexp-quote strings "\\|")
            close-paren)))
```

`regexp-opt-depth` *regexp* [Function]

This function returns the total number of grouping constructs (parenthesized expressions) in *regexp*. This does not include shy groups (see Section 34.3.1.3 [Regexp Backslash], page 217).

`regexp-opt-charset` *chars* [Function]

This function returns a regular expression matching a character in the list of characters *chars*.

```
(regexp-opt-charset '(?a ?b ?c ?d ?e))
     ⇒ "[a-e]"
```

34.4 Regular Expression Searching

In GNU Emacs, you can search for the next match for a regular expression either incrementally or not. For incremental search commands, see Section "Regular Expression Search" in *The GNU Emacs Manual*. Here we describe only the search functions useful in programs. The principal one is `re-search-forward`.

[1] Note that `regexp-opt` does not guarantee that its result is absolutely the most efficient form possible. A hand-tuned regular expression can sometimes be slightly more efficient, but is almost never worth the effort.

These search functions convert the regular expression to multibyte if the buffer is multibyte; they convert the regular expression to unibyte if the buffer is unibyte. See Section 33.1 [Text Representations], page 182.

`re-search-forward` *regexp* **&optional** *limit noerror repeat* [Command]

This function searches forward in the current buffer for a string of text that is matched by the regular expression *regexp*. The function skips over any amount of text that is not matched by *regexp*, and leaves point at the end of the first match found. It returns the new value of point.

If *limit* is non-`nil`, it must be a position in the current buffer. It specifies the upper bound to the search. No match extending after that position is accepted.

If *repeat* is supplied, it must be a positive number; the search is repeated that many times; each repetition starts at the end of the previous match. If all these successive searches succeed, the search succeeds, moving point and returning its new value. Otherwise the search fails. What `re-search-forward` does when the search fails depends on the value of *noerror*:

`nil` Signal a `search-failed` error.

`t` Do nothing and return `nil`.

anything else
Move point to *limit* (or the end of the accessible portion of the buffer) and return `nil`.

In the following example, point is initially before the '`T`'. Evaluating the search call moves point to the end of that line (between the '`t`' of '`hat`' and the newline).

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(re-search-forward "[a-z]+" nil t 5)
     ⇒ 27

---------- Buffer: foo ----------
I read "The cat in the hat⋆
comes back" twice.
---------- Buffer: foo ----------
```

`re-search-backward` *regexp* **&optional** *limit noerror repeat* [Command]

This function searches backward in the current buffer for a string of text that is matched by the regular expression *regexp*, leaving point at the beginning of the first text found.

This function is analogous to `re-search-forward`, but they are not simple mirror images. `re-search-forward` finds the match whose beginning is as close as possible to the starting point. If `re-search-backward` were a perfect mirror image, it would find the match whose end is as close as possible. However, in fact it finds the match whose beginning is as close as possible (and yet ends before the starting point). The

reason for this is that matching a regular expression at a given spot always works from beginning to end, and starts at a specified beginning position.

A true mirror-image of `re-search-forward` would require a special feature for matching regular expressions from end to beginning. It's not worth the trouble of implementing that.

`string-match` *regexp string* **&optional** *start* [Function]

This function returns the index of the start of the first match for the regular expression *regexp* in *string*, or `nil` if there is no match. If *start* is non-`nil`, the search starts at that index in *string*.

For example,

```
(string-match
 "quick" "The quick brown fox jumped quickly.")
     ⇒ 4
(string-match
 "quick" "The quick brown fox jumped quickly." 8)
     ⇒ 27
```

The index of the first character of the string is 0, the index of the second character is 1, and so on.

After this function returns, the index of the first character beyond the match is available as `(match-end 0)`. See Section 34.6 [Match Data], page 225.

```
(string-match
 "quick" "The quick brown fox jumped quickly." 8)
     ⇒ 27

(match-end 0)
     ⇒ 32
```

`string-match-p` *regexp string* **&optional** *start* [Function]

This predicate function does what `string-match` does, but it avoids modifying the match data.

`looking-at` *regexp* [Function]

This function determines whether the text in the current buffer directly following point matches the regular expression *regexp*. "Directly following" means precisely that: the search is "anchored" and it can succeed only starting with the first character following point. The result is `t` if so, `nil` otherwise.

This function does not move point, but it does update the match data. See Section 34.6 [Match Data], page 225. If you need to test for a match without modifying the match data, use `looking-at-p`, described below.

In this example, point is located directly before the '`T`'. If it were anywhere else, the result would be `nil`.

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(looking-at "The cat in the hat$")
     ⇒ t
```

`looking-back` *regexp* **&optional** *limit greedy* [Function]

This function returns `t` if *regexp* matches the text immediately before point (i.e., ending at point), and `nil` otherwise.

Because regular expression matching works only going forward, this is implemented by searching backwards from point for a match that ends at point. That can be quite slow if it has to search a long distance. You can bound the time required by specifying *limit*, which says not to search before *limit*. In this case, the match that is found must begin at or after *limit*.

If *greedy* is non-`nil`, this function extends the match backwards as far as possible, stopping when a single additional previous character cannot be part of a match for regexp. When the match is extended, its starting position is allowed to occur before *limit*.

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(looking-back "read \"" 3)
     ⇒ t
(looking-back "read \"" 4)
     ⇒ nil
```

As a general recommendation, try to avoid using `looking-back` wherever possible, since it is slow. For this reason, there are no plans to add a `looking-back-p` function.

`looking-at-p` *regexp* [Function]

This predicate function works like `looking-at`, but without updating the match data.

`search-spaces-regexp` [Variable]

If this variable is non-`nil`, it should be a regular expression that says how to search for whitespace. In that case, any group of spaces in a regular expression being searched for stands for use of this regular expression. However, spaces inside of constructs such as '`[...]`' and '`*`', '`+`', '`?`' are not affected by `search-spaces-regexp`.

Since this variable affects all regular expression search and match constructs, you should bind it temporarily for as small as possible a part of the code.

34.5 POSIX Regular Expression Searching

The usual regular expression functions do backtracking when necessary to handle the '`\|`' and repetition constructs, but they continue this only until they find *some* match. Then they succeed and report the first match found.

This section describes alternative search functions which perform the full backtracking specified by the POSIX standard for regular expression matching. They continue backtracking until they have tried all possibilities and found all matches, so they can report the longest match, as required by POSIX. This is much slower, so use these functions only when you really need the longest match.

The POSIX search and match functions do not properly support the non-greedy repetition operators (see Section 34.3.1.1 [Regexp Special], page 212). This is because POSIX backtracking conflicts with the semantics of non-greedy repetition.

`posix-search-forward` *regexp* **&optional** *limit noerror repeat* [Command]
: This is like `re-search-forward` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

`posix-search-backward` *regexp* **&optional** *limit noerror repeat* [Command]
: This is like `re-search-backward` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

`posix-looking-at` *regexp* [Function]
: This is like `looking-at` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

`posix-string-match` *regexp string* **&optional** *start* [Function]
: This is like `string-match` except that it performs the full backtracking specified by the POSIX standard for regular expression matching.

34.6 The Match Data

Emacs keeps track of the start and end positions of the segments of text found during a search; this is called the *match data*. Thanks to the match data, you can search for a complex pattern, such as a date in a mail message, and then extract parts of the match under control of the pattern.

Because the match data normally describe the most recent search only, you must be careful not to do another search inadvertently between the search you wish to refer back to and the use of the match data. If you can't avoid another intervening search, you must save and restore the match data around it, to prevent it from being overwritten.

Notice that all functions are allowed to overwrite the match data unless they're explicitly documented not to do so. A consequence is that functions that are run implicitly in the background (see Section 39.10 [Timers], page 406, and Section 39.11 [Idle Timers], page 408) should likely save and restore the match data explicitly.

34.6.1 Replacing the Text that Matched

This function replaces all or part of the text matched by the last search. It works by means of the match data.

`replace-match` *replacement* **&optional** *fixedcase literal string subexp* [Function]
: This function replaces the text in the buffer (or in *string*) that was matched by the last search. It replaces that text with *replacement*.

If you did the last search in a buffer, you should specify `nil` for *string* and make sure that the current buffer when you call `replace-match` is the one in which you did the searching or matching. Then `replace-match` does the replacement by editing the buffer; it leaves point at the end of the replacement text, and returns `t`.

If you did the search in a string, pass the same string as *string*. Then `replace-match` does the replacement by constructing and returning a new string.

If *fixedcase* is non-`nil`, then `replace-match` uses the replacement text without case conversion; otherwise, it converts the replacement text depending upon the capitalization of the text to be replaced. If the original text is all upper case, this converts the replacement text to upper case. If all words of the original text are capitalized, this capitalizes all the words of the replacement text. If all the words are one-letter and they are all upper case, they are treated as capitalized words rather than all-upper-case words.

If *literal* is non-`nil`, then *replacement* is inserted exactly as it is, the only alterations being case changes as needed. If it is `nil` (the default), then the character '`\`' is treated specially. If a '`\`' appears in *replacement*, then it must be part of one of the following sequences:

'`\&`' '`\&`' stands for the entire text being replaced.

'`\`*n*' '`\`*n*', where *n* is a digit, stands for the text that matched the *n*th subexpression in the original regexp. Subexpressions are those expressions grouped inside '`\(...\)`'. If the *n*th subexpression never matched, an empty string is substituted.

'`\\`' '`\\`' stands for a single '`\`' in the replacement text.

These substitutions occur after case conversion, if any, so the strings they substitute are never case-converted.

If *subexp* is non-`nil`, that says to replace just subexpression number *subexp* of the regexp that was matched, not the entire match. For example, after matching '`foo \(ba*r\)`', calling `replace-match` with 1 as *subexp* means to replace just the text that matched '`\(ba*r\)`'.

`match-substitute-replacement` *replacement* **&optional** *fixedcase* *literal string subexp* [Function]

This function returns the text that would be inserted into the buffer by `replace-match`, but without modifying the buffer. It is useful if you want to present the user with actual replacement result, with constructs like '`\`*n*' or '`\&`' substituted with matched groups. Arguments *replacement* and optional *fixedcase*, *literal*, *string* and *subexp* have the same meaning as for `replace-match`.

34.6.2 Simple Match Data Access

This section explains how to use the match data to find out what was matched by the last search or match operation, if it succeeded.

You can ask about the entire matching text, or about a particular parenthetical subexpression of a regular expression. The *count* argument in the functions below specifies which. If *count* is zero, you are asking about the entire match. If *count* is positive, it specifies which subexpression you want.

Recall that the subexpressions of a regular expression are those expressions grouped with escaped parentheses, '`\(...\)`'. The *count*th subexpression is found by counting occurrences of '`\(`' from the beginning of the whole regular expression. The first subexpression is numbered 1, the second 2, and so on. Only regular expressions can have subexpressions—after a simple string search, the only information available is about the entire match.

Every successful search sets the match data. Therefore, you should query the match data immediately after searching, before calling any other function that might perform another search. Alternatively, you may save and restore the match data (see Section 34.6.4 [Saving Match Data], page 229) around the call to functions that could perform another search. Or use the functions that explicitly do not modify the match data; e.g. `string-match-p`.

A search which fails may or may not alter the match data. In the current implementation, it does not, but we may change it in the future. Don't try to rely on the value of the match data after a failing search.

`match-string` *count* **&optional** *in-string* [Function]
: This function returns, as a string, the text matched in the last search or match operation. It returns the entire text if *count* is zero, or just the portion corresponding to the *count*th parenthetical subexpression, if *count* is positive.

  If the last such operation was done against a string with `string-match`, then you should pass the same string as the argument *in-string*. After a buffer search or match, you should omit *in-string* or pass `nil` for it; but you should make sure that the current buffer when you call `match-string` is the one in which you did the searching or matching. Failure to follow this advice will lead to incorrect results.

  The value is `nil` if *count* is out of range, or for a subexpression inside a '`\|`' alternative that wasn't used or a repetition that repeated zero times.

`match-string-no-properties` *count* **&optional** *in-string* [Function]
: This function is like `match-string` except that the result has no text properties.

`match-beginning` *count* [Function]
: This function returns the position of the start of the text matched by the last regular expression searched for, or a subexpression of it.

  If *count* is zero, then the value is the position of the start of the entire match. Otherwise, *count* specifies a subexpression in the regular expression, and the value of the function is the starting position of the match for that subexpression.

  The value is `nil` for a subexpression inside a '`\|`' alternative that wasn't used or a repetition that repeated zero times.

`match-end` *count* [Function]
: This function is like `match-beginning` except that it returns the position of the end of the match, rather than the position of the beginning.

Here is an example of using the match data, with a comment showing the positions within the text:

```
(string-match "\\(qu\\)\\(ick\\)"
              "The quick fox jumped quickly.")
              ;0123456789
    ⇒ 4
```

```
(match-string 0 "The quick fox jumped quickly.")
     ⇒ "quick"
(match-string 1 "The quick fox jumped quickly.")
     ⇒ "qu"
(match-string 2 "The quick fox jumped quickly.")
     ⇒ "ick"

(match-beginning 1)       ; The beginning of the match
     ⇒ 4                  ;    with 'qu' is at index 4.

(match-beginning 2)       ; The beginning of the match
     ⇒ 6                  ;    with 'ick' is at index 6.

(match-end 1)             ; The end of the match
     ⇒ 6                  ;    with 'qu' is at index 6.

(match-end 2)             ; The end of the match
     ⇒ 9                  ;    with 'ick' is at index 9.
```

Here is another example. Point is initially located at the beginning of the line. Searching moves point to between the space and the word '`in`'. The beginning of the entire match is at the 9th character of the buffer ('`T`'), and the beginning of the match for the first subexpression is at the 13th character ('`c`').

```
(list
  (re-search-forward "The \\(cat \\)")
  (match-beginning 0)
  (match-beginning 1))
    ⇒ (17 9 13)

---------- Buffer: foo ----------
I read "The cat ⋆in the hat comes back" twice.
        ^   ^
        9  13
---------- Buffer: foo ----------
```

(In this case, the index returned is a buffer position; the first character of the buffer counts as 1.)

34.6.3 Accessing the Entire Match Data

The functions `match-data` and `set-match-data` read or write the entire match data, all at once.

`match-data` **&optional** *integers reuse reseat* [Function]

This function returns a list of positions (markers or integers) that record all the information on the text that the last search matched. Element zero is the position of the beginning of the match for the whole expression; element one is the position of the end of the match for the expression. The next two elements are the positions of

the beginning and end of the match for the first subexpression, and so on. In general, element number $2n$ corresponds to `(match-beginning n)`; and element number $2n+1$ corresponds to `(match-end n)`.

Normally all the elements are markers or `nil`, but if *integers* is non-`nil`, that means to use integers instead of markers. (In that case, the buffer itself is appended as an additional element at the end of the list, to facilitate complete restoration of the match data.) If the last match was done on a string with `string-match`, then integers are always used, since markers can't point into a string.

If *reuse* is non-`nil`, it should be a list. In that case, `match-data` stores the match data in *reuse*. That is, *reuse* is destructively modified. *reuse* does not need to have the right length. If it is not long enough to contain the match data, it is extended. If it is too long, the length of *reuse* stays the same, but the elements that were not used are set to `nil`. The purpose of this feature is to reduce the need for garbage collection.

If *reseat* is non-`nil`, all markers on the *reuse* list are reseated to point to nowhere.

As always, there must be no possibility of intervening searches between the call to a search function and the call to `match-data` that is intended to access the match data for that search.

```
(match-data)
     ⇒  (#<marker at 9 in foo>
         #<marker at 17 in foo>
         #<marker at 13 in foo>
         #<marker at 17 in foo>)
```

`set-match-data` *match-list* **&optional** *reseat* [Function]

This function sets the match data from the elements of *match-list*, which should be a list that was the value of a previous call to `match-data`. (More precisely, anything that has the same format will work.)

If *match-list* refers to a buffer that doesn't exist, you don't get an error; that sets the match data in a meaningless but harmless way.

If *reseat* is non-`nil`, all markers on the *match-list* list are reseated to point to nowhere.

`store-match-data` is a semi-obsolete alias for `set-match-data`.

34.6.4 Saving and Restoring the Match Data

When you call a function that may search, you may need to save and restore the match data around that call, if you want to preserve the match data from an earlier search for later use. Here is an example that shows the problem that arises if you fail to save the match data:

```
(re-search-forward "The \\(cat \\)")
     ⇒ 48
(foo)                   ; foo does more searching.
(match-end 0)
     ⇒ 61               ; Unexpected result—not 48!
```

You can save and restore the match data with `save-match-data`:

`save-match-data` *body...* [Macro]
: This macro executes *body*, saving and restoring the match data around it. The return value is the value of the last form in *body*.

You could use `set-match-data` together with `match-data` to imitate the effect of the special form `save-match-data`. Here is how:

```
(let ((data (match-data)))
  (unwind-protect
      ...   ; Ok to change the original match data.
    (set-match-data data)))
```

Emacs automatically saves and restores the match data when it runs process filter functions (see Section 37.9.2 [Filter Functions], page 273) and process sentinels (see Section 37.10 [Sentinels], page 276).

34.7 Search and Replace

If you want to find all matches for a regexp in part of the buffer, and replace them, the best way is to write an explicit loop using `re-search-forward` and `replace-match`, like this:

```
(while (re-search-forward "foo[ \t]+bar" nil t)
  (replace-match "foobar"))
```

See Section 34.6.1 [Replacing the Text that Matched], page 225, for a description of `replace-match`.

However, replacing matches in a string is more complex, especially if you want to do it efficiently. So Emacs provides a function to do this.

`replace-regexp-in-string` *regexp rep string* **&optional** *fixedcase literal subexp start* [Function]
: This function copies *string* and searches it for matches for *regexp*, and replaces them with *rep*. It returns the modified copy. If *start* is non-`nil`, the search for matches starts at that index in *string*, so matches starting before that index are not changed.

  This function uses `replace-match` to do the replacement, and it passes the optional arguments *fixedcase*, *literal* and *subexp* along to `replace-match`.

  Instead of a string, *rep* can be a function. In that case, `replace-regexp-in-string` calls *rep* for each match, passing the text of the match as its sole argument. It collects the value *rep* returns and passes that to `replace-match` as the replacement string. The match data at this point are the result of matching *regexp* against a substring of *string*.

If you want to write a command along the lines of `query-replace`, you can use `perform-replace` to do the work.

`perform-replace` *from-string replacements query-flag regexp-flag delimited-flag* **&optional** *repeat-count map start end* [Function]
: This function is the guts of `query-replace` and related commands. It searches for occurrences of *from-string* in the text between positions *start* and *end* and replaces some or all of them. If *start* is `nil` (or omitted), point is used instead, and the end of the buffer's accessible portion is used for *end*.

If *query-flag* is `nil`, it replaces all occurrences; otherwise, it asks the user what to do about each one.

If *regexp-flag* is non-`nil`, then *from-string* is considered a regular expression; otherwise, it must match literally. If *delimited-flag* is non-`nil`, then only replacements surrounded by word boundaries are considered.

The argument *replacements* specifies what to replace occurrences with. If it is a string, that string is used. It can also be a list of strings, to be used in cyclic order.

If *replacements* is a cons cell, `(function . data)`, this means to call *function* after each match to get the replacement text. This function is called with two arguments: *data*, and the number of replacements already made.

If *repeat-count* is non-`nil`, it should be an integer. Then it specifies how many times to use each of the strings in the *replacements* list before advancing cyclically to the next one.

If *from-string* contains upper-case letters, then `perform-replace` binds `case-fold-search` to `nil`, and it uses the *replacements* without altering their case.

Normally, the keymap `query-replace-map` defines the possible user responses for queries. The argument *map*, if non-`nil`, specifies a keymap to use instead of `query-replace-map`.

This function uses one of two functions to search for the next occurrence of *from-string*. These functions are specified by the values of two variables: `replace-re-search-function` and `replace-search-function`. The former is called when the argument *regexp-flag* is non-`nil`, the latter when it is `nil`.

`query-replace-map` [Variable]

This variable holds a special keymap that defines the valid user responses for `perform-replace` and the commands that use it, as well as `y-or-n-p` and `map-y-or-n-p`. This map is unusual in two ways:

- The "key bindings" are not commands, just symbols that are meaningful to the functions that use this map.
- Prefix keys are not supported; each key binding must be for a single-event key sequence. This is because the functions don't use `read-key-sequence` to get the input; instead, they read a single event and look it up "by hand".

Here are the meaningful "bindings" for `query-replace-map`. Several of them are meaningful only for `query-replace` and friends.

`act` Do take the action being considered—in other words, "yes".

`skip` Do not take action for this question—in other words, "no".

`exit` Answer this question "no", and give up on the entire series of questions, assuming that the answers will be "no".

`exit-prefix`
Like `exit`, but add the key that was pressed to `unread-comment-events`.

`act-and-exit`
Answer this question "yes", and give up on the entire series of questions, assuming that subsequent answers will be "no".

`act-and-show`
: Answer this question "yes", but show the results—don't advance yet to the next question.

`automatic`
: Answer this question and all subsequent questions in the series with "yes", without further user interaction.

`backup`
: Move back to the previous place that a question was asked about.

`edit`
: Enter a recursive edit to deal with this question—instead of any other action that would normally be taken.

`edit-replacement`
: Edit the replacement for this question in the minibuffer.

`delete-and-edit`
: Delete the text being considered, then enter a recursive edit to replace it.

`recenter`
: Redisplay and center the window, then ask the same question again.

`quit`
: Perform a quit right away. Only `y-or-n-p` and related functions use this answer.

`help`
: Display some help, then ask again.

`multi-query-replace-map` [Variable]

This variable holds a keymap that extends `query-replace-map` by providing additional keybindings that are useful in multi-buffer replacements. The additional "bindings" are:

`automatic-all`
: Answer this question and all subsequent questions in the series with "yes", without further user interaction, for all remaining buffers.

`exit-current`
: Answer this question "no", and give up on the entire series of questions for the current buffer. Continue to the next buffer in the sequence.

`replace-search-function` [Variable]

This variable specifies a function that `perform-replace` calls to search for the next string to replace. Its default value is `search-forward`. Any other value should name a function of 3 arguments: the first 3 arguments of `search-forward` (see Section 34.1 [String Search], page 209).

`replace-re-search-function` [Variable]

This variable specifies a function that `perform-replace` calls to search for the next regexp to replace. Its default value is `re-search-forward`. Any other value should name a function of 3 arguments: the first 3 arguments of `re-search-forward` (see Section 34.4 [Regexp Search], page 221).

34.8 Standard Regular Expressions Used in Editing

This section describes some variables that hold regular expressions used for certain purposes in editing:

`page-delimiter` [User Option]
: This is the regular expression describing line-beginnings that separate pages. The default value is `"^\014"` (i.e., `"^^L"` or `"^\C-l"`); this matches a line that starts with a formfeed character.

The following two regular expressions should *not* assume the match always starts at the beginning of a line; they should not use '`^`' to anchor the match. Most often, the paragraph commands do check for a match only at the beginning of a line, which means that '`^`' would be superfluous. When there is a nonzero left margin, they accept matches that start after the left margin. In that case, a '`^`' would be incorrect. However, a '`^`' is harmless in modes where a left margin is never used.

`paragraph-separate` [User Option]
: This is the regular expression for recognizing the beginning of a line that separates paragraphs. (If you change this, you may have to change `paragraph-start` also.) The default value is `"[ \t\f]*$"`, which matches a line that consists entirely of spaces, tabs, and form feeds (after its left margin).

`paragraph-start` [User Option]
: This is the regular expression for recognizing the beginning of a line that starts *or* separates paragraphs. The default value is `"\f\\|[ \t]*$"`, which matches a line containing only whitespace or starting with a form feed (after its left margin).

`sentence-end` [User Option]
: If non-`nil`, the value should be a regular expression describing the end of a sentence, including the whitespace following the sentence. (All paragraph boundaries also end sentences, regardless.)

  If the value is `nil`, as it is by default, then the function `sentence-end` constructs the regexp. That is why you should always call the function `sentence-end` to obtain the regexp to be used to recognize the end of a sentence.

`sentence-end` [Function]
: This function returns the value of the variable `sentence-end`, if non-`nil`. Otherwise it returns a default value based on the values of the variables `sentence-end-double-space` (see [Definition of sentence-end-double-space], page 142), `sentence-end-without-period`, and `sentence-end-without-space`.

35 Syntax Tables

A *syntax table* specifies the syntactic role of each character in a buffer. It can be used to determine where words, symbols, and other syntactic constructs begin and end. This information is used by many Emacs facilities, including Font Lock mode (see Section 23.6 [Font Lock Mode], page 429, vol. 1) and the various complex movement commands (see Section 30.2 [Motion], page 100).

35.1 Syntax Table Concepts

A syntax table is a char-table (see Section 6.6 [Char-Tables], page 92, vol. 1). The element at index *c* describes the character with code *c*. The element's value should be a list that encodes the syntax of the character in question.

Syntax tables are used only for moving across text, not for the Emacs Lisp reader. Emacs Lisp uses built-in syntactic rules when reading Lisp expressions, and these rules cannot be changed. (Some Lisp systems provide ways to redefine the read syntax, but we decided to leave this feature out of Emacs Lisp for simplicity.)

Each buffer has its own major mode, and each major mode has its own idea of the syntactic class of various characters. For example, in Lisp mode, the character '`;`' begins a comment, but in C mode, it terminates a statement. To support these variations, Emacs makes the syntax table local to each buffer. Typically, each major mode has its own syntax table and installs that table in each buffer that uses that mode. Changing this table alters the syntax in all those buffers as well as in any buffers subsequently put in that mode. Occasionally several similar modes share one syntax table. See Section 23.2.9 [Example Major Modes], page 411, vol. 1, for an example of how to set up a syntax table.

A syntax table can inherit the data for some characters from the standard syntax table, while specifying other characters itself. The "inherit" syntax class means "inherit this character's syntax from the standard syntax table". Just changing the standard syntax for a character affects all syntax tables that inherit from it.

`syntax-table-p` *object* [Function]

This function returns `t` if *object* is a syntax table.

35.2 Syntax Descriptors

The syntactic role of a character is called its *syntax class*. Each syntax table specifies the syntax class of each character. There is no necessary relationship between the class of a character in one syntax table and its class in any other table.

Each syntax class is designated by a mnemonic character, which serves as the name of the class when you need to specify a class. Usually, this designator character is one that is often assigned that class; however, its meaning as a designator is unvarying and independent of what syntax that character currently has. Thus, '`\`' as a designator character always means "escape character" syntax, regardless of whether the '`\`' character actually has that syntax in the current syntax table.

A *syntax descriptor* is a Lisp string that describes the syntax classes and other syntactic properties of a character. When you want to modify the syntax of a character, that is done

by calling the function `modify-syntax-entry` and passing a syntax descriptor as one of its arguments (see Section 35.3 [Syntax Table Functions], page 238).

The first character in a syntax descriptor designates the syntax class. The second character specifies a matching character (e.g. in Lisp, the matching character for '`(`' is '`)`'); if there is no matching character, put a space there. Then come the characters for any desired flags.

If no matching character or flags are needed, only one character (specifying the syntax class) is sufficient.

For example, the syntax descriptor for the character '`*`' in C mode is `". 23"` (i.e., punctuation, matching character slot unused, second character of a comment-starter, first character of a comment-ender), and the entry for '`/`' is '`. 14`' (i.e., punctuation, matching character slot unused, first character of a comment-starter, second character of a comment-ender).

35.2.1 Table of Syntax Classes

Here is a table of syntax classes, the characters that designate them, their meanings, and examples of their use.

Whitespace characters: '` `' or '`-`'
: Characters that separate symbols and words from each other. Typically, whitespace characters have no other syntactic significance, and multiple whitespace characters are syntactically equivalent to a single one. Space, tab, and formfeed are classified as whitespace in almost all major modes.

  This syntax class can be designated by either '` `' or '`-`'. Both designators are equivalent.

Word constituents: '`w`'
: Parts of words in human languages. These are typically used in variable and command names in programs. All upper- and lower-case letters, and the digits, are typically word constituents.

Symbol constituents: '`_`'
: Extra characters used in variable and command names along with word constituents. Examples include the characters '`$&*+-_<>`' in Lisp mode, which may be part of a symbol name even though they are not part of English words. In standard C, the only non-word-constituent character that is valid in symbols is underscore ('`_`').

Punctuation characters: '`.`'
: Characters used as punctuation in a human language, or used in a programming language to separate symbols from one another. Some programming language modes, such as Emacs Lisp mode, have no characters in this class since the few characters that are not symbol or word constituents all have other uses. Other programming language modes, such as C mode, use punctuation syntax for operators.

Open parenthesis characters: '`(`'
Close parenthesis characters: '`)`'
: Characters used in dissimilar pairs to surround sentences or expressions. Such a grouping is begun with an open parenthesis character and terminated with a

close. Each open parenthesis character matches a particular close parenthesis character, and vice versa. Normally, Emacs indicates momentarily the matching open parenthesis when you insert a close parenthesis. See Section 38.19 [Blinking], page 375.

In human languages, and in C code, the parenthesis pairs are '`()`', '`[]`', and '`{}`'. In Emacs Lisp, the delimiters for lists and vectors ('`()`' and '`[]`') are classified as parenthesis characters.

String quotes: '`"`'
: Characters used to delimit string constants. The same string quote character appears at the beginning and the end of a string. Such quoted strings do not nest.

  The parsing facilities of Emacs consider a string as a single token. The usual syntactic meanings of the characters in the string are suppressed.

  The Lisp modes have two string quote characters: double-quote ('`"`') and vertical bar ('`|`'). '`|`' is not used in Emacs Lisp, but it is used in Common Lisp. C also has two string quote characters: double-quote for strings, and single-quote ('`'`') for character constants.

  Human text has no string quote characters. We do not want quotation marks to turn off the usual syntactic properties of other characters in the quotation.

Escape-syntax characters: '`\`'
: Characters that start an escape sequence, such as is used in string and character constants. The character '`\`' belongs to this class in both C and Lisp. (In C, it is used thus only inside strings, but it turns out to cause no trouble to treat it this way throughout C code.)

  Characters in this class count as part of words if `words-include-escapes` is non-`nil`. See Section 30.2.2 [Word Motion], page 101.

Character quotes: '`/`'
: Characters used to quote the following character so that it loses its normal syntactic meaning. This differs from an escape character in that only the character immediately following is ever affected.

  Characters in this class count as part of words if `words-include-escapes` is non-`nil`. See Section 30.2.2 [Word Motion], page 101.

  This class is used for backslash in TEX mode.

Paired delimiters: '`$`'
: Similar to string quote characters, except that the syntactic properties of the characters between the delimiters are not suppressed. Only TEX mode uses a paired delimiter presently—the '`$`' that both enters and leaves math mode.

Expression prefixes: '`'`'
: Characters used for syntactic operators that are considered as part of an expression if they appear next to one. In Lisp modes, these characters include the apostrophe, '`'`' (used for quoting), the comma, '`,`' (used in macros), and '`#`' (used in the read syntax for certain data types).

Comment starters: '`<`'
Comment enders: '`>`'

Characters used in various languages to delimit comments. Human text has no comment characters. In Lisp, the semicolon ('`;`') starts a comment and a newline or formfeed ends one.

Inherit standard syntax: '`@`'

This syntax class does not specify a particular syntax. It says to look in the standard syntax table to find the syntax of this character.

Generic comment delimiters: '`!`'

Characters that start or end a special kind of comment. *Any* generic comment delimiter matches *any* generic comment delimiter, but they cannot match a comment starter or comment ender; generic comment delimiters can only match each other.

This syntax class is primarily meant for use with the `syntax-table` text property (see Section 35.4 [Syntax Properties], page 240). You can mark any range of characters as forming a comment, by giving the first and last characters of the range `syntax-table` properties identifying them as generic comment delimiters.

Generic string delimiters: '`|`'

Characters that start or end a string. This class differs from the string quote class in that *any* generic string delimiter can match any other generic string delimiter; but they do not match ordinary string quote characters.

This syntax class is primarily meant for use with the `syntax-table` text property (see Section 35.4 [Syntax Properties], page 240). You can mark any range of characters as forming a string constant, by giving the first and last characters of the range `syntax-table` properties identifying them as generic string delimiters.

35.2.2 Syntax Flags

In addition to the classes, entries for characters in a syntax table can specify flags. There are eight possible flags, represented by the characters '`1`', '`2`', '`3`', '`4`', '`b`', '`c`', '`n`', and '`p`'.

All the flags except '`p`' are used to describe comment delimiters. The digit flags are used for comment delimiters made up of 2 characters. They indicate that a character can *also* be part of a comment sequence, in addition to the syntactic properties associated with its character class. The flags are independent of the class and each other for the sake of characters such as '`*`' in C mode, which is a punctuation character, *and* the second character of a start-of-comment sequence ('`/*`'), *and* the first character of an end-of-comment sequence ('`*/`'). The flags '`b`', '`c`', and '`n`' are used to qualify the corresponding comment delimiter.

Here is a table of the possible flags for a character *c*, and what they mean:

- '`1`' means *c* is the start of a two-character comment-start sequence.
- '`2`' means *c* is the second character of such a sequence.
- '`3`' means *c* is the start of a two-character comment-end sequence.
- '`4`' means *c* is the second character of such a sequence.

- '`b`' means that *c* as a comment delimiter belongs to the alternative "b" comment style. For a two-character comment starter, this flag is only significant on the second char, and for a 2-character comment ender it is only significant on the first char.
- '`c`' means that *c* as a comment delimiter belongs to the alternative "c" comment style. For a two-character comment delimiter, '`c`' on either character makes it of style "c".
- '`n`' on a comment delimiter character specifies that this kind of comment can be nested. For a two-character comment delimiter, '`n`' on either character makes it nestable.

  Emacs supports several comment styles simultaneously in any one syntax table. A comment style is a set of flags '`b`', '`c`', and '`n`', so there can be up to 8 different comment styles. Each comment delimiter has a style and only matches comment delimiters of the same style. Thus if a comment starts with the comment-start sequence of style "bn", it will extend until the next matching comment-end sequence of style "bn".

  The appropriate comment syntax settings for C++ can be as follows:

  '`/`' '`124`'

  '`*`' '`23b`'

  newline '`>`'

  This defines four comment-delimiting sequences:

  '`/*`' This is a comment-start sequence for "b" style because the second character, '`*`', has the '`b`' flag.

  '`//`' This is a comment-start sequence for "a" style because the second character, '`/`', does not have the '`b`' flag.

  '`*/`' This is a comment-end sequence for "b" style because the first character, '`*`', has the '`b`' flag.

  newline This is a comment-end sequence for "a" style, because the newline character does not have the '`b`' flag.
- '`p`' identifies an additional "prefix character" for Lisp syntax. These characters are treated as whitespace when they appear between expressions. When they appear within an expression, they are handled according to their usual syntax classes.

  The function `backward-prefix-chars` moves back over these characters, as well as over characters whose primary syntax class is prefix ('`'`'). See Section 35.5 [Motion and Syntax], page 241.

35.3 Syntax Table Functions

In this section we describe functions for creating, accessing and altering syntax tables.

`make-syntax-table` **&optional** *table* [Function]

This function creates a new syntax table, with all values initialized to `nil`. If *table* is non-`nil`, it becomes the parent of the new syntax table, otherwise the standard syntax table is the parent. Like all char-tables, a syntax table inherits from its parent. Thus the original syntax of all characters in the returned syntax table is determined by the parent. See Section 6.6 [Char-Tables], page 92, vol. 1.

Most major mode syntax tables are created in this way.

`copy-syntax-table` **&optional** *table* [Function]
This function constructs a copy of *table* and returns it. If *table* is not supplied (or is `nil`), it returns a copy of the standard syntax table. Otherwise, an error is signaled if *table* is not a syntax table.

`modify-syntax-entry` *char syntax-descriptor* **&optional** *table* [Command]
This function sets the syntax entry for *char* according to *syntax-descriptor*. *char* must be a character, or a cons cell of the form `(`*min* `.` *max*`)`; in the latter case, the function sets the syntax entries for all characters in the range between *min* and *max*, inclusive.

The syntax is changed only for *table*, which defaults to the current buffer's syntax table, and not in any other syntax table.

The argument *syntax-descriptor* is a syntax descriptor for the desired syntax (i.e. a string beginning with a class designator character, and optionally containing a matching character and syntax flags). An error is signaled if the first character is not one of the seventeen syntax class designators. See Section 35.2 [Syntax Descriptors], page 234.

This function always returns `nil`. The old syntax information in the table for this character is discarded.

Examples:

```
;; Put the space character in class whitespace.
(modify-syntax-entry ?\s " ")
     ⇒ nil

;; Make '$' an open parenthesis character,
;;    with '^' as its matching close.
(modify-syntax-entry ?$ "(^")
     ⇒ nil

;; Make '^' a close parenthesis character,
;;    with '$' as its matching open.
(modify-syntax-entry ?^ ")$")
     ⇒ nil

;; Make '/' a punctuation character,
;;    the first character of a start-comment sequence,
;;    and the second character of an end-comment sequence.
;;    This is used in C mode.
(modify-syntax-entry ?/ ". 14")
     ⇒ nil
```

`char-syntax` *character* [Function]
This function returns the syntax class of *character*, represented by its mnemonic designator character. This returns *only* the class, not any matching parenthesis or flags.

An error is signaled if *char* is not a character.

The following examples apply to C mode. The first example shows that the syntax class of space is whitespace (represented by a space). The second example shows that the syntax of '/' is punctuation. This does not show the fact that it is also part of comment-start and -end sequences. The third example shows that open parenthesis is in the class of open parentheses. This does not show the fact that it has a matching character, ')'.

```
(string (char-syntax ?\s))
     ⇒ " "

(string (char-syntax ?/))
     ⇒ "."

(string (char-syntax ?\())
     ⇒ "("
```

We use `string` to make it easier to see the character returned by `char-syntax`.

`set-syntax-table` *table* [Function]
This function makes *table* the syntax table for the current buffer. It returns *table*.

`syntax-table` [Function]
This function returns the current syntax table, which is the table for the current buffer.

`with-syntax-table` *table body...* [Macro]
This macro executes *body* using *table* as the current syntax table. It returns the value of the last form in *body*, after restoring the old current syntax table.

Since each buffer has its own current syntax table, we should make that more precise: `with-syntax-table` temporarily alters the current syntax table of whichever buffer is current at the time the macro execution starts. Other buffers are not affected.

35.4 Syntax Properties

When the syntax table is not flexible enough to specify the syntax of a language, you can override the syntax table for specific character occurrences in the buffer, by applying a `syntax-table` text property. See Section 32.19 [Text Properties], page 156, for how to apply text properties.

The valid values of `syntax-table` text property are:

syntax-table
If the property value is a syntax table, that table is used instead of the current buffer's syntax table to determine the syntax for the underlying text character.

`(`*syntax-code* `.` *matching-char*`)`
A cons cell of this format specifies the syntax for the underlying text character. (see Section 35.8 [Syntax Table Internals], page 246)

`nil` If the property is `nil`, the character's syntax is determined from the current syntax table in the usual way.

`parse-sexp-lookup-properties` [Variable]
: If this is non-`nil`, the syntax scanning functions, like `forward-sexp`, pay attention to syntax text properties. Otherwise they use only the current syntax table.

`syntax-propertize-function` [Variable]
: This variable, if non-`nil`, should store a function for applying `syntax-table` properties to a specified stretch of text. It is intended to be used by major modes to install a function which applies `syntax-table` properties in some mode-appropriate way.

  The function is called by `syntax-ppss` (see Section 35.6.2 [Position Parse], page 243), and by Font Lock mode during syntactic fontification (see Section 23.6.8 [Syntactic Font Lock], page 437, vol. 1). It is called with two arguments, *start* and *end*, which are the starting and ending positions of the text on which it should act. It is allowed to call `syntax-ppss` on any position before *end*. However, it should not call `syntax-ppss-flush-cache`; so, it is not allowed to call `syntax-ppss` on some position and later modify the buffer at an earlier position.

`syntax-propertize-extend-region-functions` [Variable]
: This abnormal hook is run by the syntax parsing code prior to calling `syntax-propertize-function`. Its role is to help locate safe starting and ending buffer positions for passing to `syntax-propertize-function`. For example, a major mode can add a function to this hook to identify multi-line syntactic constructs, and ensure that the boundaries do not fall in the middle of one.

  Each function in this hook should accept two arguments, *start* and *end*. It should return either a cons cell of two adjusted buffer positions, `(`*`new-start . new-end`*`)`, or `nil` if no adjustment is necessary. The hook functions are run in turn, repeatedly, until they all return `nil`.

35.5 Motion and Syntax

This section describes functions for moving across characters that have certain syntax classes.

`skip-syntax-forward` *syntaxes* **&optional** *limit* [Function]
: This function moves point forward across characters having syntax classes mentioned in *syntaxes* (a string of syntax class characters). It stops when it encounters the end of the buffer, or position *limit* (if specified), or a character it is not supposed to skip.

  If *syntaxes* starts with '`^`', then the function skips characters whose syntax is *not* in *syntaxes*.

  The return value is the distance traveled, which is a nonnegative integer.

`skip-syntax-backward` *syntaxes* **&optional** *limit* [Function]
: This function moves point backward across characters whose syntax classes are mentioned in *syntaxes*. It stops when it encounters the beginning of the buffer, or position *limit* (if specified), or a character it is not supposed to skip.

  If *syntaxes* starts with '`^`', then the function skips characters whose syntax is *not* in *syntaxes*.

  The return value indicates the distance traveled. It is an integer that is zero or less.

`backward-prefix-chars` [Function]
: This function moves point backward over any number of characters with expression prefix syntax. This includes both characters in the expression prefix syntax class, and characters with the '`p`' flag.

35.6 Parsing Expressions

This section describes functions for parsing and scanning balanced expressions. We will refer to such expressions as *sexps*, following the terminology of Lisp, even though these functions can act on languages other than Lisp. Basically, a sexp is either a balanced parenthetical grouping, a string, or a "symbol" (i.e. a sequence of characters whose syntax is either word constituent or symbol constituent). However, characters in the expression prefix syntax class (see Section 35.2.1 [Syntax Class Table], page 235) are treated as part of the sexp if they appear next to it.

The syntax table controls the interpretation of characters, so these functions can be used for Lisp expressions when in Lisp mode and for C expressions when in C mode. See Section 30.2.6 [List Motion], page 106, for convenient higher-level functions for moving over balanced expressions.

A character's syntax controls how it changes the state of the parser, rather than describing the state itself. For example, a string delimiter character toggles the parser state between "in-string" and "in-code", but the syntax of characters does not directly say whether they are inside a string. For example (note that 15 is the syntax code for generic string delimiters),

```
(put-text-property 1 9 'syntax-table '(15 . nil))
```

does not tell Emacs that the first eight chars of the current buffer are a string, but rather that they are all string delimiters. As a result, Emacs treats them as four consecutive empty string constants.

35.6.1 Motion Commands Based on Parsing

This section describes simple point-motion functions that operate based on parsing expressions.

`scan-lists` *from count depth* [Function]
: This function scans forward *count* balanced parenthetical groupings from position *from*. It returns the position where the scan stops. If *count* is negative, the scan moves backwards.

  If *depth* is nonzero, treat the starting position as being *depth* parentheses deep. The scanner moves forward or backward through the buffer until the depth changes to zero *count* times. Hence, a positive value for *depth* has the effect of moving out *depth* levels of parenthesis from the starting position, while a negative *depth* has the effect of moving deeper by *-depth* levels of parenthesis.

  Scanning ignores comments if `parse-sexp-ignore-comments` is non-`nil`.

  If the scan reaches the beginning or end of the accessible part of the buffer before it has scanned over *count* parenthetical groupings, the return value is `nil` if the depth at that point is zero; if the depth is non-zero, a `scan-error` error is signaled.

`scan-sexps` *from count* [Function]

This function scans forward *count* sexps from position *from*. It returns the position where the scan stops. If *count* is negative, the scan moves backwards.

Scanning ignores comments if `parse-sexp-ignore-comments` is non-`nil`.

If the scan reaches the beginning or end of (the accessible part of) the buffer while in the middle of a parenthetical grouping, an error is signaled. If it reaches the beginning or end between groupings but before count is used up, `nil` is returned.

`forward-comment` *count* [Function]

This function moves point forward across *count* complete comments (that is, including the starting delimiter and the terminating delimiter if any), plus any whitespace encountered on the way. It moves backward if *count* is negative. If it encounters anything other than a comment or whitespace, it stops, leaving point at the place where it stopped. This includes (for instance) finding the end of a comment when moving forward and expecting the beginning of one. The function also stops immediately after moving over the specified number of complete comments. If *count* comments are found as expected, with nothing except whitespace between them, it returns `t`; otherwise it returns `nil`.

This function cannot tell whether the "comments" it traverses are embedded within a string. If they look like comments, it treats them as comments.

To move forward over all comments and whitespace following point, use `(forward-comment (buffer-size))`. `(buffer-size)` is a good argument to use, because the number of comments in the buffer cannot exceed that many.

35.6.2 Finding the Parse State for a Position

For syntactic analysis, such as in indentation, often the useful thing is to compute the syntactic state corresponding to a given buffer position. This function does that conveniently.

`syntax-ppss` **&optional** *pos* [Function]

This function returns the parser state that the parser would reach at position *pos* starting from the beginning of the buffer. See the next section for for a description of the parser state.

The return value is the same as if you call the low-level parsing function `parse-partial-sexp` to parse from the beginning of the buffer to *pos* (see Section 35.6.4 [Low-Level Parsing], page 245). However, `syntax-ppss` uses a cache to speed up the computation. Due to this optimization, the second value (previous complete subexpression) and sixth value (minimum parenthesis depth) in the returned parser state are not meaningful.

This function has a side effect: it adds a buffer-local entry to `before-change-functions` (see Section 32.27 [Change Hooks], page 180) for `syntax-ppss-flush-cache` (see below). This entry keeps the cache consistent as the buffer is modified. However, the cache might not be updated if `syntax-ppss` is called while `before-change-functions` is temporarily let-bound, or if the buffer is modified without running the hook, such as when using `inhibit-modification-hooks`. In those cases, it is necessary to call `syntax-ppss-flush-cache` explicitly.

`syntax-ppss-flush-cache` *beg* **&rest** *ignored-args* [Function]
: This function flushes the cache used by `syntax-ppss`, starting at position *beg*. The remaining arguments, *ignored-args*, are ignored; this function accepts them so that it can be directly used on hooks such as `before-change-functions` (see Section 32.27 [Change Hooks], page 180).

Major modes can make `syntax-ppss` run faster by specifying where it needs to start parsing.

`syntax-begin-function` [Variable]
: If this is non-`nil`, it should be a function that moves to an earlier buffer position where the parser state is equivalent to `nil`—in other words, a position outside of any comment, string, or parenthesis. `syntax-ppss` uses it to further optimize its computations, when the cache gives no help.

35.6.3 Parser State

A *parser state* is a list of ten elements describing the state of the syntactic parser, after it parses the text between a specified starting point and a specified end point in the buffer. Parsing functions such as `syntax-ppss` return a parser state as the value. Some parsing functions accept a parser state as an argument, for resuming parsing.

Here are the meanings of the elements of the parser state:

0. The depth in parentheses, counting from 0. **Warning:** this can be negative if there are more close parens than open parens between the parser's starting point and end point.
1. The character position of the start of the innermost parenthetical grouping containing the stopping point; `nil` if none.
2. The character position of the start of the last complete subexpression terminated; `nil` if none.
3. Non-`nil` if inside a string. More precisely, this is the character that will terminate the string, or `t` if a generic string delimiter character should terminate it.
4. `t` if inside a non-nestable comment (of any comment style; see Section 35.2.2 [Syntax Flags], page 237); or the comment nesting level if inside a comment that can be nested.
5. `t` if the end point is just after a quote character.
6. The minimum parenthesis depth encountered during this scan.
7. What kind of comment is active: `nil` if not in a comment or in a comment of style 'a'; 1 for a comment of style 'b'; 2 for a comment of style 'c'; and `syntax-table` for a comment that should be ended by a generic comment delimiter character.
8. The string or comment start position. While inside a comment, this is the position where the comment began; while inside a string, this is the position where the string began. When outside of strings and comments, this element is `nil`.
9. Internal data for continuing the parsing. The meaning of this data is subject to change; it is used if you pass this list as the *state* argument to another call.

Elements 1, 2, and 6 are ignored in a state which you pass as an argument to continue parsing, and elements 8 and 9 are used only in trivial cases. Those elements are mainly used internally by the parser code.

One additional piece of useful information is available from a parser state using this function:

`syntax-ppss-toplevel-pos` *state* [Function]
: This function extracts, from parser state *state*, the last position scanned in the parse which was at top level in grammatical structure. "At top level" means outside of any parentheses, comments, or strings.

  The value is `nil` if *state* represents a parse which has arrived at a top level position.

35.6.4 Low-Level Parsing

The most basic way to use the expression parser is to tell it to start at a given position with a certain state, and parse up to a specified end position.

`parse-partial-sexp` *start limit* **&optional** *target-depth stop-before state stop-comment* [Function]
: This function parses a sexp in the current buffer starting at *start*, not scanning past *limit*. It stops at position *limit* or when certain criteria described below are met, and sets point to the location where parsing stops. It returns a parser state describing the status of the parse at the point where it stops.

  If the third argument *target-depth* is non-`nil`, parsing stops if the depth in parentheses becomes equal to *target-depth*. The depth starts at 0, or at whatever is given in *state*.

  If the fourth argument *stop-before* is non-`nil`, parsing stops when it comes to any character that starts a sexp. If *stop-comment* is non-`nil`, parsing stops when it comes to the start of a comment. If *stop-comment* is the symbol `syntax-table`, parsing stops after the start of a comment or a string, or the end of a comment or a string, whichever comes first.

  If *state* is `nil`, *start* is assumed to be at the top level of parenthesis structure, such as the beginning of a function definition. Alternatively, you might wish to resume parsing in the middle of the structure. To do this, you must provide a *state* argument that describes the initial status of parsing. The value returned by a previous call to `parse-partial-sexp` will do nicely.

35.6.5 Parameters to Control Parsing

`multibyte-syntax-as-symbol` [Variable]
: If this variable is non-`nil`, `scan-sexps` treats all non-ASCII characters as symbol constituents regardless of what the syntax table says about them. (However, text properties can still override the syntax.)

`parse-sexp-ignore-comments` [User Option]
: If the value is non-`nil`, then comments are treated as whitespace by the functions in this section and by `forward-sexp`, `scan-lists` and `scan-sexps`.

The behavior of `parse-partial-sexp` is also affected by `parse-sexp-lookup-properties` (see Section 35.4 [Syntax Properties], page 240).

You can use `forward-comment` to move forward or backward over one comment or several comments.

35.7 Some Standard Syntax Tables

Most of the major modes in Emacs have their own syntax tables. Here are several of them:

`standard-syntax-table` [Function]
: This function returns the standard syntax table, which is the syntax table used in Fundamental mode.

`text-mode-syntax-table` [Variable]
: The value of this variable is the syntax table used in Text mode.

`c-mode-syntax-table` [Variable]
: The value of this variable is the syntax table for C-mode buffers.

`emacs-lisp-mode-syntax-table` [Variable]
: The value of this variable is the syntax table used in Emacs Lisp mode by editing commands. (It has no effect on the Lisp `read` function.)

35.8 Syntax Table Internals

Lisp programs don't usually work with the elements directly; the Lisp-level syntax table functions usually work with syntax descriptors (see Section 35.2 [Syntax Descriptors], page 234). Nonetheless, here we document the internal format. This format is used mostly when manipulating syntax properties.

Each element of a syntax table is a cons cell of the form `(`*`syntax-code . matching-char`*`)`. The CAR, *syntax-code*, is an integer that encodes the syntax class, and any flags. The CDR, *matching-char*, is non-`nil` if a character to match was specified.

This table gives the value of *syntax-code* which corresponds to each syntactic type.

Integer	*Class*	*Integer*	*Class*	*Integer*	*Class*
0	whitespace	5	close parenthesis	10	character quote
1	punctuation	6	expression prefix	11	comment-start
2	word	7	string quote	12	comment-end
3	symbol	8	paired delimiter	13	inherit
4	open parenthesis	9	escape	14	generic comment
15	generic string				

For example, the usual syntax value for '`(`' is `(4 . 41)`. (41 is the character code for '`)`'.)

The flags are encoded in higher order bits, starting 16 bits from the least significant bit. This table gives the power of two which corresponds to each syntax flag.

Prefix	*Flag*	*Prefix*	*Flag*	*Prefix*	*Flag*
'`1`'	`(lsh 1 16)`	'`4`'	`(lsh 1 19)`	'`b`'	`(lsh 1 21)`
'`2`'	`(lsh 1 17)`	'`p`'	`(lsh 1 20)`	'`n`'	`(lsh 1 22)`
'`3`'	`(lsh 1 18)`				

`string-to-syntax` *desc* [Function]
: This function returns the internal form corresponding to the syntax descriptor *desc*, a cons cell `(`*`syntax-code . matching-char`*`)`.

`syntax-after` *pos* [Function]

This function returns the syntax code of the character in the buffer after position *pos*, taking account of syntax properties as well as the syntax table. If *pos* is outside the buffer's accessible portion (see Section 30.4 [Narrowing], page 109), this function returns `nil`.

`syntax-class` *syntax* [Function]

This function returns the syntax class of the syntax code *syntax*. (It masks off the high 16 bits that hold the flags encoded in the syntax descriptor.) If *syntax* is `nil`, it returns `nil`; this is so evaluating the expression

```
(syntax-class (syntax-after pos))
```

where `pos` is outside the buffer's accessible portion, will yield `nil` without throwing errors or producing wrong syntax class codes.

35.9 Categories

Categories provide an alternate way of classifying characters syntactically. You can define several categories as needed, then independently assign each character to one or more categories. Unlike syntax classes, categories are not mutually exclusive; it is normal for one character to belong to several categories.

Each buffer has a *category table* which records which categories are defined and also which characters belong to each category. Each category table defines its own categories, but normally these are initialized by copying from the standard categories table, so that the standard categories are available in all modes.

Each category has a name, which is an ASCII printing character in the range ' ' to '~'. You specify the name of a category when you define it with `define-category`.

The category table is actually a char-table (see Section 6.6 [Char-Tables], page 92, vol. 1). The element of the category table at index *c* is a *category set*—a bool-vector—that indicates which categories character *c* belongs to. In this category set, if the element at index *cat* is `t`, that means category *cat* is a member of the set, and that character *c* belongs to category *cat*.

For the next three functions, the optional argument *table* defaults to the current buffer's category table.

`define-category` *char docstring* **&optional** *table* [Function]

This function defines a new category, with name *char* and documentation *docstring*, for the category table *table*.

Here's an example of defining a new category for characters that have strong right-to-left directionality (see Section 38.23 [Bidirectional Display], page 382) and using it in a special category table:

```
(defvar special-category-table-for-bidi
  (let ((category-table (make-category-table))
(uniprop-table (unicode-property-table-internal 'bidi-class)))
    (define-category ?R "Characters of bidi-class R, AL, or RLO"
                     category-table)
    (map-char-table
```

```
    #'(lambda (key val)
  (if (memq val '(R AL RLO))
      (modify-category-entry key ?R category-table)))
      uniprop-table)
     category-table))
```

`category-docstring` *category* **&optional** *table* [Function]
This function returns the documentation string of category *category* in category table *table*.

```
(category-docstring ?a)
     ⇒ "ASCII"
(category-docstring ?l)
     ⇒ "Latin"
```

`get-unused-category` **&optional** *table* [Function]
This function returns a category name (a character) which is not currently defined in *table*. If all possible categories are in use in *table*, it returns `nil`.

`category-table` [Function]
This function returns the current buffer's category table.

`category-table-p` *object* [Function]
This function returns `t` if *object* is a category table, otherwise `nil`.

`standard-category-table` [Function]
This function returns the standard category table.

`copy-category-table` **&optional** *table* [Function]
This function constructs a copy of *table* and returns it. If *table* is not supplied (or is `nil`), it returns a copy of the standard category table. Otherwise, an error is signaled if *table* is not a category table.

`set-category-table` *table* [Function]
This function makes *table* the category table for the current buffer. It returns *table*.

`make-category-table` [Function]
This creates and returns an empty category table. In an empty category table, no categories have been allocated, and no characters belong to any categories.

`make-category-set` *categories* [Function]
This function returns a new category set—a bool-vector—whose initial contents are the categories listed in the string *categories*. The elements of *categories* should be category names; the new category set has `t` for each of those categories, and `nil` for all other categories.

```
(make-category-set "al")
     ⇒ #&128"\0\0\0\0\0\0\0\0\0\0\0\0\2\20\0\0"
```

`char-category-set` *char* [Function]
This function returns the category set for character *char* in the current buffer's category table. This is the bool-vector which records which categories the character *char*

belongs to. The function `char-category-set` does not allocate storage, because it returns the same bool-vector that exists in the category table.

```
(char-category-set ?a)
     ⇒ #&128"\0\0\0\0\0\0\0\0\0\0\0\0\2\20\0\0"
```

`category-set-mnemonics` *category-set* [Function]

This function converts the category set *category-set* into a string containing the characters that designate the categories that are members of the set.

```
(category-set-mnemonics (char-category-set ?a))
     ⇒ "al"
```

`modify-category-entry` *char category* **&optional** *table reset* [Function]

This function modifies the category set of *char* in category table *table* (which defaults to the current buffer's category table). *char* can be a character, or a cons cell of the form `(`*`min`*` . `*`max`*`)`; in the latter case, the function modifies the category sets of all characters in the range between *min* and *max*, inclusive.

Normally, it modifies a category set by adding *category* to it. But if *reset* is non-`nil`, then it deletes *category* instead.

`describe-categories` **&optional** *buffer-or-name* [Command]

This function describes the category specifications in the current category table. It inserts the descriptions in a buffer, and then displays that buffer. If *buffer-or-name* is non-`nil`, it describes the category table of that buffer instead.

36 Abbrevs and Abbrev Expansion

An abbreviation or *abbrev* is a string of characters that may be expanded to a longer string. The user can insert the abbrev string and find it replaced automatically with the expansion of the abbrev. This saves typing.

The set of abbrevs currently in effect is recorded in an *abbrev table*. Each buffer has a local abbrev table, but normally all buffers in the same major mode share one abbrev table. There is also a global abbrev table. Normally both are used.

An abbrev table is represented as an obarray. See Section 8.3 [Creating Symbols], page 104, vol. 1, for information about obarrays. Each abbreviation is represented by a symbol in the obarray. The symbol's name is the abbreviation; its value is the expansion; its function definition is the hook function for performing the expansion (see Section 36.2 [Defining Abbrevs], page 251); and its property list cell contains various additional properties, including the use count and the number of times the abbreviation has been expanded (see Section 36.6 [Abbrev Properties], page 255).

Certain abbrevs, called *system abbrevs*, are defined by a major mode instead of the user. A system abbrev is identified by its non-`nil` `:system` property (see Section 36.6 [Abbrev Properties], page 255). When abbrevs are saved to an abbrev file, system abbrevs are omitted. See Section 36.3 [Abbrev Files], page 252.

Because the symbols used for abbrevs are not interned in the usual obarray, they will never appear as the result of reading a Lisp expression; in fact, normally they are never used except by the code that handles abbrevs. Therefore, it is safe to use them in a nonstandard way.

If the minor mode Abbrev mode is enabled, the buffer-local variable `abbrev-mode` is non-`nil`, and abbrevs are automatically expanded in the buffer. For the user-level commands for abbrevs, see Section "Abbrev Mode" in *The GNU Emacs Manual*.

36.1 Abbrev Tables

This section describes how to create and manipulate abbrev tables.

`make-abbrev-table` **&optional** *props* [Function]
: This function creates and returns a new, empty abbrev table—an obarray containing no symbols. It is a vector filled with zeros. *props* is a property list that is applied to the new table (see Section 36.7 [Abbrev Table Properties], page 256).

`abbrev-table-p` *object* [Function]
: This function returns a non-`nil` value if *object* is an abbrev table.

`clear-abbrev-table` *abbrev-table* [Function]
: This function undefines all the abbrevs in *abbrev-table*, leaving it empty.

`copy-abbrev-table` *abbrev-table* [Function]
: This function returns a copy of *abbrev-table*—a new abbrev table containing the same abbrev definitions. It does *not* copy any property lists; only the names, values, and functions.

`define-abbrev-table` *tabname definitions* **&optional** *docstring* **&rest** *props* [Function]

This function defines *tabname* (a symbol) as an abbrev table name, i.e., as a variable whose value is an abbrev table. It defines abbrevs in the table according to *definitions*, a list of elements of the form `(abbrevname expansion [hook] [props...])`. These elements are passed as arguments to `define-abbrev`.

The optional string *docstring* is the documentation string of the variable *tabname*. The property list *props* is applied to the abbrev table (see Section 36.7 [Abbrev Table Properties], page 256).

If this function is called more than once for the same *tabname*, subsequent calls add the definitions in *definitions* to *tabname*, rather than overwriting the entire original contents. (A subsequent call only overrides abbrevs explicitly redefined or undefined in *definitions*.)

`abbrev-table-name-list` [Variable]

This is a list of symbols whose values are abbrev tables. `define-abbrev-table` adds the new abbrev table name to this list.

`insert-abbrev-table-description` *name* **&optional** *human* [Function]

This function inserts before point a description of the abbrev table named *name*. The argument *name* is a symbol whose value is an abbrev table.

If *human* is non-`nil`, the description is human-oriented. System abbrevs are listed and identified as such. Otherwise the description is a Lisp expression—a call to `define-abbrev-table` that would define *name* as it is currently defined, but without the system abbrevs. (The mode or package using *name* is supposed to add these to *name* separately.)

36.2 Defining Abbrevs

`define-abbrev` is the low-level basic function for defining an abbrev in an abbrev table.

When a major mode defines a system abbrev, it should call `define-abbrev` and specify `t` for the `:system` property. Be aware that any saved non-"system" abbrevs are restored at startup, i.e. before some major modes are loaded. Therefore, major modes should not assume that their abbrev tables are empty when they are first loaded.

`define-abbrev` *abbrev-table name expansion* **&optional** *hook* **&rest** *props* [Function]

This function defines an abbrev named *name*, in *abbrev-table*, to expand to *expansion* and call *hook*, with properties *props* (see Section 36.6 [Abbrev Properties], page 255). The return value is *name*. The `:system` property in *props* is treated specially here: if it has the value `force`, then it will overwrite an existing definition even for a non-"system" abbrev of the same name.

name should be a string. The argument *expansion* is normally the desired expansion (a string), or `nil` to undefine the abbrev. If it is anything but a string or `nil`, then the abbreviation "expands" solely by running *hook*.

The argument *hook* is a function or `nil`. If *hook* is non-`nil`, then it is called with no arguments after the abbrev is replaced with *expansion*; point is located at the end of *expansion* when *hook* is called.

If *hook* is a non-`nil` symbol whose `no-self-insert` property is non-`nil`, *hook* can explicitly control whether to insert the self-inserting input character that triggered the expansion. If *hook* returns non-`nil` in this case, that inhibits insertion of the character. By contrast, if *hook* returns `nil`, `expand-abbrev` (or `abbrev-insert`) also returns `nil`, as if expansion had not really occurred.

Normally, `define-abbrev` sets the variable `abbrevs-changed` to `t`, if it actually changes the abbrev. This is so that some commands will offer to save the abbrevs. It does not do this for a system abbrev, since those aren't saved anyway.

`only-global-abbrevs` [User Option]
If this variable is non-`nil`, it means that the user plans to use global abbrevs only. This tells the commands that define mode-specific abbrevs to define global ones instead. This variable does not alter the behavior of the functions in this section; it is examined by their callers.

36.3 Saving Abbrevs in Files

A file of saved abbrev definitions is actually a file of Lisp code. The abbrevs are saved in the form of a Lisp program to define the same abbrev tables with the same contents. Therefore, you can load the file with `load` (see Section 15.1 [How Programs Do Loading], page 209, vol. 1). However, the function `quietly-read-abbrev-file` is provided as a more convenient interface. Emacs automatically calls this function at startup.

User-level facilities such as `save-some-buffers` can save abbrevs in a file automatically, under the control of variables described here.

`abbrev-file-name` [User Option]
This is the default file name for reading and saving abbrevs.

`quietly-read-abbrev-file` **&optional** *filename* [Function]
This function reads abbrev definitions from a file named *filename*, previously written with `write-abbrev-file`. If *filename* is omitted or `nil`, the file specified in `abbrev-file-name` is used.

As the name implies, this function does not display any messages.

`save-abbrevs` [User Option]
A non-`nil` value for `save-abbrevs` means that Emacs should offer to save abbrevs (if any have changed) when files are saved. If the value is `silently`, Emacs saves the abbrevs without asking the user. `abbrev-file-name` specifies the file to save the abbrevs in.

`abbrevs-changed` [Variable]
This variable is set non-`nil` by defining or altering any abbrevs (except system abbrevs). This serves as a flag for various Emacs commands to offer to save your abbrevs.

`write-abbrev-file` **&optional** *filename* [Command]
Save all abbrev definitions (except system abbrevs), for all abbrev tables listed in `abbrev-table-name-list`, in the file *filename*, in the form of a Lisp program that when loaded will define the same abbrevs. If *filename* is `nil` or omitted, `abbrev-file-name` is used. This function returns `nil`.

36.4 Looking Up and Expanding Abbreviations

Abbrevs are usually expanded by certain interactive commands, including `self-insert-command`. This section describes the subroutines used in writing such commands, as well as the variables they use for communication.

`abbrev-symbol` *abbrev* **&optional** *table* [Function]
: This function returns the symbol representing the abbrev named *abbrev*. It returns `nil` if that abbrev is not defined. The optional second argument *table* is the abbrev table in which to look it up. If *table* is `nil`, this function tries first the current buffer's local abbrev table, and second the global abbrev table.

`abbrev-expansion` *abbrev* **&optional** *table* [Function]
: This function returns the string that *abbrev* would expand into (as defined by the abbrev tables used for the current buffer). It returns `nil` if *abbrev* is not a valid abbrev. The optional argument *table* specifies the abbrev table to use, as in `abbrev-symbol`.

`expand-abbrev` [Command]
: This command expands the abbrev before point, if any. If point does not follow an abbrev, this command does nothing. The command returns the abbrev symbol if it did expansion, `nil` otherwise.

  If the abbrev symbol has a hook function that is a symbol whose `no-self-insert` property is non-`nil`, and if the hook function returns `nil` as its value, then `expand-abbrev` returns `nil` even though expansion did occur.

`abbrev-insert` *abbrev* **&optional** *name start end* [Function]
: This function inserts the abbrev expansion of `abbrev`, replacing the text between `start` and `end`. If `start` is omitted, it defaults to point. `name`, if non-`nil`, should be the name by which this abbrev was found (a string); it is used to figure out whether to adjust the capitalization of the expansion. The function returns `abbrev` if the abbrev was successfully inserted.

`abbrev-prefix-mark` **&optional** *arg* [Command]
: This command marks the current location of point as the beginning of an abbrev. The next call to `expand-abbrev` will use the text from here to point (where it is then) as the abbrev to expand, rather than using the previous word as usual.

  First, this command expands any abbrev before point, unless *arg* is non-`nil`. (Interactively, *arg* is the prefix argument.) Then it inserts a hyphen before point, to indicate the start of the next abbrev to be expanded. The actual expansion removes the hyphen.

`abbrev-all-caps` [User Option]
: When this is set non-`nil`, an abbrev entered entirely in upper case is expanded using all upper case. Otherwise, an abbrev entered entirely in upper case is expanded by capitalizing each word of the expansion.

`abbrev-start-location` [Variable]
: The value of this variable is a buffer position (an integer or a marker) for `expand-abbrev` to use as the start of the next abbrev to be expanded. The value can also be

`nil`, which means to use the word before point instead. `abbrev-start-location` is set to `nil` each time `expand-abbrev` is called. This variable is also set by `abbrev-prefix-mark`.

`abbrev-start-location-buffer` [Variable]
The value of this variable is the buffer for which `abbrev-start-location` has been set. Trying to expand an abbrev in any other buffer clears `abbrev-start-location`. This variable is set by `abbrev-prefix-mark`.

`last-abbrev` [Variable]
This is the `abbrev-symbol` of the most recent abbrev expanded. This information is left by `expand-abbrev` for the sake of the `unexpand-abbrev` command (see Section "Expanding Abbrevs" in *The GNU Emacs Manual*).

`last-abbrev-location` [Variable]
This is the location of the most recent abbrev expanded. This contains information left by `expand-abbrev` for the sake of the `unexpand-abbrev` command.

`last-abbrev-text` [Variable]
This is the exact expansion text of the most recent abbrev expanded, after case conversion (if any). Its value is `nil` if the abbrev has already been unexpanded. This contains information left by `expand-abbrev` for the sake of the `unexpand-abbrev` command.

`abbrev-expand-functions` [Variable]
This is a wrapper hook (see Section 23.1.1 [Running Hooks], page 396, vol. 1) run around the `expand-abbrev` function. Each function on this hook is called with a single argument: a function that performs the normal abbrev expansion. The hook function can hence do anything it wants before and after performing the expansion. It can also choose not to call its argument, thus overriding the default behavior; or it may even call it several times. The function should return the abbrev symbol if expansion took place.

The following sample code shows a simple use of `abbrev-expand-functions`. It assumes that `foo-mode` is a mode for editing certain files in which lines that start with '`#`' are comments. You want to use Text mode abbrevs for those lines. The regular local abbrev table, `foo-mode-abbrev-table` is appropriate for all other lines. See Section 36.5 [Standard Abbrev Tables], page 255, for the definitions of `local-abbrev-table` and `text-mode-abbrev-table`.

```
(defun foo-mode-abbrev-expand-function (expand)
  (if (not (save-excursion (forward-line 0) (eq (char-after) ?#)))
      ;; Performs normal expansion.
      (funcall expand)
    ;; We're inside a comment: use the text-mode abbrevs.
    (let ((local-abbrev-table text-mode-abbrev-table))
      (funcall expand))))

(add-hook 'foo-mode-hook
          #'(lambda ()
              (add-hook 'abbrev-expand-functions
                        'foo-mode-abbrev-expand-function
                        nil t)))
```

36.5 Standard Abbrev Tables

Here we list the variables that hold the abbrev tables for the preloaded major modes of Emacs.

`global-abbrev-table` [Variable]
: This is the abbrev table for mode-independent abbrevs. The abbrevs defined in it apply to all buffers. Each buffer may also have a local abbrev table, whose abbrev definitions take precedence over those in the global table.

`local-abbrev-table` [Variable]
: The value of this buffer-local variable is the (mode-specific) abbreviation table of the current buffer. It can also be a list of such tables.

`abbrev-minor-mode-table-alist` [Variable]
: The value of this variable is a list of elements of the form `(`*`mode . abbrev-table`*`)` where *mode* is the name of a variable: if the variable is bound to a non-`nil` value, then the *abbrev-table* is active, otherwise it is ignored. *abbrev-table* can also be a list of abbrev tables.

`fundamental-mode-abbrev-table` [Variable]
: This is the local abbrev table used in Fundamental mode; in other words, it is the local abbrev table in all buffers in Fundamental mode.

`text-mode-abbrev-table` [Variable]
: This is the local abbrev table used in Text mode.

`lisp-mode-abbrev-table` [Variable]
: This is the local abbrev table used in Lisp mode. It is the parent of the local abbrev table used in Emacs Lisp mode. See Section 36.7 [Abbrev Table Properties], page 256.

36.6 Abbrev Properties

Abbrevs have properties, some of which influence the way they work. You can provide them as arguments to `define-abbrev`, and manipulate them with the following functions:

`abbrev-put` *abbrev prop val* [Function]
: Set the property *prop* of *abbrev* to value *val*.

`abbrev-get` *abbrev prop* [Function]
: Return the property *prop* of *abbrev*, or `nil` if the abbrev has no such property.

The following properties have special meanings:

`:count`
: This property counts the number of times the abbrev has been expanded. If not explicitly set, it is initialized to 0 by `define-abbrev`.

`:system`
: If non-`nil`, this property marks the abbrev as a system abbrev. Such abbrevs are not saved (see Section 36.3 [Abbrev Files], page 252).

`:enable-function`
: If non-`nil`, this property should be a function of no arguments which returns `nil` if the abbrev should not be used and `t` otherwise.

`:case-fixed`
: If non-`nil`, this property indicates that the case of the abbrev's name is significant and should only match a text with the same pattern of capitalization. It also disables the code that modifies the capitalization of the expansion.

36.7 Abbrev Table Properties

Like abbrevs, abbrev tables have properties, some of which influence the way they work. You can provide them as arguments to `define-abbrev-table`, and manipulate them with the functions:

`abbrev-table-put` *table prop val* [Function]
: Set the property *prop* of abbrev table *table* to value *val*.

`abbrev-table-get` *table prop* [Function]
: Return the property *prop* of abbrev table *table*, or `nil` if the abbrev has no such property.

The following properties have special meaning:

`:enable-function`
: This is like the `:enable-function` abbrev property except that it applies to all abbrevs in the table. It is used before even trying to find the abbrev before point, so it can dynamically modify the abbrev table.

`:case-fixed`
: This is like the `:case-fixed` abbrev property except that it applies to all abbrevs in the table.

`:regexp` If non-`nil`, this property is a regular expression that indicates how to extract the name of the abbrev before point, before looking it up in the table. When the regular expression matches before point, the abbrev name is expected to be in submatch 1. If this property is `nil`, the default is to use `backward-word` and `forward-word` to find the name. This property allows the use of abbrevs whose name contains characters of non-word syntax.

`:parents` This property holds a list of tables from which to inherit other abbrevs.

`:abbrev-table-modiff`
: This property holds a counter incremented each time a new abbrev is added to the table.

37 Processes

In the terminology of operating systems, a *process* is a space in which a program can execute. Emacs runs in a process. Emacs Lisp programs can invoke other programs in processes of their own. These are called *subprocesses* or *child processes* of the Emacs process, which is their *parent process*.

A subprocess of Emacs may be *synchronous* or *asynchronous*, depending on how it is created. When you create a synchronous subprocess, the Lisp program waits for the subprocess to terminate before continuing execution. When you create an asynchronous subprocess, it can run in parallel with the Lisp program. This kind of subprocess is represented within Emacs by a Lisp object which is also called a "process". Lisp programs can use this object to communicate with the subprocess or to control it. For example, you can send signals, obtain status information, receive output from the process, or send input to it.

`processp` *object* [Function]
: This function returns `t` if *object* represents an Emacs subprocess, `nil` otherwise.

In addition to subprocesses of the current Emacs session, you can also access other processes running on your machine. See Section 37.12 [System Processes], page 278.

37.1 Functions that Create Subprocesses

There are three primitives that create a new subprocess in which to run a program. One of them, `start-process`, creates an asynchronous process and returns a process object (see Section 37.4 [Asynchronous Processes], page 264). The other two, `call-process` and `call-process-region`, create a synchronous process and do not return a process object (see Section 37.3 [Synchronous Processes], page 260). There are various higher-level functions that make use of these primitives to run particular types of process.

Synchronous and asynchronous processes are explained in the following sections. Since the three functions are all called in a similar fashion, their common arguments are described here.

In all cases, the function's *program* argument specifies the program to be run. An error is signaled if the file is not found or cannot be executed. If the file name is relative, the variable `exec-path` contains a list of directories to search. Emacs initializes `exec-path` when it starts up, based on the value of the environment variable `PATH`. The standard file name constructs, '`~`', '`.`', and '`..`', are interpreted as usual in `exec-path`, but environment variable substitutions ('`$HOME`', etc.) are not recognized; use `substitute-in-file-name` to perform them (see Section 25.8.4 [File Name Expansion], page 486, vol. 1). `nil` in this list refers to `default-directory`.

Executing a program can also try adding suffixes to the specified name:

`exec-suffixes` [Variable]
: This variable is a list of suffixes (strings) to try adding to the specified program file name. The list should include `""` if you want the name to be tried exactly as specified. The default value is system-dependent.

Please note: The argument *program* contains only the name of the program; it may not contain any command-line arguments. You must use a separate argument, *args*, to provide those, as described below.

Each of the subprocess-creating functions has a *buffer-or-name* argument that specifies where the standard output from the program will go. It should be a buffer or a buffer name; if it is a buffer name, that will create the buffer if it does not already exist. It can also be `nil`, which says to discard the output unless a filter function handles it. (See Section 37.9.2 [Filter Functions], page 273, and Chapter 19 [Read and Print], page 274, vol. 1.) Normally, you should avoid having multiple processes send output to the same buffer because their output would be intermixed randomly. For synchronous processes, you can send the output to a file instead of a buffer.

All three of the subprocess-creating functions have a `&rest` argument, *args*. The *args* must all be strings, and they are supplied to *program* as separate command line arguments. Wildcard characters and other shell constructs have no special meanings in these strings, since the strings are passed directly to the specified program.

The subprocess inherits its environment from Emacs, but you can specify overrides for it with `process-environment`. See Section 39.3 [System Environment], page 395. The subprocess gets its current directory from the value of `default-directory`.

`exec-directory` [Variable]

The value of this variable is a string, the name of a directory that contains programs that come with GNU Emacs and are intended for Emacs to invoke. The program `movemail` is an example of such a program; Rmail uses it to fetch new mail from an inbox.

`exec-path` [User Option]

The value of this variable is a list of directories to search for programs to run in subprocesses. Each element is either the name of a directory (i.e., a string), or `nil`, which stands for the default directory (which is the value of `default-directory`).

The value of `exec-path` is used by `call-process` and `start-process` when the *program* argument is not an absolute file name.

Generally, you should not modify `exec-path` directly. Instead, ensure that your `PATH` environment variable is set appropriately before starting Emacs. Trying to modify `exec-path` independently of `PATH` can lead to confusing results.

37.2 Shell Arguments

Lisp programs sometimes need to run a shell and give it a command that contains file names that were specified by the user. These programs ought to be able to support any valid file name. But the shell gives special treatment to certain characters, and if these characters occur in the file name, they will confuse the shell. To handle these characters, use the function `shell-quote-argument`:

`shell-quote-argument` *argument* [Function]

This function returns a string that represents, in shell syntax, an argument whose actual contents are *argument*. It should work reliably to concatenate the return value into a shell command and then pass it to a shell for execution.

Precisely what this function does depends on your operating system. The function is designed to work with the syntax of your system's standard shell; if you use an unusual shell, you will need to redefine this function.

```
;; This example shows the behavior on GNU and Unix systems.
(shell-quote-argument "foo > bar")
     ⇒ "foo\\ \\>\\ bar"

;; This example shows the behavior on MS-DOS and MS-Windows.
(shell-quote-argument "foo > bar")
     ⇒ "\"foo > bar\""
```

Here's an example of using `shell-quote-argument` to construct a shell command:

```
(concat "diff -c "
        (shell-quote-argument oldfile)
        " "
        (shell-quote-argument newfile))
```

The following two functions are useful for combining a list of individual command-line argument strings into a single string, and taking a string apart into a list of individual command-line arguments. These functions are mainly intended for converting user input in the minibuffer, a Lisp string, into a list of string arguments to be passed to `call-process` or `start-process`, or for converting such lists of arguments into a single Lisp string to be presented in the minibuffer or echo area.

`split-string-and-unquote` *string* **&optional** *separators* [Function]

This function splits *string* into substrings at matches for the regular expression *separators*, like `split-string` does (see Section 4.3 [Creating Strings], page 49, vol. 1); in addition, it removes quoting from the substrings. It then makes a list of the substrings and returns it.

If *separators* is omitted or `nil`, it defaults to `"\\s-+"`, which is a regular expression that matches one or more characters with whitespace syntax (see Section 35.2.1 [Syntax Class Table], page 235).

This function supports two types of quoting: enclosing a whole string in double quotes `"..."`, and quoting individual characters with a backslash escape '`\`'. The latter is also used in Lisp strings, so this function can handle those as well.

`combine-and-quote-strings` *list-of-strings* **&optional** *separator* [Function]

This function concatenates *list-of-strings* into a single string, quoting each string as necessary. It also sticks the *separator* string between each pair of strings; if *separator* is omitted or `nil`, it defaults to `" "`. The return value is the resulting string.

The strings in *list-of-strings* that need quoting are those that include *separator* as their substring. Quoting a string encloses it in double quotes `"..."`. In the simplest case, if you are consing a command from the individual command-line arguments, every argument that includes embedded blanks will be quoted.

37.3 Creating a Synchronous Process

After a *synchronous process* is created, Emacs waits for the process to terminate before continuing. Starting Dired on GNU or Unix[1] is an example of this: it runs `ls` in a synchronous process, then modifies the output slightly. Because the process is synchronous, the entire directory listing arrives in the buffer before Emacs tries to do anything with it.

While Emacs waits for the synchronous subprocess to terminate, the user can quit by typing `C-g`. The first `C-g` tries to kill the subprocess with a `SIGINT` signal; but it waits until the subprocess actually terminates before quitting. If during that time the user types another `C-g`, that kills the subprocess instantly with `SIGKILL` and quits immediately (except on MS-DOS, where killing other processes doesn't work). See Section 21.11 [Quitting], page 351, vol. 1.

The synchronous subprocess functions return an indication of how the process terminated.

The output from a synchronous subprocess is generally decoded using a coding system, much like text read from a file. The input sent to a subprocess by `call-process-region` is encoded using a coding system, much like text written into a file. See Section 33.9 [Coding Systems], page 193.

`call-process` *program* **&optional** *infile destination display* **&rest** *args* [Function]

This function calls *program* and waits for it to finish.

The standard input for the new process comes from file *infile* if *infile* is not `nil`, and from the null device otherwise. The argument *destination* says where to put the process output. Here are the possibilities:

a buffer — Insert the output in that buffer, before point. This includes both the standard output stream and the standard error stream of the process.

a string — Insert the output in a buffer with that name, before point.

`t` — Insert the output in the current buffer, before point.

`nil` — Discard the output.

0 — Discard the output, and return `nil` immediately without waiting for the subprocess to finish.

In this case, the process is not truly synchronous, since it can run in parallel with Emacs; but you can think of it as synchronous in that Emacs is essentially finished with the subprocess as soon as this function returns.

MS-DOS doesn't support asynchronous subprocesses, so this option doesn't work there.

`(:file` *file-name*`)` — Send the output to the file name specified, overwriting it if it already exists.

[1] On other systems, Emacs uses a Lisp emulation of `ls`; see Section 25.9 [Contents of Directories], page 491, vol. 1.

`(`*real-destination error-destination*`)`
: Keep the standard output stream separate from the standard error stream; deal with the ordinary output as specified by *real-destination*, and dispose of the error output according to *error-destination*. If *error-destination* is `nil`, that means to discard the error output, `t` means mix it with the ordinary output, and a string specifies a file name to redirect error output into.

  You can't directly specify a buffer to put the error output in; that is too difficult to implement. But you can achieve this result by sending the error output to a temporary file and then inserting the file into a buffer.

If *display* is non-`nil`, then `call-process` redisplays the buffer as output is inserted. (However, if the coding system chosen for decoding output is `undecided`, meaning deduce the encoding from the actual data, then redisplay sometimes cannot continue once non-ASCII characters are encountered. There are fundamental reasons why it is hard to fix this; see Section 37.9 [Output from Processes], page 271.)

Otherwise the function `call-process` does no redisplay, and the results become visible on the screen only when Emacs redisplays that buffer in the normal course of events.

The remaining arguments, *args*, are strings that specify command line arguments for the program.

The value returned by `call-process` (unless you told it not to wait) indicates the reason for process termination. A number gives the exit status of the subprocess; 0 means success, and any other value means failure. If the process terminated with a signal, `call-process` returns a string describing the signal.

In the examples below, the buffer '`foo`' is current.

```
(call-process "pwd" nil t)
     ⇒ 0

---------- Buffer: foo ----------
/home/lewis/manual
---------- Buffer: foo ----------

(call-process "grep" nil "bar" nil "lewis" "/etc/passwd")
     ⇒ 0

---------- Buffer: bar ----------
lewis:x:1001:1001:Bil Lewis,,,,:/home/lewis:/bin/bash

---------- Buffer: bar ----------
```

Here is an example of the use of `call-process`, as used to be found in the definition of the `insert-directory` function:

```
(call-process insert-directory-program nil t nil switches
              (if full-directory-p
                  (concat (file-name-as-directory file) ".")
                file))
```

`process-file` *program* **&optional** *infile buffer display* **&rest** *args* [Function]

This function processes files synchronously in a separate process. It is similar to `call-process`, but may invoke a file handler based on the value of the variable `default-directory`, which specifies the current working directory of the subprocess.

The arguments are handled in almost the same way as for `call-process`, with the following differences:

Some file handlers may not support all combinations and forms of the arguments *infile*, *buffer*, and *display*. For example, some file handlers might behave as if *display* were `nil`, regardless of the value actually passed. As another example, some file handlers might not support separating standard output and error output by way of the *buffer* argument.

If a file handler is invoked, it determines the program to run based on the first argument *program*. For instance, suppose that a handler for remote files is invoked. Then the path that is used for searching for the program might be different from `exec-path`.

The second argument *infile* may invoke a file handler. The file handler could be different from the handler chosen for the `process-file` function itself. (For example, `default-directory` could be on one remote host, and *infile* on a different remote host. Or `default-directory` could be non-special, whereas *infile* is on a remote host.)

If *buffer* is a list of the form `(`*`real-destination error-destination`*`)`, and *error-destination* names a file, then the same remarks as for *infile* apply.

The remaining arguments (*args*) will be passed to the process verbatim. Emacs is not involved in processing file names that are present in *args*. To avoid confusion, it may be best to avoid absolute file names in *args*, but rather to specify all file names as relative to `default-directory`. The function `file-relative-name` is useful for constructing such relative file names.

`process-file-side-effects` [Variable]

This variable indicates whether a call of `process-file` changes remote files.

By default, this variable is always set to `t`, meaning that a call of `process-file` could potentially change any file on a remote host. When set to `nil`, a file handler could optimize its behavior with respect to remote file attribute caching.

You should only ever change this variable with a let-binding; never with `setq`.

`call-process-region` *start end program* **&optional** *delete destination display* **&rest** *args* [Function]

This function sends the text from *start* to *end* as standard input to a process running *program*. It deletes the text sent if *delete* is non-`nil`; this is useful when *destination* is `t`, to insert the output in the current buffer in place of the input.

The arguments *destination* and *display* control what to do with the output from the subprocess, and whether to update the display as it comes in. For details, see the description of `call-process`, above. If *destination* is the integer 0, `call-process-region` discards the output and returns `nil` immediately, without waiting for the subprocess to finish (this only works if asynchronous subprocesses are supported; i.e. not on MS-DOS).

The remaining arguments, *args*, are strings that specify command line arguments for the program.

The return value of `call-process-region` is just like that of `call-process`: `nil` if you told it to return without waiting; otherwise, a number or string which indicates how the subprocess terminated.

In the following example, we use `call-process-region` to run the `cat` utility, with standard input being the first five characters in buffer '`foo`' (the word '`input`'). `cat` copies its standard input into its standard output. Since the argument *destination* is `t`, this output is inserted in the current buffer.

```
---------- Buffer: foo ----------
input*
---------- Buffer: foo ----------

(call-process-region 1 6 "cat" nil t)
     ⇒ 0

---------- Buffer: foo ----------
inputinput*
---------- Buffer: foo ----------
```

For example, the `shell-command-on-region` command uses `call-process-region` in a manner similar to this:

```
(call-process-region
 start end
 shell-file-name      ; name of program
 nil                  ; do not delete region
 buffer               ; send output to buffer
 nil                  ; no redisplay during output
 "-c" command)        ; arguments for the shell
```

`call-process-shell-command` *command* **&optional** *infile destination display* **&rest** *args* [Function]

This function executes the shell command *command* synchronously. The final arguments *args* are additional arguments to add at the end of *command*. The other arguments are handled as in `call-process`.

`process-file-shell-command` *command* **&optional** *infile destination display* **&rest** *args* [Function]

This function is like `call-process-shell-command`, but uses `process-file` internally. Depending on `default-directory`, *command* can be executed also on remote hosts.

`shell-command-to-string` *command* [Function]

This function executes *command* (a string) as a shell command, then returns the command's output as a string.

`process-lines` *program* **&rest** *args* [Function]

This function runs *program*, waits for it to finish, and returns its output as a list of strings. Each string in the list holds a single line of text output by the program; the end-of-line characters are stripped from each line. The arguments beyond *program*, *args*, are strings that specify command-line arguments with which to run the program.

If *program* exits with a non-zero exit status, this function signals an error.

This function works by calling `call-process`, so program output is decoded in the same way as for `call-process`.

37.4 Creating an Asynchronous Process

After an *asynchronous process* is created, Emacs and the subprocess both continue running immediately. The process thereafter runs in parallel with Emacs, and the two can communicate with each other using the functions described in the following sections. However, communication is only partially asynchronous: Emacs sends data to the process only when certain functions are called, and Emacs accepts data from the process only when Emacs is waiting for input or for a time delay.

Here we describe how to create an asynchronous process.

`start-process` *name buffer-or-name program* **&rest** *args* [Function]

This function creates a new asynchronous subprocess and starts the program *program* running in it. It returns a process object that stands for the new subprocess in Lisp. The argument *name* specifies the name for the process object; if a process with this name already exists, then *name* is modified (by appending '`<1>`', etc.) to be unique. The buffer *buffer-or-name* is the buffer to associate with the process.

If *program* is `nil`, Emacs opens a new pseudoterminal (pty) and associates its input and output with *buffer-or-name*, without creating a subprocess. In that case, the remaining arguments *args* are ignored.

The remaining arguments, *args*, are strings that specify command line arguments for the subprocess.

In the example below, the first process is started and runs (rather, sleeps) for 100 seconds (the output buffer '`foo`' is created immediately). Meanwhile, the second process is started, and given the name '`my-process<1>`' for the sake of uniqueness. It inserts the directory listing at the end of the buffer '`foo`', before the first process finishes. Then it finishes, and a message to that effect is inserted in the buffer. Much later, the first process finishes, and another message is inserted in the buffer for it.

```
(start-process "my-process" "foo" "sleep" "100")
     ⇒ #<process my-process>

(start-process "my-process" "foo" "ls" "-l" "/bin")
     ⇒ #<process my-process<1>>

---------- Buffer: foo ----------
total 8336
-rwxr-xr-x 1 root root 971384 Mar 30 10:14 bash
-rwxr-xr-x 1 root root 146920 Jul  5  2011 bsd-csh
...
-rwxr-xr-x 1 root root 696880 Feb 28 15:55 zsh4

Process my-process<1> finished

Process my-process finished
---------- Buffer: foo ----------
```

`start-file-process` *name buffer-or-name program* **&rest** *args* [Function]
Like `start-process`, this function starts a new asynchronous subprocess running *program* in it, and returns its process object.

The difference from `start-process` is that this function may invoked a file handler based on the value of `default-directory`. This handler ought to run *program*, perhaps on the local host, perhaps on a remote host that corresponds to `default-directory`. In the latter case, the local part of `default-directory` becomes the working directory of the process.

This function does not try to invoke file name handlers for *program* or for the *program-args*.

Depending on the implementation of the file handler, it might not be possible to apply `process-filter` or `process-sentinel` to the resulting process object. See Section 37.9.2 [Filter Functions], page 273, and Section 37.10 [Sentinels], page 276.

Some file handlers may not support `start-file-process` (for example the function `ange-ftp-hook-function`). In such cases, this function does nothing and returns `nil`.

`start-process-shell-command` *name buffer-or-name command* [Function]
This function is like `start-process`, except that it uses a shell to execute the specified command. The argument *command* is a shell command name. The variable `shell-file-name` specifies which shell to use.

The point of running a program through the shell, rather than directly with `start-process`, is so that you can employ shell features such as wildcards in the arguments. It follows that if you include any arbitrary user-specified arguments in the command, you should quote them with `shell-quote-argument` first, so that any special shell characters do *not* have their special shell meanings. See Section 37.2 [Shell Arguments], page 258. Of course, when executing commands based on user input you should also consider the security implications.

`start-file-process-shell-command` *name buffer-or-name command* [Function]
This function is like `start-process-shell-command`, but uses `start-file-process` internally. Because of this, *command* can also be executed on remote hosts, depending on `default-directory`.

`process-connection-type` [Variable]
This variable controls the type of device used to communicate with asynchronous subprocesses. If it is non-`nil`, then PTYs are used, when available. Otherwise, pipes are used.

PTYs are usually preferable for processes visible to the user, as in Shell mode, because they allow job control (`C-c`, `C-z`, etc.) to work between the process and its children, whereas pipes do not. For subprocesses used for internal purposes by programs, it is often better to use a pipe, because they are more efficient. In addition, the total number of PTYs is limited on many systems and it is good not to waste them.

The value of `process-connection-type` takes effect when `start-process` is called. So you can specify how to communicate with one subprocess by binding the variable around the call to `start-process`.

```
(let ((process-connection-type nil))  ; use a pipe
  (start-process ...))
```

To determine whether a given subprocess actually got a pipe or a PTY, use the function `process-tty-name` (see Section 37.6 [Process Information], page 266).

37.5 Deleting Processes

Deleting a process disconnects Emacs immediately from the subprocess. Processes are deleted automatically after they terminate, but not necessarily right away. You can delete a process explicitly at any time. If you explicitly delete a terminated process before it is deleted automatically, no harm results. Deleting a running process sends a signal to terminate it (and its child processes, if any), and calls the process sentinel if it has one. See Section 37.10 [Sentinels], page 276.

When a process is deleted, the process object itself continues to exist as long as other Lisp objects point to it. All the Lisp primitives that work on process objects accept deleted processes, but those that do I/O or send signals will report an error. The process mark continues to point to the same place as before, usually into a buffer where output from the process was being inserted.

`delete-exited-processes` [User Option]

This variable controls automatic deletion of processes that have terminated (due to calling `exit` or to a signal). If it is `nil`, then they continue to exist until the user runs `list-processes`. Otherwise, they are deleted immediately after they exit.

`delete-process` *process* [Function]

This function deletes a process, killing it with a `SIGKILL` signal. The argument may be a process, the name of a process, a buffer, or the name of a buffer. (A buffer or buffer-name stands for the process that `get-buffer-process` returns.) Calling `delete-process` on a running process terminates it, updates the process status, and runs the sentinel (if any) immediately. If the process has already terminated, calling `delete-process` has no effect on its status, or on the running of its sentinel (which will happen sooner or later).

```
(delete-process "*shell*")
     ⇒ nil
```

37.6 Process Information

Several functions return information about processes.

`list-processes` **&optional** *query-only buffer* [Command]

This command displays a listing of all living processes. In addition, it finally deletes any process whose status was '`Exited`' or '`Signaled`'. It returns `nil`.

The processes are shown in a buffer named '`*Process List*`' (unless you specify otherwise using the optional argument *buffer*), whose major mode is Process Menu mode.

If *query-only* is non-`nil`, it only lists processes whose query flag is non-`nil`. See Section 37.11 [Query Before Exit], page 277.

`process-list` [Function]

This function returns a list of all processes that have not been deleted.

```
(process-list)
     ⇒ (#<process display-time> #<process shell>)
```

`get-process` *name* [Function]

This function returns the process named *name* (a string), or `nil` if there is none.

```
(get-process "shell")
     ⇒ #<process shell>
```

`process-command` *process* [Function]

This function returns the command that was executed to start *process*. This is a list of strings, the first string being the program executed and the rest of the strings being the arguments that were given to the program.

```
(process-command (get-process "shell"))
     ⇒ ("bash" "-i")
```

`process-contact` *process* **&optional** *key* [Function]

This function returns information about how a network or serial process was set up. When *key* is `nil`, it returns `(`*`hostname service`*`)` for a network process, and `(`*`port speed`*`)` for a serial process. For an ordinary child process, this function always returns `t`.

If *key* is `t`, the value is the complete status information for the connection, server, or serial port; that is, the list of keywords and values specified in `make-network-process` or `make-serial-process`, except that some of the values represent the current status instead of what you specified.

For a network process, the values include (see `make-network-process` for a complete list):

`:buffer` The associated value is the process buffer.

`:filter` The associated value is the process filter function.

`:sentinel`
: The associated value is the process sentinel function.

`:remote` In a connection, the address in internal format of the remote peer.

`:local` The local address, in internal format.

`:service` In a server, if you specified `t` for *service*, this value is the actual port number.

`:local` and `:remote` are included even if they were not specified explicitly in `make-network-process`.

For a serial process, see `make-serial-process` and `serial-process-configure` for a list of keys.

If *key* is a keyword, the function returns the value corresponding to that keyword.

`process-id` *process* [Function]

This function returns the PID of *process*. This is an integer that distinguishes the process *process* from all other processes running on the same computer at the current time. The PID of a process is chosen by the operating system kernel when the process is started and remains constant as long as the process exists.

`process-name` *process* [Function]
This function returns the name of *process*, as a string.

`process-status` *process-name* [Function]
This function returns the status of *process-name* as a symbol. The argument *process-name* must be a process, a buffer, or a process name (a string).

The possible values for an actual subprocess are:

`run` for a process that is running.

`stop` for a process that is stopped but continuable.

`exit` for a process that has exited.

`signal` for a process that has received a fatal signal.

`open` for a network connection that is open.

`closed` for a network connection that is closed. Once a connection is closed, you cannot reopen it, though you might be able to open a new connection to the same place.

`connect` for a non-blocking connection that is waiting to complete.

`failed` for a non-blocking connection that has failed to complete.

`listen` for a network server that is listening.

`nil` if *process-name* is not the name of an existing process.

```
(process-status (get-buffer "*shell*"))
     ⇒ run
```

For a network connection, `process-status` returns one of the symbols `open` or `closed`. The latter means that the other side closed the connection, or Emacs did `delete-process`.

`process-live-p` *process* [Function]
This function returns non-`nil` if *process* is alive. A process is considered alive if its status is `run`, `open`, `listen`, `connect` or `stop`.

`process-type` *process* [Function]
This function returns the symbol `network` for a network connection or server, `serial` for a serial port connection, or `real` for a real subprocess.

`process-exit-status` *process* [Function]
This function returns the exit status of *process* or the signal number that killed it. (Use the result of `process-status` to determine which of those it is.) If *process* has not yet terminated, the value is 0.

`process-tty-name` *process* [Function]
This function returns the terminal name that *process* is using for its communication with Emacs—or `nil` if it is using pipes instead of a terminal (see `process-connection-type` in Section 37.4 [Asynchronous Processes], page 264). If *process* represents a program running on a remote host, the terminal name used by that program on the remote host is provided as process property `remote-tty`.

`process-coding-system` *process* [Function]
: This function returns a cons cell `(decode . encode)`, describing the coding systems in use for decoding output from, and encoding input to, *process* (see Section 33.9 [Coding Systems], page 193).

`set-process-coding-system` *process* **&optional** *decoding-system* *encoding-system* [Function]
: This function specifies the coding systems to use for subsequent output from and input to *process*. It will use *decoding-system* to decode subprocess output, and *encoding-system* to encode subprocess input.

Every process also has a property list that you can use to store miscellaneous values associated with the process.

`process-get` *process propname* [Function]
: This function returns the value of the *propname* property of *process*.

`process-put` *process propname value* [Function]
: This function sets the value of the *propname* property of *process* to *value*.

`process-plist` *process* [Function]
: This function returns the process plist of *process*.

`set-process-plist` *process plist* [Function]
: This function sets the process plist of *process* to *plist*.

37.7 Sending Input to Processes

Asynchronous subprocesses receive input when it is sent to them by Emacs, which is done with the functions in this section. You must specify the process to send input to, and the input data to send. The data appears on the "standard input" of the subprocess.

Some operating systems have limited space for buffered input in a PTY. On these systems, Emacs sends an EOF periodically amidst the other characters, to force them through. For most programs, these EOFs do no harm.

Subprocess input is normally encoded using a coding system before the subprocess receives it, much like text written into a file. You can use `set-process-coding-system` to specify which coding system to use (see Section 37.6 [Process Information], page 266). Otherwise, the coding system comes from `coding-system-for-write`, if that is non-`nil`; or else from the defaulting mechanism (see Section 33.9.5 [Default Coding Systems], page 199).

Sometimes the system is unable to accept input for that process, because the input buffer is full. When this happens, the send functions wait a short while, accepting output from subprocesses, and then try again. This gives the subprocess a chance to read more of its pending input and make space in the buffer. It also allows filters, sentinels and timers to run—so take account of that in writing your code.

In these functions, the *process* argument can be a process or the name of a process, or a buffer or buffer name (which stands for a process via `get-buffer-process`). `nil` means the current buffer's process.

`process-send-string` *process string* [Function]
This function sends *process* the contents of *string* as standard input. It returns `nil`. For example, to make a Shell buffer list files:

```
(process-send-string "shell<1>" "ls\n")
    ⇒ nil
```

`process-send-region` *process start end* [Function]
This function sends the text in the region defined by *start* and *end* as standard input to *process*.

An error is signaled unless both *start* and *end* are integers or markers that indicate positions in the current buffer. (It is unimportant which number is larger.)

`process-send-eof` **&optional** *process* [Function]
This function makes *process* see an end-of-file in its input. The EOF comes after any text already sent to it. The function returns *process*.

```
(process-send-eof "shell")
    ⇒ "shell"
```

`process-running-child-p` **&optional** *process* [Function]
This function will tell you whether a *process* has given control of its terminal to its own child process. The value is `t` if this is true, or if Emacs cannot tell; it is `nil` if Emacs can be certain that this is not so.

37.8 Sending Signals to Processes

Sending a signal to a subprocess is a way of interrupting its activities. There are several different signals, each with its own meaning. The set of signals and their names is defined by the operating system. For example, the signal `SIGINT` means that the user has typed `C-c`, or that some analogous thing has happened.

Each signal has a standard effect on the subprocess. Most signals kill the subprocess, but some stop (or resume) execution instead. Most signals can optionally be handled by programs; if the program handles the signal, then we can say nothing in general about its effects.

You can send signals explicitly by calling the functions in this section. Emacs also sends signals automatically at certain times: killing a buffer sends a `SIGHUP` signal to all its associated processes; killing Emacs sends a `SIGHUP` signal to all remaining processes. (`SIGHUP` is a signal that usually indicates that the user "hung up the phone", i.e., disconnected.)

Each of the signal-sending functions takes two optional arguments: *process* and *current-group*.

The argument *process* must be either a process, a process name, a buffer, a buffer name, or `nil`. A buffer or buffer name stands for a process through `get-buffer-process`. `nil` stands for the process associated with the current buffer. An error is signaled if *process* does not identify a process.

The argument *current-group* is a flag that makes a difference when you are running a job-control shell as an Emacs subprocess. If it is non-`nil`, then the signal is sent to the current process-group of the terminal that Emacs uses to communicate with the subprocess. If the process is a job-control shell, this means the shell's current subjob. If it is `nil`, the

signal is sent to the process group of the immediate subprocess of Emacs. If the subprocess is a job-control shell, this is the shell itself.

The flag *current-group* has no effect when a pipe is used to communicate with the subprocess, because the operating system does not support the distinction in the case of pipes. For the same reason, job-control shells won't work when a pipe is used. See `process-connection-type` in Section 37.4 [Asynchronous Processes], page 264.

`interrupt-process` **&optional** *process current-group* [Function]
: This function interrupts the process *process* by sending the signal `SIGINT`. Outside of Emacs, typing the "interrupt character" (normally `C-c` on some systems, and `DEL` on others) sends this signal. When the argument *current-group* is non-`nil`, you can think of this function as "typing `C-c`" on the terminal by which Emacs talks to the subprocess.

`kill-process` **&optional** *process current-group* [Function]
: This function kills the process *process* by sending the signal `SIGKILL`. This signal kills the subprocess immediately, and cannot be handled by the subprocess.

`quit-process` **&optional** *process current-group* [Function]
: This function sends the signal `SIGQUIT` to the process *process*. This signal is the one sent by the "quit character" (usually `C-b` or `C-\`) when you are not inside Emacs.

`stop-process` **&optional** *process current-group* [Function]
: This function stops the process *process* by sending the signal `SIGTSTP`. Use `continue-process` to resume its execution.

  Outside of Emacs, on systems with job control, the "stop character" (usually `C-z`) normally sends this signal. When *current-group* is non-`nil`, you can think of this function as "typing `C-z`" on the terminal Emacs uses to communicate with the subprocess.

`continue-process` **&optional** *process current-group* [Function]
: This function resumes execution of the process *process* by sending it the signal `SIGCONT`. This presumes that *process* was stopped previously.

`signal-process` *process signal* [Command]
: This function sends a signal to process *process*. The argument *signal* specifies which signal to send; it should be an integer, or a symbol whose name is a signal.

  The *process* argument can be a system process ID (an integer); that allows you to send signals to processes that are not children of Emacs. See Section 37.12 [System Processes], page 278.

37.9 Receiving Output from Processes

There are two ways to receive the output that a subprocess writes to its standard output stream. The output can be inserted in a buffer, which is called the associated buffer of the process (see Section 37.9.1 [Process Buffers], page 272), or a function called the *filter function* can be called to act on the output. If the process has no buffer and no filter function, its output is discarded.

When a subprocess terminates, Emacs reads any pending output, then stops reading output from that subprocess. Therefore, if the subprocess has children that are still live and still producing output, Emacs won't receive that output.

Output from a subprocess can arrive only while Emacs is waiting: when reading terminal input (see the function `waiting-for-user-input-p`), in `sit-for` and `sleep-for` (see Section 21.10 [Waiting], page 350, vol. 1), and in `accept-process-output` (see Section 37.9.4 [Accepting Output], page 275). This minimizes the problem of timing errors that usually plague parallel programming. For example, you can safely create a process and only then specify its buffer or filter function; no output can arrive before you finish, if the code in between does not call any primitive that waits.

`process-adaptive-read-buffering` [Variable]

On some systems, when Emacs reads the output from a subprocess, the output data is read in very small blocks, potentially resulting in very poor performance. This behavior can be remedied to some extent by setting the variable `process-adaptive-read-buffering` to a non-`nil` value (the default), as it will automatically delay reading from such processes, thus allowing them to produce more output before Emacs tries to read it.

It is impossible to separate the standard output and standard error streams of the subprocess, because Emacs normally spawns the subprocess inside a pseudo-TTY, and a pseudo-TTY has only one output channel. If you want to keep the output to those streams separate, you should redirect one of them to a file—for example, by using an appropriate shell command.

37.9.1 Process Buffers

A process can (and usually does) have an *associated buffer*, which is an ordinary Emacs buffer that is used for two purposes: storing the output from the process, and deciding when to kill the process. You can also use the buffer to identify a process to operate on, since in normal practice only one process is associated with any given buffer. Many applications of processes also use the buffer for editing input to be sent to the process, but this is not built into Emacs Lisp.

Unless the process has a filter function (see Section 37.9.2 [Filter Functions], page 273), its output is inserted in the associated buffer. The position to insert the output is determined by the `process-mark`, which is then updated to point to the end of the text just inserted. Usually, but not always, the `process-mark` is at the end of the buffer.

Killing the associated buffer of a process also kills the process. Emacs asks for confirmation first, if the process's `process-query-on-exit-flag` is non-`nil` (see Section 37.11 [Query Before Exit], page 277). This confirmation is done by the function `process-kill-buffer-query-function`, which is run from `kill-buffer-query-functions` (see Section 27.10 [Killing Buffers], page 13).

`process-buffer` *process* [Function]

This function returns the associated buffer of the process *process*.

```
(process-buffer (get-process "shell"))
     ⇒ #<buffer *shell*>
```

`process-mark` *process* [Function]

This function returns the process marker for *process*, which is the marker that says where to insert output from the process.

If *process* does not have a buffer, `process-mark` returns a marker that points nowhere.

Insertion of process output in a buffer uses this marker to decide where to insert, and updates it to point after the inserted text. That is why successive batches of output are inserted consecutively.

Filter functions normally should use this marker in the same fashion as is done by direct insertion of output in the buffer. For an example of a filter function that uses `process-mark`, see [Process Filter Example], page 274.

When the user is expected to enter input in the process buffer for transmission to the process, the process marker separates the new input from previous output.

`set-process-buffer` *process buffer* [Function]

This function sets the buffer associated with *process* to *buffer*. If *buffer* is `nil`, the process becomes associated with no buffer.

`get-buffer-process` *buffer-or-name* [Function]

This function returns a nondeleted process associated with the buffer specified by *buffer-or-name*. If there are several processes associated with it, this function chooses one (currently, the one most recently created, but don't count on that). Deletion of a process (see `delete-process`) makes it ineligible for this function to return.

It is usually a bad idea to have more than one process associated with the same buffer.

```
(get-buffer-process "*shell*")
     ⇒ #<process shell>
```

Killing the process's buffer deletes the process, which kills the subprocess with a `SIGHUP` signal (see Section 37.8 [Signals to Processes], page 270).

37.9.2 Process Filter Functions

A process *filter function* is a function that receives the standard output from the associated process. If a process has a filter, then *all* output from that process is passed to the filter. The process buffer is used directly for output from the process only when there is no filter.

The filter function can only be called when Emacs is waiting for something, because process output arrives only at such times. Emacs waits when reading terminal input (see the function `waiting-for-user-input-p`), in `sit-for` and `sleep-for` (see Section 21.10 [Waiting], page 350, vol. 1), and in `accept-process-output` (see Section 37.9.4 [Accepting Output], page 275).

A filter function must accept two arguments: the associated process and a string, which is output just received from it. The function is then free to do whatever it chooses with the output.

Quitting is normally inhibited within a filter function—otherwise, the effect of typing `C-g` at command level or to quit a user command would be unpredictable. If you want to permit quitting inside a filter function, bind `inhibit-quit` to `nil`. In most cases, the right way to do this is with the macro `with-local-quit`. See Section 21.11 [Quitting], page 351, vol. 1.

If an error happens during execution of a filter function, it is caught automatically, so that it doesn't stop the execution of whatever program was running when the filter function was started. However, if `debug-on-error` is non-`nil`, errors are not caught. This makes it possible to use the Lisp debugger to debug the filter function. See Section 18.1 [Debugger], page 243, vol. 1.

Many filter functions sometimes (or always) insert the output in the process's buffer, mimicking the actions of Emacs when there is no filter. Such filter functions need to make sure that they save the current buffer, select the correct buffer (if different) before inserting output, and then restore the original buffer. They should also check whether the buffer is still alive, update the process marker, and in some cases update the value of point. Here is how to do these things:

```
(defun ordinary-insertion-filter (proc string)
  (when (buffer-live-p (process-buffer proc))
    (with-current-buffer (process-buffer proc)
      (let ((moving (= (point) (process-mark proc))))
        (save-excursion
          ;; Insert the text, advancing the process marker.
          (goto-char (process-mark proc))
          (insert string)
          (set-marker (process-mark proc) (point)))
        (if moving (goto-char (process-mark proc)))))))
```

To make the filter force the process buffer to be visible whenever new text arrives, you could insert a line like the following just before the `with-current-buffer` construct:

```
(display-buffer (process-buffer proc))
```

To force point to the end of the new output, no matter where it was previously, eliminate the variable `moving` and call `goto-char` unconditionally.

Note that Emacs automatically saves and restores the match data while executing filter functions. See Section 34.6 [Match Data], page 225.

The output to the filter may come in chunks of any size. A program that produces the same output twice in a row may send it as one batch of 200 characters one time, and five batches of 40 characters the next. If the filter looks for certain text strings in the subprocess output, make sure to handle the case where one of these strings is split across two or more batches of output; one way to do this is to insert the received text into a temporary buffer, which can then be searched.

`set-process-filter` *process filter* [Function]
: This function gives *process* the filter function *filter*. If *filter* is `nil`, it gives the process no filter.

`process-filter` *process* [Function]
: This function returns the filter function of *process*, or `nil` if it has none.

Here is an example of the use of a filter function:

```
(defun keep-output (process output)
   (setq kept (cons output kept)))
     ⇒ keep-output
(setq kept nil)
     ⇒ nil
(set-process-filter (get-process "shell") 'keep-output)
     ⇒ keep-output
```

```
(process-send-string "shell" "ls ~/other\n")
     ⇒ nil
kept
     ⇒ ("lewis@slug:$ "
"FINAL-W87-SHORT.MSS    backup.otl              kolstad.mss~
address.txt             backup.psf              kolstad.psf
backup.bib~             david.mss               resume-Dec-86.mss~
backup.err              david.psf               resume-Dec.psf
backup.mss              dland                   syllabus.mss
"
"#backups.mss#          backup.mss~             kolstad.mss
")
```

37.9.3 Decoding Process Output

When Emacs writes process output directly into a multibyte buffer, it decodes the output according to the process output coding system. If the coding system is `raw-text` or `no-conversion`, Emacs converts the unibyte output to multibyte using `string-to-multibyte`, and inserts the resulting multibyte text.

You can use `set-process-coding-system` to specify which coding system to use (see Section 37.6 [Process Information], page 266). Otherwise, the coding system comes from `coding-system-for-read`, if that is non-`nil`; or else from the defaulting mechanism (see Section 33.9.5 [Default Coding Systems], page 199). If the text output by a process contains null bytes, Emacs by default uses `no-conversion` for it; see Section 33.9.3 [Lisp and Coding Systems], page 195, for how to control this behavior.

Warning: Coding systems such as `undecided`, which determine the coding system from the data, do not work entirely reliably with asynchronous subprocess output. This is because Emacs has to process asynchronous subprocess output in batches, as it arrives. Emacs must try to detect the proper coding system from one batch at a time, and this does not always work. Therefore, if at all possible, specify a coding system that determines both the character code conversion and the end of line conversion—that is, one like `latin-1-unix`, rather than `undecided` or `latin-1`.

When Emacs calls a process filter function, it provides the process output as a multibyte string or as a unibyte string according to the process's filter coding system. Emacs decodes the output according to the process output coding system, which usually produces a multibyte string, except for coding systems such as `binary` and `raw-text`.

37.9.4 Accepting Output from Processes

Output from asynchronous subprocesses normally arrives only while Emacs is waiting for some sort of external event, such as elapsed time or terminal input. Occasionally it is useful in a Lisp program to explicitly permit output to arrive at a specific point, or even to wait until output arrives from a process.

`accept-process-output` **&optional** *process seconds millisec just-this-one* [Function]

This function allows Emacs to read pending output from processes. The output is inserted in the associated buffers or given to their filter functions. If *process* is non-`nil` then this function does not return until some output has been received from *process*.

The arguments *seconds* and *millisec* let you specify timeout periods. The former specifies a period measured in seconds and the latter specifies one measured in milliseconds. The two time periods thus specified are added together, and `accept-process-output` returns after that much time, whether or not there has been any subprocess output.

The argument *millisec* is obsolete (and should not be used), because *seconds* can be a floating point number to specify waiting a fractional number of seconds. If *seconds* is 0, the function accepts whatever output is pending but does not wait.

If *process* is a process, and the argument *just-this-one* is non-`nil`, only output from that process is handled, suspending output from other processes until some output has been received from that process or the timeout expires. If *just-this-one* is an integer, also inhibit running timers. This feature is generally not recommended, but may be necessary for specific applications, such as speech synthesis.

The function `accept-process-output` returns non-`nil` if it did get some output, or `nil` if the timeout expired before output arrived.

37.10 Sentinels: Detecting Process Status Changes

A *process sentinel* is a function that is called whenever the associated process changes status for any reason, including signals (whether sent by Emacs or caused by the process's own actions) that terminate, stop, or continue the process. The process sentinel is also called if the process exits. The sentinel receives two arguments: the process for which the event occurred, and a string describing the type of event.

The string describing the event looks like one of the following:

- `"finished\n"`.
- `"exited abnormally with code` *`exitcode`*`\n"`.
- `"`*`name-of-signal`*`\n"`.
- `"`*`name-of-signal`*` (core dumped)\n"`.

A sentinel runs only while Emacs is waiting (e.g., for terminal input, or for time to elapse, or for process output). This avoids the timing errors that could result from running sentinels at random places in the middle of other Lisp programs. A program can wait, so that sentinels will run, by calling `sit-for` or `sleep-for` (see Section 21.10 [Waiting], page 350, vol. 1), or `accept-process-output` (see Section 37.9.4 [Accepting Output], page 275). Emacs also allows sentinels to run when the command loop is reading input. `delete-process` calls the sentinel when it terminates a running process.

Emacs does not keep a queue of multiple reasons to call the sentinel of one process; it records just the current status and the fact that there has been a change. Therefore two changes in status, coming in quick succession, can call the sentinel just once. However, process termination will always run the sentinel exactly once. This is because the process status can't change again after termination.

Emacs explicitly checks for output from the process before running the process sentinel. Once the sentinel runs due to process termination, no further output can arrive from the process.

A sentinel that writes the output into the buffer of the process should check whether the buffer is still alive. If it tries to insert into a dead buffer, it will get an error. If the buffer is dead, `(buffer-name (process-buffer` *`process`*`))` returns `nil`.

Quitting is normally inhibited within a sentinel—otherwise, the effect of typing `C-g` at command level or to quit a user command would be unpredictable. If you want to permit quitting inside a sentinel, bind `inhibit-quit` to `nil`. In most cases, the right way to do this is with the macro `with-local-quit`. See Section 21.11 [Quitting], page 351, vol. 1.

If an error happens during execution of a sentinel, it is caught automatically, so that it doesn't stop the execution of whatever programs was running when the sentinel was started. However, if `debug-on-error` is non-`nil`, errors are not caught. This makes it possible to use the Lisp debugger to debug the sentinel. See Section 18.1 [Debugger], page 243, vol. 1.

While a sentinel is running, the process sentinel is temporarily set to `nil` so that the sentinel won't run recursively. For this reason it is not possible for a sentinel to specify a new sentinel.

Note that Emacs automatically saves and restores the match data while executing sentinels. See Section 34.6 [Match Data], page 225.

`set-process-sentinel` *process sentinel* [Function]
: This function associates *sentinel* with *process*. If *sentinel* is `nil`, then the process will have no sentinel. The default behavior when there is no sentinel is to insert a message in the process's buffer when the process status changes.

  Changes in process sentinels take effect immediately—if the sentinel is slated to be run but has not been called yet, and you specify a new sentinel, the eventual call to the sentinel will use the new one.

```
(defun msg-me (process event)
   (princ
     (format "Process: %s had the event `%s'" process event)))
(set-process-sentinel (get-process "shell") 'msg-me)
     ⇒ msg-me
(kill-process (get-process "shell"))
     ⊣ Process: #<process shell> had the event `killed'
     ⇒ #<process shell>
```

`process-sentinel` *process* [Function]
: This function returns the sentinel of *process*, or `nil` if it has none.

`waiting-for-user-input-p` [Function]
: While a sentinel or filter function is running, this function returns non-`nil` if Emacs was waiting for keyboard input from the user at the time the sentinel or filter function was called, or `nil` if it was not.

37.11 Querying Before Exit

When Emacs exits, it terminates all its subprocesses by sending them the `SIGHUP` signal. Because subprocesses may be doing valuable work, Emacs normally asks the user to confirm that it is ok to terminate them. Each process has a query flag, which, if non-`nil`, says that Emacs should ask for confirmation before exiting and thus killing that process. The default for the query flag is `t`, meaning *do* query.

`process-query-on-exit-flag` *process* [Function]
: This returns the query flag of *process*.

`set-process-query-on-exit-flag` *process flag* [Function]
This function sets the query flag of *process* to *flag*. It returns *flag*.

Here is an example of using `set-process-query-on-exit-flag` on a shell process to avoid querying:

```
(set-process-query-on-exit-flag (get-process "shell") nil)
     ⇒ nil
```

37.12 Accessing Other Processes

In addition to accessing and manipulating processes that are subprocesses of the current Emacs session, Emacs Lisp programs can also access other processes running on the same machine. We call these *system processes*, to distinguish them from Emacs subprocesses.

Emacs provides several primitives for accessing system processes. Not all platforms support these primitives; on those which don't, these primitives return `nil`.

`list-system-processes` [Function]
This function returns a list of all the processes running on the system. Each process is identified by its PID, a numerical process ID that is assigned by the OS and distinguishes the process from all the other processes running on the same machine at the same time.

`process-attributes` *pid* [Function]
This function returns an alist of attributes for the process specified by its process ID *pid*. Each association in the alist is of the form `(`*`key`*` . `*`value`*`)`, where *key* designates the attribute and *value* is the value of that attribute. The various attribute *key*s that this function can return are listed below. Not all platforms support all of these attributes; if an attribute is not supported, its association will not appear in the returned alist. Values that are numbers can be either integer or floating-point, depending on the magnitude of the value.

`euid` The effective user ID of the user who invoked the process. The corresponding *value* is a number. If the process was invoked by the same user who runs the current Emacs session, the value is identical to what `user-uid` returns (see Section 39.4 [User Identification], page 398).

`user` User name corresponding to the process's effective user ID, a string.

`egid` The group ID of the effective user ID, a number.

`group` Group name corresponding to the effective user's group ID, a string.

`comm` The name of the command that runs in the process. This is a string that usually specifies the name of the executable file of the process, without the leading directories. However, some special system processes can report strings that do not correspond to an executable file of a program.

`state` The state code of the process. This is a short string that encodes the scheduling state of the process. Here's a list of the most frequently seen codes:

`"D"` uninterruptible sleep (usually I/O)

`"R"`	running
`"S"`	interruptible sleep (waiting for some event)
`"T"`	stopped, e.g., by a job control signal
`"Z"`	"zombie": a process that terminated, but was not reaped by its parent

For the full list of the possible states, see the manual page of the `ps` command.

`ppid` The process ID of the parent process, a number.

`pgrp` The process group ID of the process, a number.

`sess` The session ID of the process. This is a number that is the process ID of the process's *session leader*.

`ttname` A string that is the name of the process's controlling terminal. On Unix and GNU systems, this is normally the file name of the corresponding terminal device, such as '`/dev/pts65`'.

`tpgid` The numerical process group ID of the foreground process group that uses the process's terminal.

`minflt` The number of minor page faults caused by the process since its beginning. (Minor page faults are those that don't involve reading from disk.)

`majflt` The number of major page faults caused by the process since its beginning. (Major page faults require a disk to be read, and are thus more expensive than minor page faults.)

`cminflt`
`cmajflt` Like `minflt` and `majflt`, but include the number of page faults for all the child processes of the given process.

`utime` Time spent by the process in the user context, for running the application's code. The corresponding *value* is in the `(`*`high low microsec`*`)` format, the same format used by functions `current-time` (see Section 39.5 [Time of Day], page 399) and `file-attributes` (see Section 25.6.4 [File Attributes], page 475, vol. 1).

`stime` Time spent by the process in the system (kernel) context, for processing system calls. The corresponding *value* is in the same format as for `utime`.

`time` The sum of `utime` and `stime`. The corresponding *value* is in the same format as for `utime`.

`cutime`
`cstime`
`ctime` Like `utime`, `stime`, and `time`, but include the times of all the child processes of the given process.

`pri` The numerical priority of the process.

`nice` The *nice value* of the process, a number. (Processes with smaller nice values get scheduled more favorably.)

`thcount` The number of threads in the process.

`start` The time when the process was started, in the same (*high low microsec*) format used by `current-time` and by `file-attributes`.

`etime` The time elapsed since the process started, in the (*high low microsec*) format.

`vsize` The virtual memory size of the process, measured in kilobytes.

`rss` The size of the process's *resident set*, the number of kilobytes occupied by the process in the machine's physical memory.

`pcpu` The percentage of the CPU time used by the process since it started. The corresponding *value* is a floating-point number between 0 and 100.

`pmem` The percentage of the total physical memory installed on the machine used by the process's resident set. The value is a floating-point number between 0 and 100.

`args` The command-line with which the process was invoked. This is a string in which individual command-line arguments are separated by blanks; whitespace characters that are embedded in the arguments are quoted as appropriate for the system's shell: escaped by backslash characters on GNU and Unix, and enclosed in double quote characters on Windows. Thus, this command-line string can be directly used in primitives such as `shell-command`.

37.13 Transaction Queues

You can use a *transaction queue* to communicate with a subprocess using transactions. First use `tq-create` to create a transaction queue communicating with a specified process. Then you can call `tq-enqueue` to send a transaction.

`tq-create` *process* [Function]

This function creates and returns a transaction queue communicating with *process*. The argument *process* should be a subprocess capable of sending and receiving streams of bytes. It may be a child process, or it may be a TCP connection to a server, possibly on another machine.

`tq-enqueue` *queue question regexp closure fn* **&optional** *delay-question* [Function]

This function sends a transaction to queue *queue*. Specifying the queue has the effect of specifying the subprocess to talk to.

The argument *question* is the outgoing message that starts the transaction. The argument *fn* is the function to call when the corresponding answer comes back; it is called with two arguments: *closure*, and the answer received.

The argument *regexp* is a regular expression that should match text at the end of the entire answer, but nothing before; that's how `tq-enqueue` determines where the answer ends.

If the argument *delay-question* is non-`nil`, delay sending this question until the process has finished replying to any previous questions. This produces more reliable results with some processes.

`tq-close` *queue* [Function]
: Shut down transaction queue *queue*, waiting for all pending transactions to complete, and then terminate the connection or child process.

Transaction queues are implemented by means of a filter function. See Section 37.9.2 [Filter Functions], page 273.

37.14 Network Connections

Emacs Lisp programs can open stream (TCP) and datagram (UDP) network connections (see Section 37.16 [Datagrams], page 284) to other processes on the same machine or other machines. A network connection is handled by Lisp much like a subprocess, and is represented by a process object. However, the process you are communicating with is not a child of the Emacs process, has no process ID, and you can't kill it or send it signals. All you can do is send and receive data. `delete-process` closes the connection, but does not kill the program at the other end; that program must decide what to do about closure of the connection.

Lisp programs can listen for connections by creating network servers. A network server is also represented by a kind of process object, but unlike a network connection, the network server never transfers data itself. When it receives a connection request, it creates a new network connection to represent the connection just made. (The network connection inherits certain information, including the process plist, from the server.) The network server then goes back to listening for more connection requests.

Network connections and servers are created by calling `make-network-process` with an argument list consisting of keyword/argument pairs, for example `:server t` to create a server process, or `:type 'datagram` to create a datagram connection. See Section 37.17 [Low-Level Network], page 284, for details. You can also use the `open-network-stream` function described below.

To distinguish the different types of processes, the `process-type` function returns the symbol `network` for a network connection or server, `serial` for a serial port connection, or `real` for a real subprocess.

The `process-status` function returns `open`, `closed`, `connect`, or `failed` for network connections. For a network server, the status is always `listen`. None of those values is possible for a real subprocess. See Section 37.6 [Process Information], page 266.

You can stop and resume operation of a network process by calling `stop-process` and `continue-process`. For a server process, being stopped means not accepting new connections. (Up to 5 connection requests will be queued for when you resume the server; you can increase this limit, unless it is imposed by the operating system—see the `:server` keyword of `make-network-process`, Section 37.17.1 [Network Processes], page 284.) For a network stream connection, being stopped means not processing input (any arriving input waits until you resume the connection). For a datagram connection, some number of packets may be queued but input may be lost. You can use the function `process-command` to determine whether a network connection or server is stopped; a non-`nil` value means yes.

Emacs can create encrypted network connections, using either built-in or external support. The built-in support uses the GnuTLS ("Transport Layer Security") library; see the GnuTLS project page. If your Emacs was compiled with GnuTLS support, the function `gnutls-available-p` is defined and returns non-`nil`. For more details, see Section

"Overview" in *The Emacs-GnuTLS manual*. The external support uses the '`starttls.el`' library, which requires a helper utility such as `gnutls-cli` to be installed on the system. The `open-network-stream` function can transparently handle the details of creating encrypted connections for you, using whatever support is available.

`open-network-stream` *name buffer host service* **&rest** *parameters* [Function]

This function opens a TCP connection, with optional encryption, and returns a process object that represents the connection.

The *name* argument specifies the name for the process object. It is modified as necessary to make it unique.

The *buffer* argument is the buffer to associate with the connection. Output from the connection is inserted in the buffer, unless you specify a filter function to handle the output. If *buffer* is `nil`, it means that the connection is not associated with any buffer.

The arguments *host* and *service* specify where to connect to; *host* is the host name (a string), and *service* is the name of a defined network service (a string) or a port number (an integer).

The remaining arguments *parameters* are keyword/argument pairs that are mainly relevant to encrypted connections:

`:nowait` *boolean*
: If non-`nil`, try to make an asynchronous connection.

`:type` *type*
: The type of connection. Options are:

  `plain` An ordinary, unencrypted connection.

  `tls`
  `ssl` A TLS ("Transport Layer Security") connection.

  `nil`
  `network` Start with a plain connection, and if parameters '`:success`' and '`:capability-command`' are supplied, try to upgrade to an encrypted connection via STARTTLS. If that fails, retain the unencrypted connection.

  `starttls` As for `nil`, but if STARTTLS fails drop the connection.

  `shell` A shell connection.

`:always-query-capabilities` *boolean*
: If non-`nil`, always ask for the server's capabilities, even when doing a '`plain`' connection.

`:capability-command` *capability-command*
: Command string to query the host capabilities.

`:end-of-command` *regexp*
`:end-of-capability` *regexp*
: Regular expression matching the end of a command, or the end of the command *capability-command*. The latter defaults to the former.

`:starttls-function` *function*
: Function of one argument (the response to *capability-command*), which returns either `nil`, or the command to activate STARTTLS if supported.

`:success` *regexp*
: Regular expression matching a successful STARTTLS negotiation.

`:use-starttls-if-possible` *boolean*
: If non-`nil`, do opportunistic STARTTLS upgrades even if Emacs doesn't have built-in TLS support.

`:client-certificate` *list-or-t*
: Either a list of the form `(`*key-file cert-file*`)`, naming the certificate key file and certificate file itself, or `t`, meaning to query `auth-source` for this information (see Section "Overview" in *The Auth-Source Manual*). Only used for TLS or STARTTLS.

`:return-list` *cons-or-nil*
: The return value of this function. If omitted or `nil`, return a process object. Otherwise, a cons of the form `(`*process-object* `.` *plist*`)`, where *plist* has keywords:

  `:greeting` *string-or-nil*
  : If non-`nil`, the greeting string returned by the host.

  `:capabilities` *string-or-nil*
  : If non-`nil`, the host's capability string.

  `:type` *symbol*
  : The connection type: '`plain`' or '`tls`'.

37.15 Network Servers

You create a server by calling `make-network-process` (see Section 37.17.1 [Network Processes], page 284) with `:server t`. The server will listen for connection requests from clients. When it accepts a client connection request, that creates a new network connection, itself a process object, with the following parameters:

- The connection's process name is constructed by concatenating the server process's *name* with a client identification string. The client identification string for an IPv4 connection looks like '`<`*a.b.c.d*`:`*p*`>`', which represents an address and port number. Otherwise, it is a unique number in brackets, as in '`<`*nnn*`>`'. The number is unique for each connection in the Emacs session.
- If the server's filter is non-`nil`, the connection process does not get a separate process buffer; otherwise, Emacs creates a new buffer for the purpose. The buffer name is the server's buffer name or process name, concatenated with the client identification string.

  The server's process buffer value is never used directly, but the log function can retrieve it and use it to log connections by inserting text there.
- The communication type and the process filter and sentinel are inherited from those of the server. The server never directly uses its filter and sentinel; their sole purpose is to initialize connections made to the server.

- The connection's process contact information is set according to the client's addressing information (typically an IP address and a port number). This information is associated with the `process-contact` keywords `:host`, `:service`, `:remote`.
- The connection's local address is set up according to the port number used for the connection.
- The client process's plist is initialized from the server's plist.

37.16 Datagrams

A *datagram* connection communicates with individual packets rather than streams of data. Each call to `process-send` sends one datagram packet (see Section 37.7 [Input to Processes], page 269), and each datagram received results in one call to the filter function.

The datagram connection doesn't have to talk with the same remote peer all the time. It has a *remote peer address* which specifies where to send datagrams to. Each time an incoming datagram is passed to the filter function, the peer address is set to the address that datagram came from; that way, if the filter function sends a datagram, it will go back to that place. You can specify the remote peer address when you create the datagram connection using the `:remote` keyword. You can change it later on by calling `set-process-datagram-address`.

`process-datagram-address` *process* [Function]
: If *process* is a datagram connection or server, this function returns its remote peer address.

`set-process-datagram-address` *process address* [Function]
: If *process* is a datagram connection or server, this function sets its remote peer address to *address*.

37.17 Low-Level Network Access

You can also create network connections by operating at a lower level than that of `open-network-stream`, using `make-network-process`.

37.17.1 `make-network-process`

The basic function for creating network connections and network servers is `make-network-process`. It can do either of those jobs, depending on the arguments you give it.

`make-network-process` **&rest** *args* [Function]
: This function creates a network connection or server and returns the process object that represents it. The arguments *args* are a list of keyword/argument pairs. Omitting a keyword is always equivalent to specifying it with value `nil`, except for `:coding`, `:filter-multibyte`, and `:reuseaddr`. Here are the meaningful keywords (those corresponding to network options are listed in the following section):

  :name *name*
  : Use the string *name* as the process name. It is modified if necessary to make it unique.

:type *type* Specify the communication type. A value of `nil` specifies a stream connection (the default); `datagram` specifies a datagram connection; `seqpacket` specifies a "sequenced packet stream" connection. Both connections and servers can be of these types.

:server *server-flag*
If *server-flag* is non-`nil`, create a server. Otherwise, create a connection. For a stream type server, *server-flag* may be an integer, which then specifies the length of the queue of pending connections to the server. The default queue length is 5.

:host *host* Specify the host to connect to. *host* should be a host name or Internet address, as a string, or the symbol `local` to specify the local host. If you specify *host* for a server, it must specify a valid address for the local host, and only clients connecting to that address will be accepted.

:service *service*
service specifies a port number to connect to; or, for a server, the port number to listen on. It should be a service name that translates to a port number, or an integer specifying the port number directly. For a server, it can also be `t`, which means to let the system select an unused port number.

:family *family*
family specifies the address (and protocol) family for communication. `nil` means determine the proper address family automatically for the given *host* and *service*. `local` specifies a Unix socket, in which case *host* is ignored. `ipv4` and `ipv6` specify to use IPv4 and IPv6, respectively.

:local *local-address*
For a server process, *local-address* is the address to listen on. It overrides *family*, *host* and *service*, so you might as well not specify them.

:remote *remote-address*
For a connection, *remote-address* is the address to connect to. It overrides *family*, *host* and *service*, so you might as well not specify them.

For a datagram server, *remote-address* specifies the initial setting of the remote datagram address.

The format of *local-address* or *remote-address* depends on the address family:

- An IPv4 address is represented as a five-element vector of four 8-bit integers and one 16-bit integer `[a b c d p]` corresponding to numeric IPv4 address *a.b.c.d* and port number *p*.
- An IPv6 address is represented as a nine-element vector of 16-bit integers `[a b c d e f g h p]` corresponding to numeric IPv6 address *a*:*b*:*c*:*d*:*e*:*f*:*g*:*h* and port number *p*.
- A local address is represented as a string, which specifies the address in the local address space.

- An "unsupported family" address is represented by a cons `(f . av)`, where *f* is the family number and *av* is a vector specifying the socket address using one element per address data byte. Do not rely on this format in portable code, as it may depend on implementation defined constants, data sizes, and data structure alignment.

:nowait *bool*
: If *bool* is non-`nil` for a stream connection, return without waiting for the connection to complete. When the connection succeeds or fails, Emacs will call the sentinel function, with a second argument matching `"open"` (if successful) or `"failed"`. The default is to block, so that `make-network-process` does not return until the connection has succeeded or failed.

:stop *stopped*
: If *stopped* is non-`nil`, start the network connection or server in the "stopped" state.

:buffer *buffer*
: Use *buffer* as the process buffer.

:coding *coding*
: Use *coding* as the coding system for this process. To specify different coding systems for decoding data from the connection and for encoding data sent to it, specify `(decoding . encoding)` for *coding*.

  If you don't specify this keyword at all, the default is to determine the coding systems from the data.

:noquery *query-flag*
: Initialize the process query flag to *query-flag*. See Section 37.11 [Query Before Exit], page 277.

:filter *filter*
: Initialize the process filter to *filter*.

:filter-multibyte *multibyte*
: If *multibyte* is non-`nil`, strings given to the process filter are multibyte, otherwise they are unibyte. The default is the default value of `enable-multibyte-characters`.

:sentinel *sentinel*
: Initialize the process sentinel to *sentinel*.

:log *log*
: Initialize the log function of a server process to *log*. The log function is called each time the server accepts a network connection from a client. The arguments passed to the log function are *server*, *connection*, and *message*; where *server* is the server process, *connection* is the new process for the connection, and *message* is a string describing what has happened.

:plist *plist*
: Initialize the process plist to *plist*.

The original argument list, modified with the actual connection information, is available via the `process-contact` function.

37.17.2 Network Options

The following network options can be specified when you create a network process. Except for `:reuseaddr`, you can also set or modify these options later, using `set-network-process-option`.

For a server process, the options specified with `make-network-process` are not inherited by the client connections, so you will need to set the necessary options for each child connection as it is created.

:bindtodevice *device-name*
: If *device-name* is a non-empty string identifying a network interface name (see `network-interface-list`), only handle packets received on that interface. If *device-name* is `nil` (the default), handle packets received on any interface.

  Using this option may require special privileges on some systems.

:broadcast *broadcast-flag*
: If *broadcast-flag* is non-`nil` for a datagram process, the process will receive datagram packet sent to a broadcast address, and be able to send packets to a broadcast address. This is ignored for a stream connection.

:dontroute *dontroute-flag*
: If *dontroute-flag* is non-`nil`, the process can only send to hosts on the same network as the local host.

:keepalive *keepalive-flag*
: If *keepalive-flag* is non-`nil` for a stream connection, enable exchange of low-level keep-alive messages.

:linger *linger-arg*
: If *linger-arg* is non-`nil`, wait for successful transmission of all queued packets on the connection before it is deleted (see `delete-process`). If *linger-arg* is an integer, it specifies the maximum time in seconds to wait for queued packets to be sent before closing the connection. The default is `nil`, which means to discard unsent queued packets when the process is deleted.

:oobinline *oobinline-flag*
: If *oobinline-flag* is non-`nil` for a stream connection, receive out-of-band data in the normal data stream. Otherwise, ignore out-of-band data.

:priority *priority*
: Set the priority for packets sent on this connection to the integer *priority*. The interpretation of this number is protocol specific; such as setting the TOS (type of service) field on IP packets sent on this connection. It may also have system dependent effects, such as selecting a specific output queue on the network interface.

:reuseaddr *reuseaddr-flag*
: If *reuseaddr-flag* is non-`nil` (the default) for a stream server process, allow this server to reuse a specific port number (see `:service`), unless another process on this host is already listening on that port. If *reuseaddr-flag* is `nil`, there may be a period of time after the last use of that port (by any process on the host) where it is not possible to make a new server on that port.

`set-network-process-option` *process option value* **&optional** *no-error* [Function]
: This function sets or modifies a network option for network process *process*. The accepted options and values are as for `make-network-process`. If *no-error* is non-`nil`, this function returns `nil` instead of signaling an error if *option* is not a supported option. If the function successfully completes, it returns `t`.

  The current setting of an option is available via the `process-contact` function.

37.17.3 Testing Availability of Network Features

To test for the availability of a given network feature, use `featurep` like this:

```
(featurep 'make-network-process '(keyword value))
```

The result of this form is `t` if it works to specify *keyword* with value *value* in `make-network-process`. Here are some of the *keyword—value* pairs you can test in this way.

`(:nowait t)`
: Non-`nil` if non-blocking connect is supported.

`(:type datagram)`
: Non-`nil` if datagrams are supported.

`(:family local)`
: Non-`nil` if local (a.k.a. "UNIX domain") sockets are supported.

`(:family ipv6)`
: Non-`nil` if IPv6 is supported.

`(:service t)`
: Non-`nil` if the system can select the port for a server.

To test for the availability of a given network option, use `featurep` like this:

```
(featurep 'make-network-process 'keyword)
```

The accepted *keyword* values are `:bindtodevice`, etc. For the complete list, see Section 37.17.2 [Network Options], page 287. This form returns non-`nil` if that particular network option is supported by `make-network-process` (or `set-network-process-option`).

37.18 Misc Network Facilities

These additional functions are useful for creating and operating on network connections. Note that they are supported only on some systems.

`network-interface-list` [Function]
: This function returns a list describing the network interfaces of the machine you are using. The value is an alist whose elements have the form `(name . address)`. *address* has the same form as the *local-address* and *remote-address* arguments to `make-network-process`.

`network-interface-info` *ifname* [Function]
: This function returns information about the network interface named *ifname*. The value is a list of the form `(addr bcast netmask hwaddr flags)`.

  addr The Internet protocol address.

bcast The broadcast address.

netmask The network mask.

hwaddr The layer 2 address (Ethernet MAC address, for instance).

flags The current flags of the interface.

`format-network-address` *address* **&optional** *omit-port* [Function]

This function converts the Lisp representation of a network address to a string.

A five-element vector `[a b c d p]` represents an IPv4 address *a.b.c.d* and port number *p*. `format-network-address` converts that to the string `"a.b.c.d:p"`.

A nine-element vector `[a b c d e f g h p]` represents an IPv6 address along with a port number. `format-network-address` converts that to the string `"[a:b:c:d:e:f:g:h]:p"`.

If the vector does not include the port number, *p*, or if *omit-port* is non-`nil`, the result does not include the `:p` suffix.

37.19 Communicating with Serial Ports

Emacs can communicate with serial ports. For interactive use, `M-x serial-term` opens a terminal window. In a Lisp program, `make-serial-process` creates a process object.

The serial port can be configured at run-time, without having to close and re-open it. The function `serial-process-configure` lets you change the speed, bytesize, and other parameters. In a terminal window created by `serial-term`, you can click on the mode line for configuration.

A serial connection is represented by a process object, which can be used in a similar way to a subprocess or network process. You can send and receive data, and configure the serial port. A serial process object has no process ID, however, and you can't send signals to it, and the status codes are different from other types of processes. `delete-process` on the process object or `kill-buffer` on the process buffer close the connection, but this does not affect the device connected to the serial port.

The function `process-type` returns the symbol `serial` for a process object representing a serial port connection.

Serial ports are available on GNU/Linux, Unix, and MS Windows systems.

`serial-term` *port speed* [Command]

Start a terminal-emulator for a serial port in a new buffer. *port* is the name of the serial port to connect to. For example, this could be '`/dev/ttyS0`' on Unix. On MS Windows, this could be '`COM1`', or '`\\.\COM10`' (double the backslashes in Lisp strings).

speed is the speed of the serial port in bits per second. 9600 is a common value. The buffer is in Term mode; see Section "Term Mode" in *The GNU Emacs Manual*, for the commands to use in that buffer. You can change the speed and the configuration in the mode line menu.

`make-serial-process` **&rest** *args* [Function]

This function creates a process and a buffer. Arguments are specified as keyword/argument pairs. Here's the list of the meaningful keywords, with the first two (*port* and *speed*) being mandatory:

`:port` *port*
: This is the name of the serial port. On Unix and GNU systems, this is a file name such as '`/dev/ttyS0`'. On Windows, this could be '`COM1`', or '`\\.\COM10`' for ports higher than '`COM9`' (double the backslashes in Lisp strings).

`:speed` *speed*
: The speed of the serial port in bits per second. This function calls `serial-process-configure` to handle the speed; see the following documentation of that function for more details.

`:name` *name*
: The name of the process. If *name* is not given, *port* will serve as the process name as well.

`:buffer` *buffer*
: The buffer to associate with the process. The value can be either a buffer or a string that names a buffer. Process output goes at the end of that buffer, unless you specify an output stream or filter function to handle the output. If *buffer* is not given, the process buffer's name is taken from the value of the `:name` keyword.

`:coding` *coding*
: If *coding* is a symbol, it specifies the coding system used for both reading and writing for this process. If *coding* is a cons (*decoding* . *encoding*), *decoding* is used for reading, and *encoding* is used for writing. If not specified, the default is to determine the coding systems from the data itself.

`:noquery` *query-flag*
: Initialize the process query flag to *query-flag*. See Section 37.11 [Query Before Exit], page 277. The flags defaults to `nil` if unspecified.

`:stop` *bool*
: Start process in the "stopped" state if *bool* is non-`nil`. In the stopped state, a serial process does not accept incoming data, but you can send outgoing data. The stopped state is cleared by `continue-process` and set by `stop-process`.

`:filter` *filter*
: Install *filter* as the process filter.

`:sentinel` *sentinel*
: Install *sentinel* as the process sentinel.

`:plist` *plist*
: Install *plist* as the initial plist of the process.

`:bytesize`
`:parity`
`:stopbits`
`:flowcontrol`
: These are handled by `serial-process-configure`, which is called by `make-serial-process`.

The original argument list, possibly modified by later configuration, is available via the function `process-contact`.

Here is an example:

```
(make-serial-process :port "/dev/ttyS0" :speed 9600)
```

`serial-process-configure` **&rest** *args* [Function]

This functions configures a serial port connection. Arguments are specified as keyword/argument pairs. Attributes that are not given are re-initialized from the process's current configuration (available via the function `process-contact`), or set to reasonable default values. The following arguments are defined:

`:process` *process*
`:name` *name*
`:buffer` *buffer*
`:port` *port*
: Any of these arguments can be given to identify the process that is to be configured. If none of these arguments is given, the current buffer's process is used.

`:speed` *speed*
: The speed of the serial port in bits per second, a.k.a. *baud rate*. The value can be any number, but most serial ports work only at a few defined values between 1200 and 115200, with 9600 being the most common value. If *speed* is `nil`, the function ignores all other arguments and does not configure the port. This may be useful for special serial ports such as Bluetooth-to-serial converters, which can only be configured through '`AT`' commands sent through the connection. The value of `nil` for *speed* is valid only for connections that were already opened by a previous call to `make-serial-process` or `serial-term`.

`:bytesize` *bytesize*
: The number of bits per byte, which can be 7 or 8. If *bytesize* is not given or `nil`, it defaults to 8.

`:parity` *parity*
: The value can be `nil` (don't use parity), the symbol `odd` (use odd parity), or the symbol `even` (use even parity). If *parity* is not given, it defaults to no parity.

`:stopbits` *stopbits*
: The number of stopbits used to terminate a transmission of each byte. *stopbits* can be 1 or 2. If *stopbits* is not given or `nil`, it defaults to 1.

`:flowcontrol` *flowcontrol*
: The type of flow control to use for this connection, which is either `nil` (don't use flow control), the symbol `hw` (use RTS/CTS hardware flow control), or the symbol `sw` (use XON/XOFF software flow control). If *flowcontrol* is not given, it defaults to no flow control.

Internally, `make-serial-process` calls `serial-process-configure` for the initial configuration of the serial port.

37.20 Packing and Unpacking Byte Arrays

This section describes how to pack and unpack arrays of bytes, usually for binary network protocols. These functions convert byte arrays to alists, and vice versa. The byte array can be represented as a unibyte string or as a vector of integers, while the alist associates symbols either with fixed-size objects or with recursive sub-alists. To use the functions referred to in this section, load the `bindat` library.

Conversion from byte arrays to nested alists is also known as *deserializing* or *unpacking*, while going in the opposite direction is also known as *serializing* or *packing*.

37.20.1 Describing Data Layout

To control unpacking and packing, you write a *data layout specification*, a special nested list describing named and typed *fields*. This specification controls the length of each field to be processed, and how to pack or unpack it. We normally keep bindat specs in variables whose names end in '`-bindat-spec`'; that kind of name is automatically recognized as "risky".

A field's *type* describes the size (in bytes) of the object that the field represents and, in the case of multibyte fields, how the bytes are ordered within the field. The two possible orderings are "big endian" (also known as "network byte ordering") and "little endian". For instance, the number `#x23cd` (decimal 9165) in big endian would be the two bytes `#x23` `#xcd`; and in little endian, `#xcd` `#x23`. Here are the possible type values:

`u8`
`byte`
: Unsigned byte, with length 1.

`u16`
`word`
`short`
: Unsigned integer in network byte order, with length 2.

`u24`
: Unsigned integer in network byte order, with length 3.

`u32`
`dword`
`long`
: Unsigned integer in network byte order, with length 4. Note: These values may be limited by Emacs's integer implementation limits.

`u16r`
`u24r`
`u32r`
: Unsigned integer in little endian order, with length 2, 3 and 4, respectively.

`str` *len*
: String of length *len*.

`strz` *len*
: Zero-terminated string, in a fixed-size field with length *len*.

`vec` *len* `[`*type*`]`
: Vector of *len* elements of type *type*, defaulting to bytes. The *type* is any of the simple types above, or another vector specified as a list of the form `(vec` *len* `[`*type*`])`.

`ip`
: Four-byte vector representing an Internet address. For example: `[127 0 0 1]` for localhost.

`bits` *len*
: List of set bits in *len* bytes. The bytes are taken in big endian order and the bits are numbered starting with `8 *` *len* `- 1` and ending with zero. For example: `bits 2` unpacks `#x28 #x1c` to `(2 3 4 11 13)` and `#x1c #x28` to `(3 5 10 11 12)`.

`(eval` *form*`)`
: *form* is a Lisp expression evaluated at the moment the field is unpacked or packed. The result of the evaluation should be one of the above-listed type specifications.

For a fixed-size field, the length *len* is given as an integer specifying the number of bytes in the field.

When the length of a field is not fixed, it typically depends on the value of a preceding field. In this case, the length *len* can be given either as a list `(`*name* `...)` identifying a *field name* in the format specified for `bindat-get-field` below, or by an expression `(eval` *form*`)` where *form* should evaluate to an integer, specifying the field length.

A field specification generally has the form `([`*name*`]` *handler*`)`, where *name* is optional. Don't use names that are symbols meaningful as type specifications (above) or handler specifications (below), since that would be ambiguous. *name* can be a symbol or an expression `(eval` *form*`)`, in which case *form* should evaluate to a symbol.

handler describes how to unpack or pack the field and can be one of the following:

type
: Unpack/pack this field according to the type specification *type*.

`eval` *form*
: Evaluate *form*, a Lisp expression, for side-effect only. If the field name is specified, the value is bound to that field name.

`fill` *len*
: Skip *len* bytes. In packing, this leaves them unchanged, which normally means they remain zero. In unpacking, this means they are ignored.

`align` *len*
: Skip to the next multiple of *len* bytes.

`struct` *spec-name*
: Process *spec-name* as a sub-specification. This describes a structure nested within another structure.

`union` *form* `(`*tag* *spec*`)...`
: Evaluate *form*, a Lisp expression, find the first *tag* that matches it, and process its associated data layout specification *spec*. Matching can occur in one of three ways:

  - If a *tag* has the form `(eval` *expr*`)`, evaluate *expr* with the variable `tag` dynamically bound to the value of *form*. A non-`nil` result indicates a match.
  - *tag* matches if it is `equal` to the value of *form*.

- *tag* matches unconditionally if it is `t`.

`repeat` *`count field-specs...`*
: Process the *field-specs* recursively, in order, then repeat starting from the first one, processing all the specifications *count* times overall. The *count* is given using the same formats as a field length—if an `eval` form is used, it is evaluated just once. For correct operation, each specification in *field-specs* must include a name.

For the `(eval form)` forms used in a bindat specification, the *form* can access and update these dynamically bound variables during evaluation:

`last`
: Value of the last field processed.

`bindat-raw`
: The data as a byte array.

`bindat-idx`
: Current index (within `bindat-raw`) for unpacking or packing.

`struct`
: The alist containing the structured data that have been unpacked so far, or the entire structure being packed. You can use `bindat-get-field` to access specific fields of this structure.

`count`
`index`
: Inside a `repeat` block, these contain the maximum number of repetitions (as specified by the *count* parameter), and the current repetition number (counting from 0). Setting `count` to zero will terminate the inner-most repeat block after the current repetition has completed.

37.20.2 Functions to Unpack and Pack Bytes

In the following documentation, *spec* refers to a data layout specification, `bindat-raw` to a byte array, and *struct* to an alist representing unpacked field data.

`bindat-unpack` *spec bindat-raw* **&optional** *bindat-idx* [Function]
: This function unpacks data from the unibyte string or byte array `bindat-raw` according to *spec*. Normally, this starts unpacking at the beginning of the byte array, but if *bindat-idx* is non-`nil`, it specifies a zero-based starting position to use instead.

  The value is an alist or nested alist in which each element describes one unpacked field.

`bindat-get-field` *struct* **&rest** *name* [Function]
: This function selects a field's data from the nested alist *struct*. Usually *struct* was returned by `bindat-unpack`. If *name* corresponds to just one argument, that means to extract a top-level field value. Multiple *name* arguments specify repeated lookup of sub-structures. An integer name acts as an array index.

  For example, if *name* is `(a b 2 c)`, that means to find field `c` in the third element of subfield `b` of field `a`. (This corresponds to `struct.a.b[2].c` in C.)

Although packing and unpacking operations change the organization of data (in memory), they preserve the data's *total length*, which is the sum of all the fields' lengths, in

bytes. This value is not generally inherent in either the specification or alist alone; instead, both pieces of information contribute to its calculation. Likewise, the length of a string or array being unpacked may be longer than the data's total length as described by the specification.

`bindat-length` *spec struct* [Function]
This function returns the total length of the data in *struct*, according to *spec*.

`bindat-pack` *spec struct* **&optional** *bindat-raw bindat-idx* [Function]
This function returns a byte array packed according to *spec* from the data in the alist *struct*. It normally creates and fills a new byte array starting at the beginning. However, if *bindat-raw* is non-`nil`, it specifies a pre-allocated unibyte string or vector to pack into. If *bindat-idx* is non-`nil`, it specifies the starting offset for packing into `bindat-raw`.

When pre-allocating, you should make sure `(length bindat-raw)` meets or exceeds the total length to avoid an out-of-range error.

`bindat-ip-to-string` *ip* [Function]
Convert the Internet address vector *ip* to a string in the usual dotted notation.

```
(bindat-ip-to-string [127 0 0 1])
     ⇒ "127.0.0.1"
```

37.20.3 Examples of Byte Unpacking and Packing

Here is a complete example of byte unpacking and packing:

```
(require 'bindat)

(defvar fcookie-index-spec
  '((:version  u32)
    (:count    u32)
    (:longest  u32)
    (:shortest u32)
    (:flags    u32)
    (:delim    u8)
    (:ignored  fill 3)
    (:offset   repeat (:count) (:foo u32)))
  "Description of a fortune cookie index file's contents.")

(defun fcookie (cookies &optional index)
  "Display a random fortune cookie from file COOKIES.
Optional second arg INDEX specifies the associated index
filename, by default \"COOKIES.dat\".  Display cookie text
in buffer \"*Fortune Cookie: BASENAME*\", where BASENAME
is COOKIES without the directory part."
  (interactive "fCookies file: ")
  (let* ((info (with-temp-buffer
                 (insert-file-contents-literally
                  (or index (concat cookies ".dat")))
```

```
                (bindat-unpack fcookie-index-spec
                               (buffer-string))))
          (sel (random (bindat-get-field info :count)))
          (beg (cdar (bindat-get-field info :offset sel)))
          (end (or (cdar (bindat-get-field info
                                           :offset (1+ sel)))
                   (nth 7 (file-attributes cookies)))))
    (switch-to-buffer
     (get-buffer-create
      (format "*Fortune Cookie: %s*"
              (file-name-nondirectory cookies))))
    (erase-buffer)
    (insert-file-contents-literally
     cookies nil beg (- end 3))))

(defun fcookie-create-index (cookies &optional index delim)
  "Scan file COOKIES, and write out its index file.
Optional arg INDEX specifies the index filename, which by
default is \"COOKIES.dat\".  Optional arg DELIM specifies the
unibyte character that, when found on a line of its own in
COOKIES, indicates the border between entries."
  (interactive "fCookies file: ")
  (setq delim (or delim ?%))
  (let ((delim-line (format "\n%c\n" delim))
        (count 0)
        (max 0)
        min p q len offsets)
    (unless (= 3 (string-bytes delim-line))
      (error "Delimiter cannot be represented in one byte"))
    (with-temp-buffer
      (insert-file-contents-literally cookies)
      (while (and (setq p (point))
                  (search-forward delim-line (point-max) t)
                  (setq len (- (point) 3 p)))
        (setq count (1+ count)
              max (max max len)
              min (min (or min max) len)
              offsets (cons (1- p) offsets))))
    (with-temp-buffer
      (set-buffer-multibyte nil)
      (insert
       (bindat-pack
        fcookie-index-spec
        `((:version . 2)
          (:count . ,count)
          (:longest . ,max)
          (:shortest . ,min)
```

```
          (:flags . 0)
          (:delim . ,delim)
          (:offset . ,(mapcar (lambda (o)
                                (list (cons :foo o)))
                              (nreverse offsets))))))
      (let ((coding-system-for-write 'raw-text-unix))
        (write-file (or index (concat cookies ".dat")))))))
```

The following is an example of defining and unpacking a complex structure. Consider the following C structures:

```
struct header {
    unsigned long    dest_ip;
    unsigned long    src_ip;
    unsigned short   dest_port;
    unsigned short   src_port;
};

struct data {
    unsigned char    type;
    unsigned char    opcode;
    unsigned short   length;  /* in network byte order  */
    unsigned char    id[8];   /* null-terminated string  */
    unsigned char    data[/* (length + 3) & ~3 */];
};

struct packet {
    struct header    header;
    unsigned long    counters[2];  /* in little endian order  */
    unsigned char    items;
    unsigned char    filler[3];
    struct data      item[/* items */];

};
```

The corresponding data layout specification is:

```
(setq header-spec
      '((dest-ip   ip)
        (src-ip    ip)
        (dest-port u16)
        (src-port  u16)))

(setq data-spec
      '((type      u8)
        (opcode    u8)
        (length    u16)  ; network byte order
        (id        strz 8)
        (data      vec (length))
        (align     4)))
```

```
(setq packet-spec
      '((header    struct header-spec)
        (counters  vec 2 u32r)   ; little endian order
        (items     u8)
        (fill      3)
        (item      repeat (items)
                   (struct data-spec))))
```

A binary data representation is:

```
(setq binary-data
      [ 192 168 1 100 192 168 1 101 01 28 21 32
        160 134 1 0 5 1 0 0 2 0 0 0
        2 3 0 5 ?A ?B ?C ?D ?E ?F 0 0 1 2 3 4 5 0 0 0
        1 4 0 7 ?B ?C ?D ?E ?F ?G 0 0 6 7 8 9 10 11 12 0 ])
```

The corresponding decoded structure is:

```
(setq decoded (bindat-unpack packet-spec binary-data))
     ⇒
((header
  (dest-ip   . [192 168 1 100])
  (src-ip    . [192 168 1 101])
  (dest-port . 284)
  (src-port  . 5408))
 (counters . [100000 261])
 (items . 2)
 (item ((data . [1 2 3 4 5])
        (id . "ABCDEF")
        (length . 5)
        (opcode . 3)
        (type . 2))
       ((data . [6 7 8 9 10 11 12])
        (id . "BCDEFG")
        (length . 7)
        (opcode . 4)
        (type . 1))))
```

An example of fetching data from this structure:

```
(bindat-get-field decoded 'item 1 'id)
     ⇒ "BCDEFG"
```

38 Emacs Display

This chapter describes a number of features related to the display that Emacs presents to the user.

38.1 Refreshing the Screen

The function `redraw-frame` clears and redisplays the entire contents of a given frame (see Chapter 29 [Frames], page 66). This is useful if the screen is corrupted.

`redraw-frame` *frame* [Function]
: This function clears and redisplays frame *frame*.

Even more powerful is `redraw-display`:

`redraw-display` [Command]
: This function clears and redisplays all visible frames.

In Emacs, processing user input takes priority over redisplay. If you call these functions when input is available, they don't redisplay immediately, but the requested redisplay does happen eventually—after all the input has been processed.

On text terminals, suspending and resuming Emacs normally also refreshes the screen. Some terminal emulators record separate contents for display-oriented programs such as Emacs and for ordinary sequential display. If you are using such a terminal, you might want to inhibit the redisplay on resumption.

`no-redraw-on-reenter` [User Option]
: This variable controls whether Emacs redraws the entire screen after it has been suspended and resumed. Non-`nil` means there is no need to redraw, `nil` means redrawing is needed. The default is `nil`.

38.2 Forcing Redisplay

Emacs normally tries to redisplay the screen whenever it waits for input. With the following function, you can request an immediate attempt to redisplay, in the middle of Lisp code, without actually waiting for input.

`redisplay` **&optional** *force* [Function]
: This function tries immediately to redisplay. The optional argument *force*, if non-`nil`, forces the redisplay to be performed, instead of being preempted, even if input is pending and the variable `redisplay-dont-pause` is `nil` (see below). If `redisplay-dont-pause` is non-`nil` (the default), this function redisplays in any case, i.e. *force* does nothing.

  The function returns `t` if it actually tried to redisplay, and `nil` otherwise. A value of `t` does not mean that redisplay proceeded to completion; it could have been preempted by newly arriving input.

`redisplay-dont-pause` [Variable]
: If this variable is `nil`, arriving input events preempt redisplay; Emacs avoids starting a redisplay, and stops any redisplay that is in progress, until the input has been

processed. In particular, `(redisplay)` returns `nil` without actually redisplaying, if there is pending input.

The default value is `t`, which means that pending input does not preempt redisplay.

`redisplay-preemption-period` [Variable]

If `redisplay-dont-pause` is `nil`, this variable specifies how many seconds Emacs waits between checks for new input during redisplay; if input arrives during this interval, redisplay stops and the input is processed. The default value is 0.1; if the value is `nil`, Emacs does not check for input during redisplay.

This variable has no effect when `redisplay-dont-pause` is non-`nil` (the default).

Although `redisplay` tries immediately to redisplay, it does not change how Emacs decides which parts of its frame(s) to redisplay. By contrast, the following function adds certain windows to the pending redisplay work (as if their contents had completely changed), but does not immediately try to perform redisplay.

`force-window-update` **&optional** *object* [Function]

This function forces some or all windows to be updated the next time Emacs does a redisplay. If *object* is a window, that window is to be updated. If *object* is a buffer or buffer name, all windows displaying that buffer are to be updated. If *object* is `nil` (or omitted), all windows are to be updated.

This function does not do a redisplay immediately; Emacs does that as it waits for input, or when the function `redisplay` is called.

38.3 Truncation

When a line of text extends beyond the right edge of a window, Emacs can *continue* the line (make it "wrap" to the next screen line), or *truncate* the line (limit it to one screen line). The additional screen lines used to display a long text line are called *continuation* lines. Continuation is not the same as filling; continuation happens on the screen only, not in the buffer contents, and it breaks a line precisely at the right margin, not at a word boundary. See Section 32.11 [Filling], page 140.

On a graphical display, tiny arrow images in the window fringes indicate truncated and continued lines (see Section 38.13 [Fringes], page 344). On a text terminal, a '`$`' in the rightmost column of the window indicates truncation; a '`\`' on the rightmost column indicates a line that "wraps". (The display table can specify alternate characters to use for this; see Section 38.20.2 [Display Tables], page 377).

`truncate-lines` [User Option]

If this buffer-local variable is non-`nil`, lines that extend beyond the right edge of the window are truncated; otherwise, they are continued. As a special exception, the variable `truncate-partial-width-windows` takes precedence in *partial-width* windows (i.e. windows that do not occupy the entire frame width).

`truncate-partial-width-windows` [User Option]

This variable controls line truncation in *partial-width* windows. A partial-width window is one that does not occupy the entire frame width (see Section 28.5 [Splitting

Windows], page 26). If the value is `nil`, line truncation is determined by the variable `truncate-lines` (see above). If the value is an integer *n*, lines are truncated if the partial-width window has fewer than *n* columns, regardless of the value of `truncate-lines`; if the partial-width window has *n* or more columns, line truncation is determined by `truncate-lines`. For any other non-`nil` value, lines are truncated in every partial-width window, regardless of the value of `truncate-lines`.

When horizontal scrolling (see Section 28.21 [Horizontal Scrolling], page 56) is in use in a window, that forces truncation.

`wrap-prefix` [Variable]

If this buffer-local variable is non-`nil`, it defines a *wrap prefix* which Emacs displays at the start of every continuation line. (If lines are truncated, `wrap-prefix` is never used.) Its value may be a string or an image (see Section 38.15.4 [Other Display Specs], page 353), or a stretch of whitespace such as specified by the `:width` or `:align-to` display properties (see Section 38.15.2 [Specified Space], page 351). The value is interpreted in the same way as a `display` text property. See Section 38.15 [Display Property], page 350.

A wrap prefix may also be specified for regions of text, using the `wrap-prefix` text or overlay property. This takes precedence over the `wrap-prefix` variable. See Section 32.19.4 [Special Properties], page 162.

`line-prefix` [Variable]

If this buffer-local variable is non-`nil`, it defines a *line prefix* which Emacs displays at the start of every non-continuation line. Its value may be a string or an image (see Section 38.15.4 [Other Display Specs], page 353), or a stretch of whitespace such as specified by the `:width` or `:align-to` display properties (see Section 38.15.2 [Specified Space], page 351). The value is interpreted in the same way as a `display` text property. See Section 38.15 [Display Property], page 350.

A line prefix may also be specified for regions of text using the `line-prefix` text or overlay property. This takes precedence over the `line-prefix` variable. See Section 32.19.4 [Special Properties], page 162.

If your buffer contains *very* long lines, and you use continuation to display them, computing the continuation lines can make redisplay slow. The column computation and indentation functions also become slow. Then you might find it advisable to set `cache-long-line-scans` to `t`.

`cache-long-line-scans` [Variable]

If this variable is non-`nil`, various indentation and motion functions, and Emacs redisplay, cache the results of scanning the buffer, and consult the cache to avoid rescanning regions of the buffer unless they are modified.

Turning on the cache slows down processing of short lines somewhat.

This variable is automatically buffer-local in every buffer.

38.4 The Echo Area

The *echo area* is used for displaying error messages (see Section 10.5.3 [Errors], page 128, vol. 1), for messages made with the `message` primitive, and for echoing keystrokes. It is not the same as the minibuffer, despite the fact that the minibuffer appears (when active) in the same place on the screen as the echo area. See Section "The Minibuffer" in *The GNU Emacs Manual*.

Apart from the functions documented in this section, you can print Lisp objects to the echo area by specifying `t` as the output stream. See Section 19.4 [Output Streams], page 277, vol. 1.

38.4.1 Displaying Messages in the Echo Area

This section describes the standard functions for displaying messages in the echo area.

`message` *format-string* **&rest** *arguments* [Function]

This function displays a message in the echo area. *format-string* is a format string, and *arguments* are the objects for its format specifications, like in the `format` function (see Section 4.7 [Formatting Strings], page 57, vol. 1). The resulting formatted string is displayed in the echo area; if it contains `face` text properties, it is displayed with the specified faces (see Section 38.12 [Faces], page 325). The string is also added to the '`*Messages*`' buffer, but without text properties (see Section 38.4.3 [Logging Messages], page 305).

In batch mode, the message is printed to the standard error stream, followed by a newline.

If *format-string* is `nil` or the empty string, `message` clears the echo area; if the echo area has been expanded automatically, this brings it back to its normal size. If the minibuffer is active, this brings the minibuffer contents back onto the screen immediately.

```
(message "Minibuffer depth is %d."
         (minibuffer-depth))
 ⊣ Minibuffer depth is 0.
⇒ "Minibuffer depth is 0."

---------- Echo Area ----------
Minibuffer depth is 0.
---------- Echo Area ----------
```

To automatically display a message in the echo area or in a pop-buffer, depending on its size, use `display-message-or-buffer` (see below).

`with-temp-message` *message* **&rest** *body* [Macro]

This construct displays a message in the echo area temporarily, during the execution of *body*. It displays *message*, executes *body*, then returns the value of the last body form while restoring the previous echo area contents.

`message-or-box` *format-string* **&rest** *arguments* [Function]

This function displays a message like `message`, but may display it in a dialog box instead of the echo area. If this function is called in a command that was invoked using

the mouse—more precisely, if `last-nonmenu-event` (see Section 21.5 [Command Loop Info], page 324, vol. 1) is either `nil` or a list—then it uses a dialog box or pop-up menu to display the message. Otherwise, it uses the echo area. (This is the same criterion that `y-or-n-p` uses to make a similar decision; see Section 20.7 [Yes-or-No Queries], page 307, vol. 1.)

You can force use of the mouse or of the echo area by binding `last-nonmenu-event` to a suitable value around the call.

`message-box` *format-string* **&rest** *arguments* [Function]
This function displays a message like `message`, but uses a dialog box (or a pop-up menu) whenever that is possible. If it is impossible to use a dialog box or pop-up menu, because the terminal does not support them, then `message-box` uses the echo area, like `message`.

`display-message-or-buffer` *message* **&optional** *buffer-name* *not-this-window* *frame* [Function]
This function displays the message *message*, which may be either a string or a buffer. If it is shorter than the maximum height of the echo area, as defined by `max-mini-window-height`, it is displayed in the echo area, using `message`. Otherwise, `display-buffer` is used to show it in a pop-up buffer.

Returns either the string shown in the echo area, or when a pop-up buffer is used, the window used to display it.

If *message* is a string, then the optional argument *buffer-name* is the name of the buffer used to display it when a pop-up buffer is used, defaulting to '`*Message*`'. In the case where *message* is a string and displayed in the echo area, it is not specified whether the contents are inserted into the buffer anyway.

The optional arguments *not-this-window* and *frame* are as for `display-buffer`, and only used if a buffer is displayed.

`current-message` [Function]
This function returns the message currently being displayed in the echo area, or `nil` if there is none.

38.4.2 Reporting Operation Progress

When an operation can take a while to finish, you should inform the user about the progress it makes. This way the user can estimate remaining time and clearly see that Emacs is busy working, not hung. A convenient way to do this is to use a *progress reporter*.

Here is a working example that does nothing useful:

```
(let ((progress-reporter
       (make-progress-reporter "Collecting mana for Emacs..."
                               0  500)))
  (dotimes (k 500)
    (sit-for 0.01)
    (progress-reporter-update progress-reporter k))
  (progress-reporter-done progress-reporter))
```

`make-progress-reporter` *message* **&optional** *min-value max-value current-value min-change min-time* [Function]

This function creates and returns a progress reporter object, which you will use as an argument for the other functions listed below. The idea is to precompute as much data as possible to make progress reporting very fast.

When this progress reporter is subsequently used, it will display *message* in the echo area, followed by progress percentage. *message* is treated as a simple string. If you need it to depend on a filename, for instance, use `format` before calling this function.

The arguments *min-value* and *max-value* should be numbers standing for the starting and final states of the operation. For instance, an operation that "scans" a buffer should set these to the results of `point-min` and `point-max` correspondingly. *max-value* should be greater than *min-value*.

Alternatively, you can set *min-value* and *max-value* to `nil`. In that case, the progress reporter does not report process percentages; it instead displays a "spinner" that rotates a notch each time you update the progress reporter.

If *min-value* and *max-value* are numbers, you can give the argument *current-value* a numerical value specifying the initial progress; if omitted, this defaults to *min-value*.

The remaining arguments control the rate of echo area updates. The progress reporter will wait for at least *min-change* more percents of the operation to be completed before printing next message; the default is one percent. *min-time* specifies the minimum time in seconds to pass between successive prints; the default is 0.2 seconds. (On some operating systems, the progress reporter may handle fractions of seconds with varying precision).

This function calls `progress-reporter-update`, so the first message is printed immediately.

`progress-reporter-update` *reporter* **&optional** *value* [Function]

This function does the main work of reporting progress of your operation. It displays the message of *reporter*, followed by progress percentage determined by *value*. If percentage is zero, or close enough according to the *min-change* and *min-time* arguments, then it is omitted from the output.

reporter must be the result of a call to `make-progress-reporter`. *value* specifies the current state of your operation and must be between *min-value* and *max-value* (inclusive) as passed to `make-progress-reporter`. For instance, if you scan a buffer, then *value* should be the result of a call to `point`.

This function respects *min-change* and *min-time* as passed to `make-progress-reporter` and so does not output new messages on every invocation. It is thus very fast and normally you should not try to reduce the number of calls to it: resulting overhead will most likely negate your effort.

`progress-reporter-force-update` *reporter* **&optional** *value new-message* [Function]

This function is similar to `progress-reporter-update` except that it prints a message in the echo area unconditionally.

The first two arguments have the same meaning as for `progress-reporter-update`. Optional *new-message* allows you to change the message of the *reporter*. Since this

functions always updates the echo area, such a change will be immediately presented to the user.

`progress-reporter-done` *reporter* [Function]
This function should be called when the operation is finished. It prints the message of *reporter* followed by word "done" in the echo area.

You should always call this function and not hope for `progress-reporter-update` to print "100%". Firstly, it may never print it, there are many good reasons for this not to happen. Secondly, "done" is more explicit.

`dotimes-with-progress-reporter` (*var count* [*result*]) *message body...* [Macro]
This is a convenience macro that works the same way as `dotimes` does, but also reports loop progress using the functions described above. It allows you to save some typing.

You can rewrite the example in the beginning of this node using this macro this way:

```
(dotimes-with-progress-reporter
    (k 500)
    "Collecting some mana for Emacs..."
  (sit-for 0.01))
```

38.4.3 Logging Messages in '*Messages*'

Almost all the messages displayed in the echo area are also recorded in the '`*Messages*`' buffer so that the user can refer back to them. This includes all the messages that are output with `message`.

`message-log-max` [User Option]
This variable specifies how many lines to keep in the '`*Messages*`' buffer. The value `t` means there is no limit on how many lines to keep. The value `nil` disables message logging entirely. Here's how to display a message and prevent it from being logged:

```
(let (message-log-max)
  (message ...))
```

To make '`*Messages*`' more convenient for the user, the logging facility combines successive identical messages. It also combines successive related messages for the sake of two cases: question followed by answer, and a series of progress messages.

A "question followed by an answer" means two messages like the ones produced by `y-or-n-p`: the first is '*`question`*', and the second is '*`question...answer`*'. The first message conveys no additional information beyond what's in the second, so logging the second message discards the first from the log.

A "series of progress messages" means successive messages like those produced by `make-progress-reporter`. They have the form '*`base...how-far`*', where *base* is the same each time, while *how-far* varies. Logging each message in the series discards the previous one, provided they are consecutive.

The functions `make-progress-reporter` and `y-or-n-p` don't have to do anything special to activate the message log combination feature. It operates whenever two consecutive messages are logged that share a common prefix ending in '`...`'.

38.4.4 Echo Area Customization

These variables control details of how the echo area works.

`cursor-in-echo-area` [Variable]
: This variable controls where the cursor appears when a message is displayed in the echo area. If it is non-`nil`, then the cursor appears at the end of the message. Otherwise, the cursor appears at point—not in the echo area at all.

  The value is normally `nil`; Lisp programs bind it to `t` for brief periods of time.

`echo-area-clear-hook` [Variable]
: This normal hook is run whenever the echo area is cleared—either by `(message nil)` or for any other reason.

`echo-keystrokes` [User Option]
: This variable determines how much time should elapse before command characters echo. Its value must be an integer or floating point number, which specifies the number of seconds to wait before echoing. If the user types a prefix key (such as `C-x`) and then delays this many seconds before continuing, the prefix key is echoed in the echo area. (Once echoing begins in a key sequence, all subsequent characters in the same key sequence are echoed immediately.)

  If the value is zero, then command input is not echoed.

`message-truncate-lines` [Variable]
: Normally, displaying a long message resizes the echo area to display the entire message. But if the variable `message-truncate-lines` is non-`nil`, the echo area does not resize, and the message is truncated to fit it.

The variable `max-mini-window-height`, which specifies the maximum height for resizing minibuffer windows, also applies to the echo area (which is really a special use of the minibuffer window; see Section 20.14 [Minibuffer Misc], page 313, vol. 1).

38.5 Reporting Warnings

Warnings are a facility for a program to inform the user of a possible problem, but continue running.

38.5.1 Warning Basics

Every warning has a textual message, which explains the problem for the user, and a *severity level* which is a symbol. Here are the possible severity levels, in order of decreasing severity, and their meanings:

`:emergency`
: A problem that will seriously impair Emacs operation soon if you do not attend to it promptly.

`:error`
: A report of data or circumstances that are inherently wrong.

`:warning`
: A report of data or circumstances that are not inherently wrong, but raise suspicion of a possible problem.

`:debug` A report of information that may be useful if you are debugging.

When your program encounters invalid input data, it can either signal a Lisp error by calling `error` or `signal` or report a warning with severity `:error`. Signaling a Lisp error is the easiest thing to do, but it means the program cannot continue processing. If you want to take the trouble to implement a way to continue processing despite the bad data, then reporting a warning of severity `:error` is the right way to inform the user of the problem. For instance, the Emacs Lisp byte compiler can report an error that way and continue compiling other functions. (If the program signals a Lisp error and then handles it with `condition-case`, the user won't see the error message; it could show the message to the user by reporting it as a warning.)

Each warning has a *warning type* to classify it. The type is a list of symbols. The first symbol should be the custom group that you use for the program's user options. For example, byte compiler warnings use the warning type `(bytecomp)`. You can also subcategorize the warnings, if you wish, by using more symbols in the list.

`display-warning` *type message* **&optional** *level buffer-name* [Function]
: This function reports a warning, using *message* as the message and *type* as the warning type. *level* should be the severity level, with `:warning` being the default.

  buffer-name, if non-`nil`, specifies the name of the buffer for logging the warning. By default, it is '`*Warnings*`'.

`lwarn` *type level message* **&rest** *args* [Function]
: This function reports a warning using the value of `(format` *`message args...`*`)` as the message. In other respects it is equivalent to `display-warning`.

`warn` *message* **&rest** *args* [Function]
: This function reports a warning using the value of `(format` *`message args...`*`)` as the message, `(emacs)` as the type, and `:warning` as the severity level. It exists for compatibility only; we recommend not using it, because you should specify a specific warning type.

38.5.2 Warning Variables

Programs can customize how their warnings appear by binding the variables described in this section.

`warning-levels` [Variable]
: This list defines the meaning and severity order of the warning severity levels. Each element defines one severity level, and they are arranged in order of decreasing severity.

  Each element has the form `(`*`level string function`*`)`, where *level* is the severity level it defines. *string* specifies the textual description of this level. *string* should use '`%s`' to specify where to put the warning type information, or it can omit the '`%s`' so as not to include that information.

  The optional *function*, if non-`nil`, is a function to call with no arguments, to get the user's attention.

  Normally you should not change the value of this variable.

`warning-prefix-function` [Variable]

If non-`nil`, the value is a function to generate prefix text for warnings. Programs can bind the variable to a suitable function. `display-warning` calls this function with the warnings buffer current, and the function can insert text in it. That text becomes the beginning of the warning message.

The function is called with two arguments, the severity level and its entry in `warning-levels`. It should return a list to use as the entry (this value need not be an actual member of `warning-levels`). By constructing this value, the function can change the severity of the warning, or specify different handling for a given severity level.

If the variable's value is `nil` then there is no function to call.

`warning-series` [Variable]

Programs can bind this variable to `t` to say that the next warning should begin a series. When several warnings form a series, that means to leave point on the first warning of the series, rather than keep moving it for each warning so that it appears on the last one. The series ends when the local binding is unbound and `warning-series` becomes `nil` again.

The value can also be a symbol with a function definition. That is equivalent to `t`, except that the next warning will also call the function with no arguments with the warnings buffer current. The function can insert text which will serve as a header for the series of warnings.

Once a series has begun, the value is a marker which points to the buffer position in the warnings buffer of the start of the series.

The variable's normal value is `nil`, which means to handle each warning separately.

`warning-fill-prefix` [Variable]

When this variable is non-`nil`, it specifies a fill prefix to use for filling each warning's text.

`warning-type-format` [Variable]

This variable specifies the format for displaying the warning type in the warning message. The result of formatting the type this way gets included in the message under the control of the string in the entry in `warning-levels`. The default value is `" (%s)"`. If you bind it to `""` then the warning type won't appear at all.

38.5.3 Warning Options

These variables are used by users to control what happens when a Lisp program reports a warning.

`warning-minimum-level` [User Option]

This user option specifies the minimum severity level that should be shown immediately to the user. The default is `:warning`, which means to immediately display all warnings except `:debug` warnings.

`warning-minimum-log-level` [User Option]

This user option specifies the minimum severity level that should be logged in the warnings buffer. The default is `:warning`, which means to log all warnings except `:debug` warnings.

`warning-suppress-types` [User Option]
: This list specifies which warning types should not be displayed immediately for the user. Each element of the list should be a list of symbols. If its elements match the first elements in a warning type, then that warning is not displayed immediately.

`warning-suppress-log-types` [User Option]
: This list specifies which warning types should not be logged in the warnings buffer. Each element of the list should be a list of symbols. If it matches the first few elements in a warning type, then that warning is not logged.

38.5.4 Delayed Warnings

Sometimes, you may wish to avoid showing a warning while a command is running, and only show it only after the end of the command. You can use the variable `delayed-warnings-list` for this.

`delayed-warnings-list` [Variable]
: The value of this variable is a list of warnings to be displayed after the current command has finished. Each element must be a list

```
(type message [level [buffer-name]])
```

with the same form, and the same meanings, as the argument list of `display-warning` (see Section 38.5.1 [Warning Basics], page 306). Immediately after running `post-command-hook` (see Section 21.1 [Command Overview], page 315, vol. 1), the Emacs command loop displays all the warnings specified by this variable, then resets it to `nil`.

Programs which need to further customize the delayed warnings mechanism can change the variable `delayed-warnings-hook`:

`delayed-warnings-hook` [Variable]
: This is a normal hook which is run by the Emacs command loop, after `post-command-hook`, in order to to process and display delayed warnings.

Its default value is a list of two functions:

```
(collapse-delayed-warnings display-delayed-warnings)
```

The function `collapse-delayed-warnings` removes repeated entries from `delayed-warnings-list`. The function `display-delayed-warnings` calls `display-warning` on each of the entries in `delayed-warnings-list`, in turn, and then sets `delayed-warnings-list` to `nil`.

38.6 Invisible Text

You can make characters *invisible*, so that they do not appear on the screen, with the `invisible` property. This can be either a text property (see Section 32.19 [Text Properties], page 156) or an overlay property (see Section 38.9 [Overlays], page 315). Cursor motion also partly ignores these characters; if the command loop finds that point is inside a range of invisible text after a command, it relocates point to the other side of the text.

In the simplest case, any non-`nil` `invisible` property makes a character invisible. This is the default case—if you don't alter the default value of `buffer-invisibility-spec`,

this is how the `invisible` property works. You should normally use `t` as the value of the `invisible` property if you don't plan to set `buffer-invisibility-spec` yourself.

More generally, you can use the variable `buffer-invisibility-spec` to control which values of the `invisible` property make text invisible. This permits you to classify the text into different subsets in advance, by giving them different `invisible` values, and subsequently make various subsets visible or invisible by changing the value of `buffer-invisibility-spec`.

Controlling visibility with `buffer-invisibility-spec` is especially useful in a program to display the list of entries in a database. It permits the implementation of convenient filtering commands to view just a part of the entries in the database. Setting this variable is very fast, much faster than scanning all the text in the buffer looking for properties to change.

`buffer-invisibility-spec` [Variable]

This variable specifies which kinds of `invisible` properties actually make a character invisible. Setting this variable makes it buffer-local.

`t` A character is invisible if its `invisible` property is non-`nil`. This is the default.

a list Each element of the list specifies a criterion for invisibility; if a character's `invisible` property fits any one of these criteria, the character is invisible. The list can have two kinds of elements:

`atom` A character is invisible if its `invisible` property value is *atom* or if it is a list with *atom* as a member; comparison is done with `eq`.

`(`*`atom`*` . t)`
A character is invisible if its `invisible` property value is *atom* or if it is a list with *atom* as a member; comparison is done with `eq`. Moreover, a sequence of such characters displays as an ellipsis.

Two functions are specifically provided for adding elements to `buffer-invisibility-spec` and removing elements from it.

`add-to-invisibility-spec` *element* [Function]

This function adds the element *element* to `buffer-invisibility-spec`. If `buffer-invisibility-spec` was `t`, it changes to a list, `(t)`, so that text whose `invisible` property is `t` remains invisible.

`remove-from-invisibility-spec` *element* [Function]

This removes the element *element* from `buffer-invisibility-spec`. This does nothing if *element* is not in the list.

A convention for use of `buffer-invisibility-spec` is that a major mode should use the mode's own name as an element of `buffer-invisibility-spec` and as the value of the `invisible` property:

```
;; If you want to display an ellipsis:
(add-to-invisibility-spec '(my-symbol . t))
;; If you don't want ellipsis:
(add-to-invisibility-spec 'my-symbol)

(overlay-put (make-overlay beginning end)
             'invisible 'my-symbol)

;; When done with the invisibility:
(remove-from-invisibility-spec '(my-symbol . t))
;; Or respectively:
(remove-from-invisibility-spec 'my-symbol)
```

You can check for invisibility using the following function:

`invisible-p` *pos-or-prop* [Function]

If *pos-or-prop* is a marker or number, this function returns a non-`nil` value if the text at that position is invisible.

If *pos-or-prop* is any other kind of Lisp object, that is taken to mean a possible value of the `invisible` text or overlay property. In that case, this function returns a non-`nil` value if that value would cause text to become invisible, based on the current value of `buffer-invisibility-spec`.

Ordinarily, functions that operate on text or move point do not care whether the text is invisible. The user-level line motion commands ignore invisible newlines if `line-move-ignore-invisible` is non-`nil` (the default), but only because they are explicitly programmed to do so.

However, if a command ends with point inside or at the boundary of invisible text, the main editing loop relocates point to one of the two ends of the invisible text. Emacs chooses the direction of relocation so that it is the same as the overall movement direction of the command; if in doubt, it prefers a position where an inserted char would not inherit the `invisible` property. Additionally, if the text is not replaced by an ellipsis and the command only moved within the invisible text, then point is moved one extra character so as to try and reflect the command's movement by a visible movement of the cursor.

Thus, if the command moved point back to an invisible range (with the usual stickiness), Emacs moves point back to the beginning of that range. If the command moved point forward into an invisible range, Emacs moves point forward to the first visible character that follows the invisible text and then forward one more character.

Incremental search can make invisible overlays visible temporarily and/or permanently when a match includes invisible text. To enable this, the overlay should have a non-`nil` `isearch-open-invisible` property. The property value should be a function to be called with the overlay as an argument. This function should make the overlay visible permanently; it is used when the match overlaps the overlay on exit from the search.

During the search, such overlays are made temporarily visible by temporarily modifying their invisible and intangible properties. If you want this to be done differently for a certain overlay, give it an `isearch-open-invisible-temporary` property which is a function. The function is called with two arguments: the first is the overlay, and the second is `nil` to make the overlay visible, or `t` to make it invisible again.

38.7 Selective Display

Selective display refers to a pair of related features for hiding certain lines on the screen.

The first variant, explicit selective display, is designed for use in a Lisp program: it controls which lines are hidden by altering the text. This kind of hiding in some ways resembles the effect of the `invisible` property (see Section 38.6 [Invisible Text], page 309), but the two features are different and do not work the same way.

In the second variant, the choice of lines to hide is made automatically based on indentation. This variant is designed to be a user-level feature.

The way you control explicit selective display is by replacing a newline (control-j) with a carriage return (control-m). The text that was formerly a line following that newline is now hidden. Strictly speaking, it is temporarily no longer a line at all, since only newlines can separate lines; it is now part of the previous line.

Selective display does not directly affect editing commands. For example, `C-f` (`forward-char`) moves point unhesitatingly into hidden text. However, the replacement of newline characters with carriage return characters affects some editing commands. For example, `next-line` skips hidden lines, since it searches only for newlines. Modes that use selective display can also define commands that take account of the newlines, or that control which parts of the text are hidden.

When you write a selectively displayed buffer into a file, all the control-m's are output as newlines. This means that when you next read in the file, it looks OK, with nothing hidden. The selective display effect is seen only within Emacs.

`selective-display` [Variable]

This buffer-local variable enables selective display. This means that lines, or portions of lines, may be made hidden.

- If the value of `selective-display` is `t`, then the character control-m marks the start of hidden text; the control-m, and the rest of the line following it, are not displayed. This is explicit selective display.
- If the value of `selective-display` is a positive integer, then lines that start with more than that many columns of indentation are not displayed.

When some portion of a buffer is hidden, the vertical movement commands operate as if that portion did not exist, allowing a single `next-line` command to skip any number of hidden lines. However, character movement commands (such as `forward-char`) do not skip the hidden portion, and it is possible (if tricky) to insert or delete text in an hidden portion.

In the examples below, we show the *display appearance* of the buffer `foo`, which changes with the value of `selective-display`. The *contents* of the buffer do not change.

```
(setq selective-display nil)
     ⇒ nil

---------- Buffer: foo ----------
1 on this column
 2on this column
  3n this column
  3n this column
 2on this column
1 on this column
---------- Buffer: foo ----------

(setq selective-display 2)
     ⇒ 2

---------- Buffer: foo ----------
1 on this column
 2on this column
 2on this column
1 on this column
---------- Buffer: foo ----------
```

`selective-display-ellipses` [User Option]

If this buffer-local variable is non-`nil`, then Emacs displays '`...`' at the end of a line that is followed by hidden text. This example is a continuation of the previous one.

```
(setq selective-display-ellipses t)
     ⇒ t

---------- Buffer: foo ----------
1 on this column
 2on this column ...
 2on this column
1 on this column
---------- Buffer: foo ----------
```

You can use a display table to substitute other text for the ellipsis ('`...`'). See Section 38.20.2 [Display Tables], page 377.

38.8 Temporary Displays

Temporary displays are used by Lisp programs to put output into a buffer and then present it to the user for perusal rather than for editing. Many help commands use this feature.

`with-output-to-temp-buffer` *buffer-name forms...* [Macro]

This function executes *forms* while arranging to insert any output they print into the buffer named *buffer-name*, which is first created if necessary, and put into Help mode. Finally, the buffer is displayed in some window, but not selected.

If the *forms* do not change the major mode in the output buffer, so that it is still Help mode at the end of their execution, then `with-output-to-temp-buffer` makes this

buffer read-only at the end, and also scans it for function and variable names to make them into clickable cross-references. See [Tips for Documentation Strings], page 451, in particular the item on hyperlinks in documentation strings, for more details.

The string *buffer-name* specifies the temporary buffer, which need not already exist. The argument must be a string, not a buffer. The buffer is erased initially (with no questions asked), and it is marked as unmodified after `with-output-to-temp-buffer` exits.

`with-output-to-temp-buffer` binds `standard-output` to the temporary buffer, then it evaluates the forms in *forms*. Output using the Lisp output functions within *forms* goes by default to that buffer (but screen display and messages in the echo area, although they are "output" in the general sense of the word, are not affected). See Section 19.5 [Output Functions], page 279, vol. 1.

Several hooks are available for customizing the behavior of this construct; they are listed below.

The value of the last form in *forms* is returned.

```
---------- Buffer: foo ----------
 This is the contents of foo.
---------- Buffer: foo ----------

(with-output-to-temp-buffer "foo"
    (print 20)
    (print standard-output))
⇒ #<buffer foo>

---------- Buffer: foo ----------
20

#<buffer foo>

---------- Buffer: foo ----------
```

`temp-buffer-show-function` [User Option]

If this variable is non-`nil`, `with-output-to-temp-buffer` calls it as a function to do the job of displaying a help buffer. The function gets one argument, which is the buffer it should display.

It is a good idea for this function to run `temp-buffer-show-hook` just as `with-output-to-temp-buffer` normally would, inside of `save-selected-window` and with the chosen window and buffer selected.

`temp-buffer-setup-hook` [Variable]

This normal hook is run by `with-output-to-temp-buffer` before evaluating *body*. When the hook runs, the temporary buffer is current. This hook is normally set up with a function to put the buffer in Help mode.

`temp-buffer-show-hook` [Variable]
: This normal hook is run by `with-output-to-temp-buffer` after displaying the temporary buffer. When the hook runs, the temporary buffer is current, and the window it was displayed in is selected.

`momentary-string-display` *string position* **&optional** *char message* [Function]
: This function momentarily displays *string* in the current buffer at *position*. It has no effect on the undo list or on the buffer's modification status.

  The momentary display remains until the next input event. If the next input event is *char*, `momentary-string-display` ignores it and returns. Otherwise, that event remains buffered for subsequent use as input. Thus, typing *char* will simply remove the string from the display, while typing (say) `C-f` will remove the string from the display and later (presumably) move point forward. The argument *char* is a space by default.

  The return value of `momentary-string-display` is not meaningful.

  If the string *string* does not contain control characters, you can do the same job in a more general way by creating (and then subsequently deleting) an overlay with a `before-string` property. See Section 38.9.2 [Overlay Properties], page 318.

  If *message* is non-`nil`, it is displayed in the echo area while *string* is displayed in the buffer. If it is `nil`, a default message says to type *char* to continue.

  In this example, point is initially located at the beginning of the second line:

```
---------- Buffer: foo ----------
This is the contents of foo.
*Second line.
---------- Buffer: foo ----------

(momentary-string-display
  "**** Important Message! ****"
  (point) ?\r
  "Type RET when done reading")
⇒ t

---------- Buffer: foo ----------
This is the contents of foo.
**** Important Message! ****Second line.
---------- Buffer: foo ----------

---------- Echo Area ----------
Type RET when done reading
---------- Echo Area ----------
```

38.9 Overlays

You can use *overlays* to alter the appearance of a buffer's text on the screen, for the sake of presentation features. An overlay is an object that belongs to a particular buffer, and has a specified beginning and end. It also has properties that you can examine and set; these affect the display of the text within the overlay.

The visual effect of an overlay is the same as of the corresponding text property (see Section 32.19 [Text Properties], page 156). However, due to a different implementation, overlays generally don't scale well (many operations take a time that is proportional to the number of overlays in the buffer). If you need to affect the visual appearance of many portions in the buffer, we recommend using text properties.

An overlay uses markers to record its beginning and end; thus, editing the text of the buffer adjusts the beginning and end of each overlay so that it stays with the text. When you create the overlay, you can specify whether text inserted at the beginning should be inside the overlay or outside, and likewise for the end of the overlay.

38.9.1 Managing Overlays

This section describes the functions to create, delete and move overlays, and to examine their contents. Overlay changes are not recorded in the buffer's undo list, since the overlays are not part of the buffer's contents.

`overlayp` *object* [Function]
: This function returns `t` if *object* is an overlay.

`make-overlay` *start end* **&optional** *buffer front-advance rear-advance* [Function]
: This function creates and returns an overlay that belongs to *buffer* and ranges from *start* to *end*. Both *start* and *end* must specify buffer positions; they may be integers or markers. If *buffer* is omitted, the overlay is created in the current buffer.

  The arguments *front-advance* and *rear-advance* specify the marker insertion type for the start of the overlay and for the end of the overlay, respectively. See Section 31.5 [Marker Insertion Types], page 116. If they are both `nil`, the default, then the overlay extends to include any text inserted at the beginning, but not text inserted at the end. If *front-advance* is non-`nil`, text inserted at the beginning of the overlay is excluded from the overlay. If *rear-advance* is non-`nil`, text inserted at the end of the overlay is included in the overlay.

`overlay-start` *overlay* [Function]
: This function returns the position at which *overlay* starts, as an integer.

`overlay-end` *overlay* [Function]
: This function returns the position at which *overlay* ends, as an integer.

`overlay-buffer` *overlay* [Function]
: This function returns the buffer that *overlay* belongs to. It returns `nil` if *overlay* has been deleted.

`delete-overlay` *overlay* [Function]
: This function deletes *overlay*. The overlay continues to exist as a Lisp object, and its property list is unchanged, but it ceases to be attached to the buffer it belonged to, and ceases to have any effect on display.

  A deleted overlay is not permanently disconnected. You can give it a position in a buffer again by calling `move-overlay`.

`move-overlay` *overlay start end* **&optional** *buffer* [Function]
This function moves *overlay* to *buffer*, and places its bounds at *start* and *end*. Both arguments *start* and *end* must specify buffer positions; they may be integers or markers.

If *buffer* is omitted, *overlay* stays in the same buffer it was already associated with; if *overlay* was deleted, it goes into the current buffer.

The return value is *overlay*.

This is the only valid way to change the endpoints of an overlay. Do not try modifying the markers in the overlay by hand, as that fails to update other vital data structures and can cause some overlays to be "lost".

`remove-overlays` **&optional** *start end name value* [Function]
This function removes all the overlays between *start* and *end* whose property *name* has the value *value*. It can move the endpoints of the overlays in the region, or split them.

If *name* is omitted or `nil`, it means to delete all overlays in the specified region. If *start* and/or *end* are omitted or `nil`, that means the beginning and end of the buffer respectively. Therefore, `(remove-overlays)` removes all the overlays in the current buffer.

`copy-overlay` *overlay* [Function]
This function returns a copy of *overlay*. The copy has the same endpoints and properties as *overlay*. However, the marker insertion type for the start of the overlay and for the end of the overlay are set to their default values (see Section 31.5 [Marker Insertion Types], page 116).

Here are some examples:

```
;; Create an overlay.
(setq foo (make-overlay 1 10))
     ⇒ #<overlay from 1 to 10 in display.texi>
(overlay-start foo)
     ⇒ 1
(overlay-end foo)
     ⇒ 10
(overlay-buffer foo)
     ⇒ #<buffer display.texi>
;; Give it a property we can check later.
(overlay-put foo 'happy t)
     ⇒ t
;; Verify the property is present.
(overlay-get foo 'happy)
     ⇒ t
;; Move the overlay.
(move-overlay foo 5 20)
     ⇒ #<overlay from 5 to 20 in display.texi>
(overlay-start foo)
     ⇒ 5
```

```
(overlay-end foo)
     ⇒ 20
;; Delete the overlay.
(delete-overlay foo)
     ⇒ nil
;; Verify it is deleted.
foo
     ⇒ #<overlay in no buffer>
;; A deleted overlay has no position.
(overlay-start foo)
     ⇒ nil
(overlay-end foo)
     ⇒ nil
(overlay-buffer foo)
     ⇒ nil
;; Undelete the overlay.
(move-overlay foo 1 20)
     ⇒ #<overlay from 1 to 20 in display.texi>
;; Verify the results.
(overlay-start foo)
     ⇒ 1
(overlay-end foo)
     ⇒ 20
(overlay-buffer foo)
     ⇒ #<buffer display.texi>
;; Moving and deleting the overlay does not change its properties.
(overlay-get foo 'happy)
     ⇒ t
```

Emacs stores the overlays of each buffer in two lists, divided around an arbitrary "center position". One list extends backwards through the buffer from that center position, and the other extends forwards from that center position. The center position can be anywhere in the buffer.

`overlay-recenter` *pos* [Function]

This function recenters the overlays of the current buffer around position *pos*. That makes overlay lookup faster for positions near *pos*, but slower for positions far away from *pos*.

A loop that scans the buffer forwards, creating overlays, can run faster if you do `(overlay-recenter (point-max))` first.

38.9.2 Overlay Properties

Overlay properties are like text properties in that the properties that alter how a character is displayed can come from either source. But in most respects they are different. See Section 32.19 [Text Properties], page 156, for comparison.

Text properties are considered a part of the text; overlays and their properties are specifically considered not to be part of the text. Thus, copying text between various buffers

and strings preserves text properties, but does not try to preserve overlays. Changing a buffer's text properties marks the buffer as modified, while moving an overlay or changing its properties does not. Unlike text property changes, overlay property changes are not recorded in the buffer's undo list.

Since more than one overlay can specify a property value for the same character, Emacs lets you specify a priority value of each overlay. You should not make assumptions about which overlay will prevail when there is a conflict and they have the same priority.

These functions read and set the properties of an overlay:

`overlay-get` *overlay prop* [Function]
: This function returns the value of property *prop* recorded in *overlay*, if any. If *overlay* does not record any value for that property, but it does have a `category` property which is a symbol, that symbol's *prop* property is used. Otherwise, the value is `nil`.

`overlay-put` *overlay prop value* [Function]
: This function sets the value of property *prop* recorded in *overlay* to *value*. It returns *value*.

`overlay-properties` *overlay* [Function]
: This returns a copy of the property list of *overlay*.

See also the function `get-char-property` which checks both overlay properties and text properties for a given character. See Section 32.19.1 [Examining Properties], page 157.

Many overlay properties have special meanings; here is a table of them:

`priority`
: This property's value (which should be a non-negative integer number) determines the priority of the overlay. No priority, or `nil`, means zero.

  The priority matters when two or more overlays cover the same character and both specify the same property; the one whose `priority` value is larger overrides the other. For the `face` property, the higher priority overlay's value does not completely override the other value; instead, its face attributes override the face attributes of the lower priority `face` property.

  Currently, all overlays take priority over text properties. Please avoid using negative priority values, as we have not yet decided just what they should mean.

`window`
: If the `window` property is non-`nil`, then the overlay applies only on that window.

`category`
: If an overlay has a `category` property, we call it the *category* of the overlay. It should be a symbol. The properties of the symbol serve as defaults for the properties of the overlay.

`face`
: This property controls the way text is displayed—for example, which font and which colors. See Section 38.12 [Faces], page 325, for more information.

  In the simplest case, the value is a face name. It can also be a list; then each element can be any of these possibilities:

  - A face name (a symbol or string).

- A property list of face attributes. This has the form (*keyword value* ...), where each *keyword* is a face attribute name and *value* is a meaningful value for that attribute. With this feature, you do not need to create a face each time you want to specify a particular attribute for certain text. See Section 38.12.2 [Face Attributes], page 327.
- A cons cell, of the form `(foreground-color . color-name)` or `(background-color . color-name)`. These elements specify just the foreground color or just the background color.

  `(foreground-color . color-name)` has the same effect as `(:foreground color-name)`; likewise for the background.

`mouse-face`
: This property is used instead of `face` when the mouse is within the range of the overlay. However, Emacs ignores all face attributes from this property that alter the text size (e.g. `:height`, `:weight`, and `:slant`). Those attributes are always the same as in the unhighlighted text.

`display`
: This property activates various features that change the way text is displayed. For example, it can make text appear taller or shorter, higher or lower, wider or narrower, or replaced with an image. See Section 38.15 [Display Property], page 350.

`help-echo`
: If an overlay has a `help-echo` property, then when you move the mouse onto the text in the overlay, Emacs displays a help string in the echo area, or in the tooltip window. For details see [Text help-echo], page 163.

`modification-hooks`
: This property's value is a list of functions to be called if any character within the overlay is changed or if text is inserted strictly within the overlay.

  The hook functions are called both before and after each change. If the functions save the information they receive, and compare notes between calls, they can determine exactly what change has been made in the buffer text.

  When called before a change, each function receives four arguments: the overlay, `nil`, and the beginning and end of the text range to be modified.

  When called after a change, each function receives five arguments: the overlay, `t`, the beginning and end of the text range just modified, and the length of the pre-change text replaced by that range. (For an insertion, the pre-change length is zero; for a deletion, that length is the number of characters deleted, and the post-change beginning and end are equal.)

  If these functions modify the buffer, they should bind `inhibit-modification-hooks` to `t` around doing so, to avoid confusing the internal mechanism that calls these hooks.

  Text properties also support the `modification-hooks` property, but the details are somewhat different (see Section 32.19.4 [Special Properties], page 162).

`insert-in-front-hooks`
: This property's value is a list of functions to be called before and after inserting text right at the beginning of the overlay. The calling conventions are the same as for the `modification-hooks` functions.

`insert-behind-hooks`
: This property's value is a list of functions to be called before and after inserting text right at the end of the overlay. The calling conventions are the same as for the `modification-hooks` functions.

`invisible`
: The `invisible` property can make the text in the overlay invisible, which means that it does not appear on the screen. See Section 38.6 [Invisible Text], page 309, for details.

`intangible`
: The `intangible` property on an overlay works just like the `intangible` text property. See Section 32.19.4 [Special Properties], page 162, for details.

`isearch-open-invisible`
: This property tells incremental search how to make an invisible overlay visible, permanently, if the final match overlaps it. See Section 38.6 [Invisible Text], page 309.

`isearch-open-invisible-temporary`
: This property tells incremental search how to make an invisible overlay visible, temporarily, during the search. See Section 38.6 [Invisible Text], page 309.

`before-string`
: This property's value is a string to add to the display at the beginning of the overlay. The string does not appear in the buffer in any sense—only on the screen.

`after-string`
: This property's value is a string to add to the display at the end of the overlay. The string does not appear in the buffer in any sense—only on the screen.

`line-prefix`
: This property specifies a display spec to prepend to each non-continuation line at display-time. See Section 38.3 [Truncation], page 300.

`wrap-prefix`
: This property specifies a display spec to prepend to each continuation line at display-time. See Section 38.3 [Truncation], page 300.

`evaporate`
: If this property is non-`nil`, the overlay is deleted automatically if it becomes empty (i.e., if its length becomes zero). If you give an empty overlay a non-`nil` `evaporate` property, that deletes it immediately.

`local-map`
: If this property is non-`nil`, it specifies a keymap for a portion of the text. The property's value replaces the buffer's local map, when the character after point is within the overlay. See Section 22.7 [Active Keymaps], page 367, vol. 1.

`keymap` The `keymap` property is similar to `local-map` but overrides the buffer's local map (and the map specified by the `local-map` property) rather than replacing it.

The `local-map` and `keymap` properties do not affect a string displayed by the `before-string`, `after-string`, or `display` properties. This is only relevant for mouse clicks and other mouse events that fall on the string, since point is never on the string. To bind special mouse events for the string, assign it a `local-map` or `keymap` text property. See Section 32.19.4 [Special Properties], page 162.

38.9.3 Searching for Overlays

`overlays-at` *pos* [Function]
This function returns a list of all the overlays that cover the character at position *pos* in the current buffer. The list is in no particular order. An overlay contains position *pos* if it begins at or before *pos*, and ends after *pos*.

To illustrate usage, here is a Lisp function that returns a list of the overlays that specify property *prop* for the character at point:

```
(defun find-overlays-specifying (prop)
  (let ((overlays (overlays-at (point)))
        found)
    (while overlays
      (let ((overlay (car overlays)))
        (if (overlay-get overlay prop)
            (setq found (cons overlay found))))
      (setq overlays (cdr overlays)))
    found))
```

`overlays-in` *beg end* [Function]
This function returns a list of the overlays that overlap the region *beg* through *end*. "Overlap" means that at least one character is contained within the overlay and also contained within the specified region; however, empty overlays are included in the result if they are located at *beg*, strictly between *beg* and *end*, or at *end* when *end* denotes the position at the end of the buffer.

`next-overlay-change` *pos* [Function]
This function returns the buffer position of the next beginning or end of an overlay, after *pos*. If there is none, it returns `(point-max)`.

`previous-overlay-change` *pos* [Function]
This function returns the buffer position of the previous beginning or end of an overlay, before *pos*. If there is none, it returns `(point-min)`.

As an example, here's a simplified (and inefficient) version of the primitive function `next-single-char-property-change` (see Section 32.19.3 [Property Search], page 160). It searches forward from position *pos* for the next position where the value of a given property `prop`, as obtained from either overlays or text properties, changes.

```
(defun next-single-char-property-change (position prop)
  (save-excursion
    (goto-char position)
    (let ((propval (get-char-property (point) prop)))
```

```
  (while (and (not (eobp))
              (eq (get-char-property (point) prop) propval))
    (goto-char (min (next-overlay-change (point))
                    (next-single-property-change (point) prop)))))
(point)))
```

38.10 Width

Since not all characters have the same width, these functions let you check the width of a character. See Section 32.17.1 [Primitive Indent], page 151, and Section 30.2.5 [Screen Lines], page 103, for related functions.

`char-width` *char* [Function]

This function returns the width in columns of the character *char*, if it were displayed in the current buffer (i.e. taking into account the buffer's display table, if any; see Section 38.20.2 [Display Tables], page 377). The width of a tab character is usually `tab-width` (see Section 38.20.1 [Usual Display], page 376).

`string-width` *string* [Function]

This function returns the width in columns of the string *string*, if it were displayed in the current buffer and the selected window.

`truncate-string-to-width` *string width* **&optional** *start-column padding ellipsis* [Function]

This function returns the part of *string* that fits within *width* columns, as a new string.

If *string* does not reach *width*, then the result ends where *string* ends. If one multi-column character in *string* extends across the column *width*, that character is not included in the result. Thus, the result can fall short of *width* but cannot go beyond it.

The optional argument *start-column* specifies the starting column. If this is non-`nil`, then the first *start-column* columns of the string are omitted from the value. If one multi-column character in *string* extends across the column *start-column*, that character is not included.

The optional argument *padding*, if non-`nil`, is a padding character added at the beginning and end of the result string, to extend it to exactly *width* columns. The padding character is used at the end of the result if it falls short of *width*. It is also used at the beginning of the result if one multi-column character in *string* extends across the column *start-column*.

If *ellipsis* is non-`nil`, it should be a string which will replace the end of *str* (including any padding) if it extends beyond *end-column*, unless the display width of *str* is equal to or less than the display width of *ellipsis*. If *ellipsis* is non-`nil` and not a string, it stands for `"..."`.

```
(truncate-string-to-width "\tab\t" 12 4)
     ⇒ "ab"
(truncate-string-to-width "\tab\t" 12 4 ?\s)
     ⇒ "    ab  "
```

38.11 Line Height

The total height of each display line consists of the height of the contents of the line, plus optional additional vertical line spacing above or below the display line.

The height of the line contents is the maximum height of any character or image on that display line, including the final newline if there is one. (A display line that is continued doesn't include a final newline.) That is the default line height, if you do nothing to specify a greater height. (In the most common case, this equals the height of the default frame font.)

There are several ways to explicitly specify a larger line height, either by specifying an absolute height for the display line, or by specifying vertical space. However, no matter what you specify, the actual line height can never be less than the default.

A newline can have a `line-height` text or overlay property that controls the total height of the display line ending in that newline.

If the property value is `t`, the newline character has no effect on the displayed height of the line—the visible contents alone determine the height. This is useful for tiling small images (or image slices) without adding blank areas between the images.

If the property value is a list of the form `(`*`height total`*`)`, that adds extra space *below* the display line. First Emacs uses *height* as a height spec to control extra space *above* the line; then it adds enough space *below* the line to bring the total line height up to *total*. In this case, the other ways to specify the line spacing are ignored.

Any other kind of property value is a height spec, which translates into a number—the specified line height. There are several ways to write a height spec; here's how each of them translates into a number:

`integer` If the height spec is a positive integer, the height value is that integer.

`float` If the height spec is a float, *float*, the numeric height value is *float* times the frame's default line height.

`(`*`face`*` . `*`ratio`*`)`
If the height spec is a cons of the format shown, the numeric height is *ratio* times the height of face *face*. *ratio* can be any type of number, or `nil` which means a ratio of 1. If *face* is `t`, it refers to the current face.

`(nil . `*`ratio`*`)`
If the height spec is a cons of the format shown, the numeric height is *ratio* times the height of the contents of the line.

Thus, any valid height spec determines the height in pixels, one way or another. If the line contents' height is less than that, Emacs adds extra vertical space above the line to achieve the specified total height.

If you don't specify the `line-height` property, the line's height consists of the contents' height plus the line spacing. There are several ways to specify the line spacing for different parts of Emacs text.

On graphical terminals, you can specify the line spacing for all lines in a frame, using the `line-spacing` frame parameter (see Section 29.3.3.4 [Layout Parameters], page 74).

However, if the default value of `line-spacing` is non-`nil`, it overrides the frame's `line-spacing` parameter. An integer value specifies the number of pixels put below lines. A floating point number specifies the spacing relative to the frame's default line height.

You can specify the line spacing for all lines in a buffer via the buffer-local `line-spacing` variable. An integer value specifies the number of pixels put below lines. A floating point number specifies the spacing relative to the default frame line height. This overrides line spacings specified for the frame.

Finally, a newline can have a `line-spacing` text or overlay property that overrides the default frame line spacing and the buffer local `line-spacing` variable, for the display line ending in that newline.

One way or another, these mechanisms specify a Lisp value for the spacing of each line. The value is a height spec, and it translates into a Lisp value as described above. However, in this case the numeric height value specifies the line spacing, rather than the line height.

On text terminals, the line spacing cannot be altered.

38.12 Faces

A *face* is a collection of graphical *attributes* for displaying text: font, foreground color, background color, optional underlining, and so on. Faces control how Emacs displays text in buffers, as well as other parts of the frame such as the mode line. See Section "Standard Faces" in *The GNU Emacs Manual*, for the list of faces Emacs normally comes with.

For most purposes, you refer to a face in Lisp programs using its *face name*, which is usually a Lisp symbol. For backward compatibility, a face name can also be a string, which is equivalent to a Lisp symbol of the same name.

`facep` *object* [Function]
: This function returns a non-`nil` value if *object* is a Lisp symbol or string that names a face. Otherwise, it returns `nil`.

By default, each face name corresponds to the same set of attributes in all frames. But you can also assign a face name a special set of attributes in one frame (see Section 38.12.3 [Attribute Functions], page 330).

38.12.1 Defining Faces

The `defface` macro defines a face and specifies its default appearance. The user can subsequently customize the face using the Customize interface (see Chapter 14 [Customization], page 190, vol. 1).

`defface` *face spec doc* [*keyword value*]... [Macro]
: This macro declares *face* as a customizable face whose default attributes are given by *spec*. You should not quote the symbol *face*, and it should not end in '`-face`' (that would be redundant). The argument *doc* is a documentation string for the face. The additional *keyword* arguments have the same meanings as in `defgroup` and `defcustom` (see Section 14.1 [Common Keywords], page 190, vol. 1).

  When `defface` executes, it defines the face according to *spec*, then uses any customizations that were read from the init file (see Section 39.1.2 [Init File], page 389) to override that specification.

When you evaluate a `defface` form with `C-M-x` in Emacs Lisp mode (`eval-defun`), a special feature of `eval-defun` overrides any customizations of the face. This way, the face reflects exactly what the `defface` says.

The *spec* argument is a *face specification*, which states how the face should appear on different kinds of terminals. It should be an alist whose elements each have the form `(`*`display atts`*`)`. *display* specifies a class of terminals (see below), while *atts* is a property list of face attributes and their values, specifying the appearance of the face on matching terminals (see the next section for details about face attributes).

The *display* part of an element of *spec* determines which frames the element matches. If more than one element of *spec* matches a given frame, the first element that matches is the one used for that frame. There are three possibilities for *display*:

`default` This element of *spec* doesn't match any frames; instead, it specifies defaults that apply to all frames. This kind of element, if used, must be the first element of *spec*. Each of the following elements can override any or all of these defaults.

`t` This element of *spec* matches all frames. Therefore, any subsequent elements of *spec* are never used. Normally `t` is used in the last (or only) element of *spec*.

a list If *display* is a list, each element should have the form `(`*`characteristic value...`*`)`. Here *characteristic* specifies a way of classifying frames, and the *value*s are possible classifications which *display* should apply to. Here are the possible values of *characteristic*:

`type` The kind of window system the frame uses—either `graphic` (any graphics-capable display), `x`, `pc` (for the MS-DOS console), `w32` (for MS Windows 9X/NT/2K/XP), or `tty` (a non-graphics-capable display). See Section 38.22 [Window Systems], page 382.

`class` What kinds of colors the frame supports—either `color`, `grayscale`, or `mono`.

`background`
The kind of background—either `light` or `dark`.

`min-colors`
An integer that represents the minimum number of colors the frame should support. This matches a frame if its `display-color-cells` value is at least the specified integer.

`supports` Whether or not the frame can display the face attributes given in *value*... (see Section 38.12.2 [Face Attributes], page 327). See [Display Face Attribute Testing], page 96, for more information on exactly how this testing is done.

If an element of *display* specifies more than one *value* for a given *characteristic*, any of those values is acceptable. If *display* has more than one element, each element should specify a different *characteristic*; then *each*

characteristic of the frame must match one of the *value*s specified for it in *display*.

Here's how the standard face `highlight` is defined:

```
(defface highlight
  '((((class color) (min-colors 88) (background light))
     :background "darkseagreen2")
    (((class color) (min-colors 88) (background dark))
     :background "darkolivegreen")
    (((class color) (min-colors 16) (background light))
     :background "darkseagreen2")
    (((class color) (min-colors 16) (background dark))
     :background "darkolivegreen")
    (((class color) (min-colors 8))
     :background "green" :foreground "black")
    (t :inverse-video t))
  "Basic face for highlighting."
  :group 'basic-faces)
```

Internally, Emacs stores the face's default specification in its `face-defface-spec` symbol property (see Section 8.4 [Property Lists], page 106, vol. 1). The `saved-face` property stores the face specification saved by the user, using the customization buffer; the `customized-face` property stores the face specification customized for the current session, but not saved; and the `theme-face` property stores an alist associating the active customization settings and Custom themes with their specifications for that face. The face's documentation string is stored in the `face-documentation` property. But normally you should not try to set any of these properties directly. See Section 14.5 [Applying Customizations], page 206, vol. 1, for the `custom-set-faces` function, which is used to apply customized face settings.

People are sometimes tempted to create variables whose values specify a face to use. In the vast majority of cases, this is not necessary; it is preferable to simply use faces directly.

`frame-background-mode` [User Option]

This option, if non-`nil`, specifies the background type to use for interpreting face definitions. If it is `dark`, then Emacs treats all frames as if they had a dark background, regardless of their actual background colors. If it is `light`, then Emacs treats all frames as if they had a light background.

38.12.2 Face Attributes

The effect of using a face is determined by a fixed set of *face attributes*. This table lists all the face attributes, their possible values, and their effects. You can specify more than one face for a given piece of text; Emacs merges the attributes of all the faces to determine how to display the text. See Section 38.12.4 [Displaying Faces], page 333.

In addition to the values given below, each face attribute can also have the value `unspecified`. This special value means the face doesn't specify that attribute. In face merging, when the first face fails to specify a particular attribute, the next face gets a chance. However, the `default` face must specify all attributes.

Some of these attributes are meaningful only on certain kinds of displays. If your display cannot handle a certain attribute, the attribute is ignored.

`:family` Font family or fontset (a string). See Section "Fonts" in *The GNU Emacs Manual*. If you specify a font family name, the wild-card characters '`*`' and '`?`' are allowed. The function `font-family-list`, described below, returns a list of available family names. See Section 38.12.11 [Fontsets], page 339, for information about fontsets.

`:foundry` The name of the *font foundry* for the font family specified by the `:family` attribute (a string). The wild-card characters '`*`' and '`?`' are allowed. See Section "Fonts" in *The GNU Emacs Manual*.

`:width` Relative character width. This should be one of the symbols `ultra-condensed`, `extra-condensed`, `condensed`, `semi-condensed`, `normal`, `semi-expanded`, `expanded`, `extra-expanded`, or `ultra-expanded`.

`:height` The height of the font. In the simplest case, this is an integer in units of 1/10 point.

The value can also be a floating point number or a function, which specifies the height relative to an *underlying face* (i.e., a face that has a lower priority in the list described in Section 38.12.4 [Displaying Faces], page 333). If the value is a floating point number, that specifies the amount by which to scale the height of the underlying face. If the value is a function, that function is called with one argument, the height of the underlying face, and returns the height of the new face. If the function is passed an integer argument, it must return an integer.

The height of the default face must be specified using an integer; floating point and function values are not allowed.

`:weight` Font weight—one of the symbols (from densest to faintest) `ultra-bold`, `extra-bold`, `bold`, `semi-bold`, `normal`, `semi-light`, `light`, `extra-light`, or `ultra-light`. On text terminals which support variable-brightness text, any weight greater than normal is displayed as extra bright, and any weight less than normal is displayed as half-bright.

`:slant` Font slant—one of the symbols `italic`, `oblique`, `normal`, `reverse-italic`, or `reverse-oblique`. On text terminals that support variable-brightness text, slanted text is displayed as half-bright.

`:foreground`
Foreground color, a string. The value can be a system-defined color name, or a hexadecimal color specification. See Section 29.20 [Color Names], page 92. On black-and-white displays, certain shades of gray are implemented by stipple patterns.

`:background`
Background color, a string. The value can be a system-defined color name, or a hexadecimal color specification. See Section 29.20 [Color Names], page 92.

`:underline`
Whether or not characters should be underlined, and in what color. If the value is `t`, underlining uses the foreground color of the face. If the value is a string, underlining uses that color. The value `nil` means do not underline.

`:overline`
: Whether or not characters should be overlined, and in what color. The value is used like that of `:underline`.

`:strike-through`
: Whether or not characters should be strike-through, and in what color. The value is used like that of `:underline`.

`:box`
: Whether or not a box should be drawn around characters, its color, the width of the box lines, and 3D appearance. Here are the possible values of the `:box` attribute, and what they mean:

  `nil`
  : Don't draw a box.

  `t`
  : Draw a box with lines of width 1, in the foreground color.

  color
  : Draw a box with lines of width 1, in color *color*.

  `(:line-width` *width* `:color` *color* `:style` *style*`)`
  : This way you can explicitly specify all aspects of the box. The value *width* specifies the width of the lines to draw; it defaults to 1. A negative width -*n* means to draw a line of width *n* that occupies the space of the underlying text, thus avoiding any increase in the character height or width.

    The value *color* specifies the color to draw with. The default is the foreground color of the face for simple boxes, and the background color of the face for 3D boxes.

    The value *style* specifies whether to draw a 3D box. If it is `released-button`, the box looks like a 3D button that is not being pressed. If it is `pressed-button`, the box looks like a 3D button that is being pressed. If it is `nil` or omitted, a plain 2D box is used.

`:inverse-video`
: Whether or not characters should be displayed in inverse video. The value should be `t` (yes) or `nil` (no).

`:stipple`
: The background stipple, a bitmap.

  The value can be a string; that should be the name of a file containing external-format X bitmap data. The file is found in the directories listed in the variable `x-bitmap-file-path`.

  Alternatively, the value can specify the bitmap directly, with a list of the form `(`*width height data*`)`. Here, *width* and *height* specify the size in pixels, and *data* is a string containing the raw bits of the bitmap, row by row. Each row occupies (*width*+7)/8 consecutive bytes in the string (which should be a unibyte string for best results). This means that each row always occupies at least one whole byte.

  If the value is `nil`, that means use no stipple pattern.

  Normally you do not need to set the stipple attribute, because it is used automatically to handle certain shades of gray.

`:font` The font used to display the face. Its value should be a font object. See Section 38.12.9 [Font Selection], page 337, for information about font objects.

When specifying this attribute using `set-face-attribute` (see Section 38.12.3 [Attribute Functions], page 330), you may also supply a font spec, a font entity, or a string. Emacs converts such values to an appropriate font object, and stores that font object as the actual attribute value. If you specify a string, the contents of the string should be a font name (see Section "Fonts" in *The GNU Emacs Manual*); if the font name is an XLFD containing wildcards, Emacs chooses the first font matching those wildcards. Specifying this attribute also changes the values of the `:family`, `:foundry`, `:width`, `:height`, `:weight`, and `:slant` attributes.

`:inherit` The name of a face from which to inherit attributes, or a list of face names. Attributes from inherited faces are merged into the face like an underlying face would be, with higher priority than underlying faces (see Section 38.12.4 [Displaying Faces], page 333). If a list of faces is used, attributes from faces earlier in the list override those from later faces.

For compatibility with Emacs 20, you can also specify values for two "fake" face attributes: `:bold` and `:italic`. Their values must be either `t` or `nil`; a value of `unspecified` is not allowed. Setting `:bold` to `t` is equivalent to setting the `:weight` attribute to `bold`, and setting it to `nil` is equivalent to setting `:weight` to `normal`. Setting `:italic` to `t` is equivalent to setting the `:slant` attribute to `italic`, and setting it to `nil` is equivalent to setting `:slant` to `normal`.

`font-family-list` **&optional** *frame* [Function]
This function returns a list of available font family names. The optional argument *frame* specifies the frame on which the text is to be displayed; if it is `nil`, the selected frame is used.

`underline-minimum-offset` [User Option]
This variable specifies the minimum distance between the baseline and the underline, in pixels, when displaying underlined text.

`x-bitmap-file-path` [User Option]
This variable specifies a list of directories for searching for bitmap files, for the `:stipple` attribute.

`bitmap-spec-p` *object* [Function]
This returns `t` if *object* is a valid bitmap specification, suitable for use with `:stipple` (see above). It returns `nil` otherwise.

38.12.3 Face Attribute Functions

This section describes the functions for accessing and modifying the attributes of an existing face.

`set-face-attribute` *face frame* **&rest** *arguments* [Function]
This function sets one or more attributes of *face* for *frame*. The attributes you specify this way override whatever the `defface` says.

The extra arguments *arguments* specify the attributes to set, and the values for them. They should consist of alternating attribute names (such as `:family` or `:underline`) and values. Thus,

```
(set-face-attribute 'foo nil
                    :width 'extended
                    :weight 'bold)
```

sets the attribute `:width` to `extended` and the attribute `:weight` to `bold`.

If *frame* is `t`, this function sets the default attributes for new frames. Default attribute values specified this way override the `defface` for newly created frames.

If *frame* is `nil`, this function sets the attributes for all existing frames, and the default for new frames.

`face-attribute` *face attribute* **&optional** *frame inherit* [Function]

This returns the value of the *attribute* attribute of *face* on *frame*. If *frame* is `nil`, that means the selected frame (see Section 29.9 [Input Focus], page 83).

If *frame* is `t`, this returns whatever new-frames default value you previously specified with `set-face-attribute` for the *attribute* attribute of *face*. If you have not specified one, it returns `nil`.

If *inherit* is `nil`, only attributes directly defined by *face* are considered, so the return value may be `unspecified`, or a relative value. If *inherit* is non-`nil`, *face*'s definition of *attribute* is merged with the faces specified by its `:inherit` attribute; however the return value may still be `unspecified` or relative. If *inherit* is a face or a list of faces, then the result is further merged with that face (or faces), until it becomes specified and absolute.

To ensure that the return value is always specified and absolute, use a value of `default` for *inherit*; this will resolve any unspecified or relative values by merging with the `default` face (which is always completely specified).

For example,

```
(face-attribute 'bold :weight)
     ⇒ bold
```

`face-attribute-relative-p` *attribute value* [Function]

This function returns non-`nil` if *value*, when used as the value of the face attribute *attribute*, is relative. This means it would modify, rather than completely override, any value that comes from a subsequent face in the face list or that is inherited from another face.

`unspecified` is a relative value for all attributes. For `:height`, floating point and function values are also relative.

For example:

```
(face-attribute-relative-p :height 2.0)
     ⇒ t
```

`face-all-attributes` *face* **&optional** *frame* [Function]

This function returns an alist of attributes of *face*. The elements of the result are name-value pairs of the form `(attr-name . attr-value)`. Optional argument *frame*

specifies the frame whose definition of *face* to return; if omitted or `nil`, the returned value describes the default attributes of *face* for newly created frames.

`merge-face-attribute` *attribute value1 value2* [Function]
If *value1* is a relative value for the face attribute *attribute*, returns it merged with the underlying value *value2*; otherwise, if *value1* is an absolute value for the face attribute *attribute*, returns *value1* unchanged.

The following commands and functions mostly provide compatibility with old versions of Emacs. They work by calling `set-face-attribute`. Values of `t` and `nil` for their *frame* argument are handled just like `set-face-attribute` and `face-attribute`. The commands read their arguments using the minibuffer, if called interactively.

`set-face-foreground` *face color* **&optional** *frame* [Command]
`set-face-background` *face color* **&optional** *frame* [Command]
These set the `:foreground` attribute (or `:background` attribute, respectively) of *face* to *color*.

`set-face-stipple` *face pattern* **&optional** *frame* [Command]
This sets the `:stipple` attribute of *face* to *pattern*.

`set-face-font` *face font* **&optional** *frame* [Command]
This sets the `:font` attribute of *face* to *font*.

`set-face-bold-p` *face bold-p* **&optional** *frame* [Function]
This sets the `:weight` attribute of *face* to *normal* if *bold-p* is `nil`, and to *bold* otherwise.

`set-face-italic-p` *face italic-p* **&optional** *frame* [Function]
This sets the `:slant` attribute of *face* to *normal* if *italic-p* is `nil`, and to *italic* otherwise.

`set-face-underline-p` *face underline* **&optional** *frame* [Function]
This sets the `:underline` attribute of *face* to *underline*.

`set-face-inverse-video-p` *face inverse-video-p* **&optional** *frame* [Function]
This sets the `:inverse-video` attribute of *face* to *inverse-video-p*.

`invert-face` *face* **&optional** *frame* [Command]
This swaps the foreground and background colors of face *face*.

The following functions examine the attributes of a face. If you don't specify *frame*, they refer to the selected frame; `t` refers to the default data for new frames. They return the symbol `unspecified` if the face doesn't define any value for that attribute.

`face-foreground` *face* **&optional** *frame inherit* [Function]
`face-background` *face* **&optional** *frame inherit* [Function]
These functions return the foreground color (or background color, respectively) of face *face*, as a string.

If *inherit* is `nil`, only a color directly defined by the face is returned. If *inherit* is non-`nil`, any faces specified by its `:inherit` attribute are considered as well, and if

inherit is a face or a list of faces, then they are also considered, until a specified color is found. To ensure that the return value is always specified, use a value of `default` for *inherit*.

`face-stipple` *face* **&optional** *frame inherit* [Function]
This function returns the name of the background stipple pattern of face *face*, or `nil` if it doesn't have one.

If *inherit* is `nil`, only a stipple directly defined by the face is returned. If *inherit* is non-`nil`, any faces specified by its `:inherit` attribute are considered as well, and if *inherit* is a face or a list of faces, then they are also considered, until a specified stipple is found. To ensure that the return value is always specified, use a value of `default` for *inherit*.

`face-font` *face* **&optional** *frame* [Function]
This function returns the name of the font of face *face*.

`face-bold-p` *face* **&optional** *frame* [Function]
This function returns a non-`nil` value if the `:weight` attribute of *face* is bolder than normal (i.e., one of `semi-bold`, `bold`, `extra-bold`, or `ultra-bold`). Otherwise, it returns `nil`.

`face-italic-p` *face* **&optional** *frame* [Function]
This function returns a non-`nil` value if the `:slant` attribute of *face* is `italic` or `oblique`, and `nil` otherwise.

`face-underline-p` *face* **&optional** *frame* [Function]
This function returns the `:underline` attribute of face *face*.

`face-inverse-video-p` *face* **&optional** *frame* [Function]
This function returns the `:inverse-video` attribute of face *face*.

38.12.4 Displaying Faces

Here is how Emacs determines the face to use for displaying any given piece of text:

- If the text consists of a special glyph, the glyph can specify a particular face. See Section 38.20.4 [Glyphs], page 379.
- If the text lies within an active region, Emacs highlights it using the `region` face. See Section "Standard Faces" in *The GNU Emacs Manual*.
- If the text lies within an overlay with a non-`nil` `face` property, Emacs applies the face or face attributes specified by that property. If the overlay has a `mouse-face` property and the mouse is "near enough" to the overlay, Emacs applies the face or face attributes specified by the `mouse-face` property instead. See Section 38.9.2 [Overlay Properties], page 318.

  When multiple overlays cover one character, an overlay with higher priority overrides those with lower priority. See Section 38.9 [Overlays], page 315.
- If the text contains a `face` or `mouse-face` property, Emacs applies the specified faces and face attributes. See Section 32.19.4 [Special Properties], page 162. (This is how Font Lock mode faces are applied. See Section 23.6 [Font Lock Mode], page 429, vol. 1.)

- If the text lies within the mode line of the selected window, Emacs applies the `mode-line` face. For the mode line of a non-selected window, Emacs applies the `mode-line-inactive` face. For a header line, Emacs applies the `header-line` face.
- If any given attribute has not been specified during the preceding steps, Emacs applies the attribute of the `default` face.

If these various sources together specify more than one face for a particular character, Emacs merges the attributes of the various faces specified. For each attribute, Emacs tries using the above order (i.e. first the face of any special glyph; then the face for region highlighting, if appropriate; and so on).

38.12.5 Face Remapping

The variable `face-remapping-alist` is used for buffer-local or global changes in the appearance of a face. For instance, it is used to implement the `text-scale-adjust` command (see Section "Text Scale" in *The GNU Emacs Manual*).

`face-remapping-alist` [Variable]

The value of this variable is an alist whose elements have the form `(face . remapping)`. This causes Emacs to display any text having the face *face* with *remapping*, rather than the ordinary definition of *face*. *remapping* may be any face specification suitable for a `face` text property: either a face name, or a property list of attribute/value pairs, or a list in which each element is either a face name or a property list (see Section 32.19.4 [Special Properties], page 162).

If `face-remapping-alist` is buffer-local, its local value takes effect only within that buffer.

Two points bear emphasizing:

1. *remapping* serves as the complete specification for the remapped face—it replaces the normal definition of *face*, instead of modifying it.
2. If *remapping* references the same face name *face*, either directly or via the `:inherit` attribute of some other face in *remapping*, that reference uses the normal definition of *face*. In other words, the remapping cannot be recursive.

   For instance, if the `mode-line` face is remapped using this entry in `face-remapping-alist`:

   ```
   (mode-line italic mode-line)
   ```

   then the new definition of the `mode-line` face inherits from the `italic` face, and the *normal* (non-remapped) definition of `mode-line` face.

The following functions implement a higher-level interface to `face-remapping-alist`. Most Lisp code should use these functions instead of setting `face-remapping-alist` directly, to avoid trampling on remappings applied elsewhere. These functions are intended for buffer-local remappings, so they all make `face-remapping-alist` buffer-local as a side-effect. They manage `face-remapping-alist` entries of the form

```
(face relative-spec-1 relative-spec-2 ... base-spec)
```

where, as explained above, each of the *relative-spec-N* and *base-spec* is either a face name, or a property list of attribute/value pairs. Each of the *relative remapping* entries, *relative-spec-N*, is managed by the `face-remap-add-relative` and `face-remap-remove-relative`

functions; these are intended for simple modifications like changing the text size. The *base remapping* entry, *base-spec*, has the lowest priority and is managed by the `face-remap-set-base` and `face-remap-reset-base` functions; it is intended for major modes to remap faces in the buffers they control.

`face-remap-add-relative` *face* **&rest** *specs* [Function]

This functions adds the face specifications in *specs* as relative remappings for face *face* in the current buffer. The remaining arguments, *specs*, should form either a list of face names, or a property list of attribute/value pairs.

The return value is a Lisp object that serves as a "cookie"; you can pass this object as an argument to `face-remap-remove-relative` if you need to remove the remapping later.

```
;; Remap the 'escape-glyph' face into a combination
;; of the 'highlight' and 'italic' faces:
(face-remap-add-relative 'escape-glyph 'highlight 'italic)

;; Increase the size of the 'default' face by 50%:
(face-remap-add-relative 'default :height 1.5)
```

`face-remap-remove-relative` *cookie* [Function]

This function removes a relative remapping previously added by `face-remap-add-relative`. *cookie* should be the Lisp object returned by `face-remap-add-relative` when the remapping was added.

`face-remap-set-base` *face* **&rest** *specs* [Function]

This function sets the base remapping of *face* in the current buffer to *specs*. If *specs* is empty, the default base remapping is restored, similar to calling `face-remap-reset-base` (see below); note that this is different from *specs* containing a single value `nil`, which has the opposite result (the global definition of *face* is ignored).

This overwrites the default *base-spec*, which inherits the global face definition, so it is up to the caller to add such inheritance if so desired.

`face-remap-reset-base` *face* [Function]

This function sets the base remapping of *face* to its default value, which inherits from *face*'s global definition.

38.12.6 Functions for Working with Faces

Here are additional functions for creating and working with faces.

`face-list` [Function]

This function returns a list of all defined face names.

`face-id` *face* [Function]

This function returns the *face number* of face *face*. This is a number that uniquely identifies a face at low levels within Emacs. It is seldom necessary to refer to a face by its face number.

`face-documentation` *face* [Function]

This function returns the documentation string of face *face*, or `nil` if none was specified for it.

`face-equal` *face1 face2* **&optional** *frame* [Function]
This returns `t` if the faces *face1* and *face2* have the same attributes for display.

`face-differs-from-default-p` *face* **&optional** *frame* [Function]
This returns non-`nil` if the face *face* displays differently from the default face.

A *face alias* provides an equivalent name for a face. You can define a face alias by giving the alias symbol the `face-alias` property, with a value of the target face name. The following example makes `modeline` an alias for the `mode-line` face.

```
(put 'modeline 'face-alias 'mode-line)
```

`define-obsolete-face-alias` *obsolete-face current-face when* [Macro]
This macro defines `obsolete-face` as an alias for *current-face*, and also marks it as obsolete, indicating that it may be removed in future. *when* should be a string indicating when `obsolete-face` was made obsolete (usually a version number string).

38.12.7 Automatic Face Assignment

This hook is used for automatically assigning faces to text in the buffer. It is part of the implementation of Jit-Lock mode, used by Font-Lock.

`fontification-functions` [Variable]
This variable holds a list of functions that are called by Emacs redisplay as needed, just before doing redisplay. They are called even when Font Lock Mode isn't enabled. When Font Lock Mode is enabled, this variable usually holds just one function, `jit-lock-function`.

The functions are called in the order listed, with one argument, a buffer position *pos*. Collectively they should attempt to assign faces to the text in the current buffer starting at *pos*.

The functions should record the faces they assign by setting the `face` property. They should also add a non-`nil` `fontified` property to all the text they have assigned faces to. That property tells redisplay that faces have been assigned to that text already.

It is probably a good idea for the functions to do nothing if the character after *pos* already has a non-`nil` `fontified` property, but this is not required. If one function overrides the assignments made by a previous one, the properties after the last function finishes are the ones that really matter.

For efficiency, we recommend writing these functions so that they usually assign faces to around 400 to 600 characters at each call.

38.12.8 Basic Faces

If your Emacs Lisp program needs to assign some faces to text, it is often a good idea to use certain existing faces or inherit from them, rather than defining entirely new faces. This way, if other users have customized the basic faces to give Emacs a certain look, your program will "fit in" without additional customization.

Some of the basic faces defined in Emacs are listed below. In addition to these, you might want to make use of the Font Lock faces for syntactic highlighting, if highlighting is not already handled by Font Lock mode, or if some Font Lock faces are not in use. See Section 23.6.7 [Faces for Font Lock], page 436, vol. 1.

`default` The default face, whose attributes are all specified. All other faces implicitly inherit from it: any unspecified attribute defaults to the attribute on this face (see Section 38.12.2 [Face Attributes], page 327).

`bold`
`italic`
`bold-italic`
`underline`
`fixed-pitch`
`variable-pitch`
These have the attributes indicated by their names (e.g. `bold` has a bold `:weight` attribute), with all other attributes unspecified (and so given by `default`).

`shadow` For "dimmed out" text. For example, it is used for the ignored part of a filename in the minibuffer (see Section "Minibuffers for File Names" in *The GNU Emacs Manual*).

`link`
`link-visited`
For clickable text buttons that send the user to a different buffer or "location".

`highlight`
For stretches of text that should temporarily stand out. For example, it is commonly assigned to the `mouse-face` property for cursor highlighting (see Section 32.19.4 [Special Properties], page 162).

`match` For text matching a search command.

`error`
`warning`
`success` For text concerning errors, warnings, or successes. For example, these are used for messages in '`*Compilation*`' buffers.

38.12.9 Font Selection

Before Emacs can draw a character on a graphical display, it must select a *font* for that character[1]. See Section "Fonts" in *The GNU Emacs Manual*. Normally, Emacs automatically chooses a font based on the faces assigned to that character—specifically, the face attributes `:family`, `:weight`, `:slant`, and `:width` (see Section 38.12.2 [Face Attributes], page 327). The choice of font also depends on the character to be displayed; some fonts can only display a limited set of characters. If no available font exactly fits the requirements, Emacs looks for the *closest matching font*. The variables in this section control how Emacs makes this selection.

`face-font-family-alternatives` [User Option]
If a given family is specified but does not exist, this variable specifies alternative font families to try. Each element should have this form:

[1] In this context, the term *font* has nothing to do with Font Lock (see Section 23.6 [Font Lock Mode], page 429, vol. 1).

```
(family alternate-families...)
```

If *family* is specified but not available, Emacs will try the other families given in *alternate-families*, one by one, until it finds a family that does exist.

`face-font-selection-order` [User Option]

If there is no font that exactly matches all desired face attributes (`:width`, `:height`, `:weight`, and `:slant`), this variable specifies the order in which these attributes should be considered when selecting the closest matching font. The value should be a list containing those four attribute symbols, in order of decreasing importance. The default is `(:width :height :weight :slant)`.

Font selection first finds the best available matches for the first attribute in the list; then, among the fonts which are best in that way, it searches for the best matches in the second attribute, and so on.

The attributes `:weight` and `:width` have symbolic values in a range centered around `normal`. Matches that are more extreme (farther from `normal`) are somewhat preferred to matches that are less extreme (closer to `normal`); this is designed to ensure that non-normal faces contrast with normal ones, whenever possible.

One example of a case where this variable makes a difference is when the default font has no italic equivalent. With the default ordering, the `italic` face will use a non-italic font that is similar to the default one. But if you put `:slant` before `:height`, the `italic` face will use an italic font, even if its height is not quite right.

`face-font-registry-alternatives` [User Option]

This variable lets you specify alternative font registries to try, if a given registry is specified and doesn't exist. Each element should have this form:

```
(registry alternate-registries...)
```

If *registry* is specified but not available, Emacs will try the other registries given in *alternate-registries*, one by one, until it finds a registry that does exist.

Emacs can make use of scalable fonts, but by default it does not use them.

`scalable-fonts-allowed` [User Option]

This variable controls which scalable fonts to use. A value of `nil`, the default, means do not use scalable fonts. `t` means to use any scalable font that seems appropriate for the text.

Otherwise, the value must be a list of regular expressions. Then a scalable font is enabled for use if its name matches any regular expression in the list. For example,

```
(setq scalable-fonts-allowed '("muleindian-2$"))
```

allows the use of scalable fonts with registry `muleindian-2`.

`face-font-rescale-alist` [Variable]

This variable specifies scaling for certain faces. Its value should be a list of elements of the form

```
(fontname-regexp . scale-factor)
```

If *fontname-regexp* matches the font name that is about to be used, this says to choose a larger similar font according to the factor *scale-factor*. You would use this feature to normalize the font size if certain fonts are bigger or smaller than their nominal heights and widths would suggest.

38.12.10 Looking Up Fonts

`x-list-fonts` *name* **&optional** *reference-face frame maximum width* [Function]

This function returns a list of available font names that match *name*. *name* should be a string containing a font name in either the Fontconfig, GTK, or XLFD format (see Section "Fonts" in *The GNU Emacs Manual*). Within an XLFD string, wildcard characters may be used: the '`*`' character matches any substring, and the '`?`' character matches any single character. Case is ignored when matching font names.

If the optional arguments *reference-face* and *frame* are specified, the returned list includes only fonts that are the same size as *reference-face* (a face name) currently is on the frame *frame*.

The optional argument *maximum* sets a limit on how many fonts to return. If it is non-`nil`, then the return value is truncated after the first *maximum* matching fonts. Specifying a small value for *maximum* can make this function much faster, in cases where many fonts match the pattern.

The optional argument *width* specifies a desired font width. If it is non-`nil`, the function only returns those fonts whose characters are (on average) *width* times as wide as *reference-face*.

`x-family-fonts` **&optional** *family frame* [Function]

This function returns a list describing the available fonts for family *family* on *frame*. If *family* is omitted or `nil`, this list applies to all families, and therefore, it contains all available fonts. Otherwise, *family* must be a string; it may contain the wildcards '`?`' and '`*`'.

The list describes the display that *frame* is on; if *frame* is omitted or `nil`, it applies to the selected frame's display (see Section 29.9 [Input Focus], page 83).

Each element in the list is a vector of the following form:

```
[family width point-size weight slant
 fixed-p full registry-and-encoding]
```

The first five elements correspond to face attributes; if you specify these attributes for a face, it will use this font.

The last three elements give additional information about the font. *fixed-p* is non-`nil` if the font is fixed-pitch. *full* is the full name of the font, and *registry-and-encoding* is a string giving the registry and encoding of the font.

`font-list-limit` [Variable]

This variable specifies maximum number of fonts to consider in font matching. The function `x-family-fonts` will not return more than that many fonts, and font selection will consider only that many fonts when searching a matching font for face attributes. The default is currently 100.

38.12.11 Fontsets

A *fontset* is a list of fonts, each assigned to a range of character codes. An individual font cannot display the whole range of characters that Emacs supports, but a fontset can. Fontsets have names, just as fonts do, and you can use a fontset name in place of a font name when you specify the "font" for a frame or a face. Here is information about defining a fontset under Lisp program control.

`create-fontset-from-fontset-spec` *fontset-spec* **&optional** [Function]
style-variant-p noerror

This function defines a new fontset according to the specification string *fontset-spec*. The string should have this format:

fontpattern, [*charset*:*font*]...

Whitespace characters before and after the commas are ignored.

The first part of the string, *fontpattern*, should have the form of a standard X font name, except that the last two fields should be '`fontset-`*alias*'.

The new fontset has two names, one long and one short. The long name is *fontpattern* in its entirety. The short name is '`fontset-`*alias*'. You can refer to the fontset by either name. If a fontset with the same name already exists, an error is signaled, unless *noerror* is non-`nil`, in which case this function does nothing.

If optional argument *style-variant-p* is non-`nil`, that says to create bold, italic and bold-italic variants of the fontset as well. These variant fontsets do not have a short name, only a long one, which is made by altering *fontpattern* to indicate the bold or italic status.

The specification string also says which fonts to use in the fontset. See below for the details.

The construct '*charset*:*font*' specifies which font to use (in this fontset) for one particular character set. Here, *charset* is the name of a character set, and *font* is the font to use for that character set. You can use this construct any number of times in the specification string.

For the remaining character sets, those that you don't specify explicitly, Emacs chooses a font based on *fontpattern*: it replaces '`fontset-`*alias*' with a value that names one character set. For the ASCII character set, '`fontset-`*alias*' is replaced with '`ISO8859-1`'.

In addition, when several consecutive fields are wildcards, Emacs collapses them into a single wildcard. This is to prevent use of auto-scaled fonts. Fonts made by scaling larger fonts are not usable for editing, and scaling a smaller font is not useful because it is better to use the smaller font in its own size, which Emacs does.

Thus if *fontpattern* is this,

```
-*-fixed-medium-r-normal-*-24-*-*-*-*-*-fontset-24
```

the font specification for ASCII characters would be this:

```
-*-fixed-medium-r-normal-*-24-*-ISO8859-1
```

and the font specification for Chinese GB2312 characters would be this:

```
-*-fixed-medium-r-normal-*-24-*-gb2312*-*
```

You may not have any Chinese font matching the above font specification. Most X distributions include only Chinese fonts that have '`song ti`' or '`fangsong ti`' in the *family* field. In such a case, '`Fontset-`*n*' can be specified as below:

```
Emacs.Fontset-0: -*-fixed-medium-r-normal-*-24-*-*-*-*-*-fontset-24,\
        chinese-gb2312:-*-*-medium-r-normal-*-24-*-gb2312*-*
```

Then, the font specifications for all but Chinese GB2312 characters have '`fixed`' in the *family* field, and the font specification for Chinese GB2312 characters has a wild card '`*`' in the *family* field.

`set-fontset-font` *name character font-spec* **&optional** *frame add* [Function]
This function modifies the existing fontset *name* to use the font matching with *font-spec* for the character *character*.

If *name* is `nil`, this function modifies the fontset of the selected frame or that of *frame* if *frame* is not `nil`.

If *name* is `t`, this function modifies the default fontset, whose short name is '`fontset-default`'.

character may be a cons; `(`*from* `.` *to*`)`, where *from* and *to* are character codepoints. In that case, use *font-spec* for all characters in the range *from* and *to* (inclusive).

character may be a charset. In that case, use *font-spec* for all character in the charsets.

character may be a script name. In that case, use *font-spec* for all character in the charsets.

font-spec may be a cons; `(`*family* `.` *registry*`)`, where *family* is a family name of a font (possibly including a foundry name at the head), *registry* is a registry name of a font (possibly including an encoding name at the tail).

font-spec may be a font name string.

The optional argument *add*, if non-`nil`, specifies how to add *font-spec* to the font specifications previously set. If it is `prepend`, *font-spec* is prepended. If it is `append`, *font-spec* is appended. By default, *font-spec* overrides the previous settings.

For instance, this changes the default fontset to use a font of which family name is '`Kochi Gothic`' for all characters belonging to the charset `japanese-jisx0208`.

```
(set-fontset-font t 'japanese-jisx0208
                  (font-spec :family "Kochi Gothic"))
```

`char-displayable-p` *char* [Function]
This function returns `t` if Emacs ought to be able to display *char*. More precisely, if the selected frame's fontset has a font to display the character set that *char* belongs to.

Fontsets can specify a font on a per-character basis; when the fontset does that, this function's value may not be accurate.

38.12.12 Low-Level Font Representation

Normally, it is not necessary to manipulate fonts directly. In case you need to do so, this section explains how.

In Emacs Lisp, fonts are represented using three different Lisp object types: *font objects*, *font specs*, and *font entities*.

`fontp` *object* **&optional** *type* [Function]
Return `t` if *object* is a font object, font spec, or font entity. Otherwise, return `nil`.

The optional argument *type*, if non-`nil`, determines the exact type of Lisp object to check for. In that case, *type* should be one of `font-object`, `font-spec`, or `font-entity`.

A font object is a Lisp object that represents a font that Emacs has *opened*. Font objects cannot be modified in Lisp, but they can be inspected.

`font-at` *position* **&optional** *window string* [Function]
Return the font object that is being used to display the character at position *position* in the window *window*. If *window* is `nil`, it defaults to the selected window. If *string* is `nil`, *position* specifies a position in the current buffer; otherwise, *string* should be a string, and *position* specifies a position in that string.

A font spec is a Lisp object that contains a set of specifications that can be used to find a font. More than one font may match the specifications in a font spec.

`font-spec` **&rest** *arguments* [Function]
Return a new font spec using the specifications in *arguments*, which should come in `property-value` pairs. The possible specifications are as follows:

`:name` The font name (a string), in either XLFD, Fontconfig, or GTK format. See Section "Fonts" in *The GNU Emacs Manual*.

`:family`
`:foundry`
`:weight`
`:slant`
`:width` These have the same meanings as the face attributes of the same name. See Section 38.12.2 [Face Attributes], page 327.

`:size` The font size—either a non-negative integer that specifies the pixel size, or a floating point number that specifies the point size.

`:adstyle` Additional typographic style information for the font, such as '`sans`'. The value should be a string or a symbol.

`:registry`
The charset registry and encoding of the font, such as '`iso8859-1`'. The value should be a string or a symbol.

`:script` The script that the font must support (a symbol).

`:otf` The font must be an OpenType font that supports these OpenType features, provided Emacs is compiled with support for '`libotf`' (a library for performing complex text layout in certain scripts). The value must be a list of the form

```
(script-tag langsys-tag gsub gpos)
```

where *script-tag* is the OpenType script tag symbol; *langsys-tag* is the OpenType language system tag symbol, or `nil` to use the default language system; `gsub` is a list of OpenType GSUB feature tag symbols, or `nil` if none is required; and `gpos` is a list of OpenType GPOS feature tag symbols, or `nil` if none is required. If `gsub` or `gpos` is a list, a `nil` element in that list means that the font must not match any of the remaining tag symbols. The `gpos` element may be omitted.

`font-put` *font-spec property value* [Function]
Set the font property *property* in the font-spec *font-spec* to *value*.

A font entity is a reference to a font that need not be open. Its properties are intermediate between a font object and a font spec: like a font object, and unlike a font spec, it refers to a single, specific font. Unlike a font object, creating a font entity does not load the contents of that font into computer memory.

`find-font` *font-spec* **&optional** *frame* [Function]
: This function returns a font entity that best matches the font spec *font-spec* on frame *frame*. If *frame* is `nil`, it defaults to the selected frame.

`list-fonts` *font-spec* **&optional** *frame num prefer* [Function]
: This function returns a list of all font entities that match the font spec *font-spec*.

  The optional argument *frame*, if non-`nil`, specifies the frame on which the fonts are to be displayed. The optional argument *num*, if non-`nil`, should be an integer that specifies the maximum length of the returned list. The optional argument *prefer*, if non-`nil`, should be another font spec, which is used to control the order of the returned list; the returned font entities are sorted in order of decreasing "closeness" to that font spec.

If you call `set-face-attribute` and pass a font spec, font entity, or font name string as the value of the `:font` attribute, Emacs opens the best "matching" font that is available for display. It then stores the corresponding font object as the actual value of the `:font` attribute for that face.

The following functions can be used to obtain information about a font. For these functions, the *font* argument can be a font object, a font entity, or a font spec.

`font-get` *font property* [Function]
: This function returns the value of the font property *property* for *font*.

  If *font* is a font spec and the font spec does not specify *property*, the return value is `nil`. If *font* is a font object or font entity, the value for the *:script* property may be a list of scripts supported by the font.

`font-face-attributes` *font* **&optional** *frame* [Function]
: This function returns a list of face attributes corresponding to *font*. The optional argument *frame* specifies the frame on which the font is to be displayed. If it is `nil`, the selected frame is used. The return value has the form

```
(:family family :height height :weight weight
   :slant slant :width width)
```

  where the values of *family*, *height*, *weight*, *slant*, and *width* are face attribute values. Some of these key-attribute pairs may be omitted from the list if they are not specified by *font*.

`font-xlfd-name` *font* **&optional** *fold-wildcards* [Function]
: This function returns the XLFD (X Logical Font Descriptor), a string, matching *font*. See Section "Fonts" in *The GNU Emacs Manual*, for information about XLFDs. If the name is too long for an XLFD (which can contain at most 255 characters), the function returns `nil`.

  If the optional argument *fold-wildcards* is non-`nil`, consecutive wildcards in the XLFD are folded into one.

38.13 Fringes

On graphical displays, Emacs draws *fringes* next to each window: thin vertical strips down the sides which can display bitmaps indicating truncation, continuation, horizontal scrolling, and so on.

38.13.1 Fringe Size and Position

The following buffer-local variables control the position and width of fringes in windows showing that buffer.

`fringes-outside-margins` [Variable]
: The fringes normally appear between the display margins and the window text. If the value is non-`nil`, they appear outside the display margins. See Section 38.15.5 [Display Margins], page 354.

`left-fringe-width` [Variable]
: This variable, if non-`nil`, specifies the width of the left fringe in pixels. A value of `nil` means to use the left fringe width from the window's frame.

`right-fringe-width` [Variable]
: This variable, if non-`nil`, specifies the width of the right fringe in pixels. A value of `nil` means to use the right fringe width from the window's frame.

Any buffer which does not specify values for these variables uses the values specified by the `left-fringe` and `right-fringe` frame parameters (see Section 29.3.3.4 [Layout Parameters], page 74).

The above variables actually take effect via the function `set-window-buffer` (see Section 28.9 [Buffers and Windows], page 36), which calls `set-window-fringes` as a subroutine. If you change one of these variables, the fringe display is not updated in existing windows showing the buffer, unless you call `set-window-buffer` again in each affected window. You can also use `set-window-fringes` to control the fringe display in individual windows.

`set-window-fringes` *window left* **&optional** *right outside-margins* [Function]
: This function sets the fringe widths of window *window*. If *window* is `nil`, the selected window is used.

  The argument *left* specifies the width in pixels of the left fringe, and likewise *right* for the right fringe. A value of `nil` for either one stands for the default width. If *outside-margins* is non-`nil`, that specifies that fringes should appear outside of the display margins.

`window-fringes` **&optional** *window* [Function]
: This function returns information about the fringes of a window *window*. If *window* is omitted or `nil`, the selected window is used. The value has the form `(left-width right-width outside-margins)`.

38.13.2 Fringe Indicators

Fringe indicators are tiny icons displayed in the window fringe to indicate truncated or continued lines, buffer boundaries, etc.

`indicate-empty-lines` [User Option]

When this is non-`nil`, Emacs displays a special glyph in the fringe of each empty line at the end of the buffer, on graphical displays. See Section 38.13 [Fringes], page 344. This variable is automatically buffer-local in every buffer.

`indicate-buffer-boundaries` [User Option]

This buffer-local variable controls how the buffer boundaries and window scrolling are indicated in the window fringes.

Emacs can indicate the buffer boundaries—that is, the first and last line in the buffer—with angle icons when they appear on the screen. In addition, Emacs can display an up-arrow in the fringe to show that there is text above the screen, and a down-arrow to show there is text below the screen.

There are three kinds of basic values:

`nil` Don't display any of these fringe icons.

`left` Display the angle icons and arrows in the left fringe.

`right` Display the angle icons and arrows in the right fringe.

any non-alist
Display the angle icons in the left fringe and don't display the arrows.

Otherwise the value should be an alist that specifies which fringe indicators to display and where. Each element of the alist should have the form `(`*`indicator`* `.` *`position`*`)`. Here, *indicator* is one of `top`, `bottom`, `up`, `down`, and `t` (which covers all the icons not yet specified), while *position* is one of `left`, `right` and `nil`.

For example, `((top . left) (t . right))` places the top angle bitmap in left fringe, and the bottom angle bitmap as well as both arrow bitmaps in right fringe. To show the angle bitmaps in the left fringe, and no arrow bitmaps, use `((top . left) (bottom . left))`.

`fringe-indicator-alist` [Variable]

This buffer-local variable specifies the mapping from logical fringe indicators to the actual bitmaps displayed in the window fringes. The value is an alist of elements `(`*`indicator`* `.` *`bitmaps`*`)`, where *indicator* specifies a logical indicator type and *bitmaps* specifies the fringe bitmaps to use for that indicator.

Each *indicator* should be one of the following symbols:

`truncation`, `continuation`.
Used for truncation and continuation lines.

`up`, `down`, `top`, `bottom`, `top-bottom`
Used when `indicate-buffer-boundaries` is non-`nil`: `up` and `down` indicate a buffer boundary lying above or below the window edge; `top` and `bottom` indicate the topmost and bottommost buffer text line; and `top-bottom` indicates where there is just one line of text in the buffer.

`empty-line`
Used to indicate empty lines when `indicate-empty-lines` is non-`nil`.

`overlay-arrow`
: Used for overlay arrows (see Section 38.13.6 [Overlay Arrow], page 348).

Each *bitmaps* value may be a list of symbols `(left right [left1 right1])`. The *left* and *right* symbols specify the bitmaps shown in the left and/or right fringe, for the specific indicator. *left1* and *right1* are specific to the `bottom` and `top-bottom` indicators, and are used to indicate that the last text line has no final newline. Alternatively, *bitmaps* may be a single symbol which is used in both left and right fringes.

See Section 38.13.4 [Fringe Bitmaps], page 346, for a list of standard bitmap symbols and how to define your own. In addition, `nil` represents the empty bitmap (i.e. an indicator that is not shown).

When `fringe-indicator-alist` has a buffer-local value, and there is no bitmap defined for a logical indicator, or the bitmap is `t`, the corresponding value from the default value of `fringe-indicator-alist` is used.

38.13.3 Fringe Cursors

When a line is exactly as wide as the window, Emacs displays the cursor in the right fringe instead of using two lines. Different bitmaps are used to represent the cursor in the fringe depending on the current buffer's cursor type.

`overflow-newline-into-fringe` [User Option]
: If this is non-`nil`, lines exactly as wide as the window (not counting the final newline character) are not continued. Instead, when point is at the end of the line, the cursor appears in the right fringe.

`fringe-cursor-alist` [Variable]
: This variable specifies the mapping from logical cursor type to the actual fringe bitmaps displayed in the right fringe. The value is an alist where each element has the form `(cursor-type . bitmap)`, which means to use the fringe bitmap *bitmap* to display cursors of type *cursor-type*.

Each *cursor-type* should be one of `box`, `hollow`, `bar`, `hbar`, or `hollow-small`. The first four have the same meanings as in the `cursor-type` frame parameter (see Section 29.3.3.7 [Cursor Parameters], page 76). The `hollow-small` type is used instead of `hollow` when the normal `hollow-rectangle` bitmap is too tall to fit on a specific display line.

Each *bitmap* should be a symbol specifying the fringe bitmap to be displayed for that logical cursor type. See the next subsection for details.

When `fringe-cursor-alist` has a buffer-local value, and there is no bitmap defined for a cursor type, the corresponding value from the default value of `fringes-indicator-alist` is used.

38.13.4 Fringe Bitmaps

The *fringe bitmaps* are the actual bitmaps which represent the logical fringe indicators for truncated or continued lines, buffer boundaries, overlay arrows, etc. Each bitmap is represented by a symbol. These symbols are referred to by the variables `fringe-indicator-alist` and `fringe-cursor-alist`, described in the previous subsections.

Lisp programs can also directly display a bitmap in the left or right fringe, by using a `display` property for one of the characters appearing in the line (see Section 38.15.4 [Other Display Specs], page 353). Such a display specification has the form

```
(fringe bitmap [face])
```

fringe is either the symbol `left-fringe` or `right-fringe`. *bitmap* is a symbol identifying the bitmap to display. The optional *face* names a face whose foreground color is used to display the bitmap; this face is automatically merged with the `fringe` face.

Here is a list of the standard fringe bitmaps defined in Emacs, and how they are currently used in Emacs (via `fringe-indicator-alist` and `fringe-cursor-alist`):

`left-arrow`, `right-arrow`
: Used to indicate truncated lines.

`left-curly-arrow`, `right-curly-arrow`
: Used to indicate continued lines.

`right-triangle`, `left-triangle`
: The former is used by overlay arrows. The latter is unused.

`up-arrow`, `down-arrow`, `top-left-angle` `top-right-angle`
`bottom-left-angle`, `bottom-right-angle`
`top-right-angle`, `top-left-angle`
`left-bracket`, `right-bracket`, `top-right-angle`, `top-left-angle`
: Used to indicate buffer boundaries.

`filled-rectangle`, `hollow-rectangle`
`filled-square`, `hollow-square`
`vertical-bar`, `horizontal-bar`
: Used for different types of fringe cursors.

`empty-line`, `question-mark`
: Unused.

The next subsection describes how to define your own fringe bitmaps.

`fringe-bitmaps-at-pos` **&optional** *pos window* [Function]
: This function returns the fringe bitmaps of the display line containing position *pos* in window *window*. The return value has the form `(left right ov)`, where *left* is the symbol for the fringe bitmap in the left fringe (or `nil` if no bitmap), *right* is similar for the right fringe, and *ov* is non-`nil` if there is an overlay arrow in the left fringe.

  The value is `nil` if *pos* is not visible in *window*. If *window* is `nil`, that stands for the selected window. If *pos* is `nil`, that stands for the value of point in *window*.

38.13.5 Customizing Fringe Bitmaps

`define-fringe-bitmap` *bitmap bits* **&optional** *height width align* [Function]
: This function defines the symbol *bitmap* as a new fringe bitmap, or replaces an existing bitmap with that name.

  The argument *bits* specifies the image to use. It should be either a string or a vector of integers, where each element (an integer) corresponds to one row of the bitmap.

Each bit of an integer corresponds to one pixel of the bitmap, where the low bit corresponds to the rightmost pixel of the bitmap.

The height is normally the length of *bits*. However, you can specify a different height with non-`nil` *height*. The width is normally 8, but you can specify a different width with non-`nil` *width*. The width must be an integer between 1 and 16.

The argument *align* specifies the positioning of the bitmap relative to the range of rows where it is used; the default is to center the bitmap. The allowed values are `top`, `center`, or `bottom`.

The *align* argument may also be a list `(`*`align periodic`*`)` where *align* is interpreted as described above. If *periodic* is non-`nil`, it specifies that the rows in `bits` should be repeated enough times to reach the specified height.

`destroy-fringe-bitmap` *bitmap* [Function]
This function destroy the fringe bitmap identified by *bitmap*. If *bitmap* identifies a standard fringe bitmap, it actually restores the standard definition of that bitmap, instead of eliminating it entirely.

`set-fringe-bitmap-face` *bitmap* **&optional** *face* [Function]
This sets the face for the fringe bitmap *bitmap* to *face*. If *face* is `nil`, it selects the `fringe` face. The bitmap's face controls the color to draw it in.

face is merged with the `fringe` face, so normally *face* should specify only the foreground color.

38.13.6 The Overlay Arrow

The *overlay arrow* is useful for directing the user's attention to a particular line in a buffer. For example, in the modes used for interface to debuggers, the overlay arrow indicates the line of code about to be executed. This feature has nothing to do with *overlays* (see Section 38.9 [Overlays], page 315).

`overlay-arrow-string` [Variable]
This variable holds the string to display to call attention to a particular line, or `nil` if the arrow feature is not in use. On a graphical display the contents of the string are ignored; instead a glyph is displayed in the fringe area to the left of the display area.

`overlay-arrow-position` [Variable]
This variable holds a marker that indicates where to display the overlay arrow. It should point at the beginning of a line. On a non-graphical display the arrow text appears at the beginning of that line, overlaying any text that would otherwise appear. Since the arrow is usually short, and the line usually begins with indentation, normally nothing significant is overwritten.

The overlay-arrow string is displayed in any given buffer if the value of `overlay-arrow-position` in that buffer points into that buffer. Thus, it is possible to display multiple overlay arrow strings by creating buffer-local bindings of `overlay-arrow-position`. However, it is usually cleaner to use `overlay-arrow-variable-list` to achieve this result.

You can do a similar job by creating an overlay with a `before-string` property. See Section 38.9.2 [Overlay Properties], page 318.

You can define multiple overlay arrows via the variable `overlay-arrow-variable-list`.

`overlay-arrow-variable-list` [Variable]
: This variable's value is a list of variables, each of which specifies the position of an overlay arrow. The variable `overlay-arrow-position` has its normal meaning because it is on this list.

Each variable on this list can have properties `overlay-arrow-string` and `overlay-arrow-bitmap` that specify an overlay arrow string (for text terminals) or fringe bitmap (for graphical terminals) to display at the corresponding overlay arrow position. If either property is not set, the default `overlay-arrow-string` or `overlay-arrow` fringe indicator is used.

38.14 Scroll Bars

Normally the frame parameter `vertical-scroll-bars` controls whether the windows in the frame have vertical scroll bars, and whether they are on the left or right. The frame parameter `scroll-bar-width` specifies how wide they are (`nil` meaning the default). See Section 29.3.3.4 [Layout Parameters], page 74.

`frame-current-scroll-bars` **&optional** *frame* [Function]
: This function reports the scroll bar type settings for frame *frame*. The value is a cons cell (*vertical-type* . *horizontal-type*), where *vertical-type* is either `left`, `right`, or `nil` (which means no scroll bar.) *horizontal-type* is meant to specify the horizontal scroll bar type, but since they are not implemented, it is always `nil`.

You can enable or disable scroll bars for a particular buffer, by setting the variable `vertical-scroll-bar`. This variable automatically becomes buffer-local when set. The possible values are `left`, `right`, `t`, which means to use the frame's default, and `nil` for no scroll bar.

You can also control this for individual windows. Call the function `set-window-scroll-bars` to specify what to do for a specific window:

`set-window-scroll-bars` *window width* **&optional** *vertical-type horizontal-type* [Function]
: This function sets the width and type of scroll bars for window *window*.

  width specifies the scroll bar width in pixels (`nil` means use the width specified for the frame). *vertical-type* specifies whether to have a vertical scroll bar and, if so, where. The possible values are `left`, `right` and `nil`, just like the values of the `vertical-scroll-bars` frame parameter.

  The argument *horizontal-type* is meant to specify whether and where to have horizontal scroll bars, but since they are not implemented, it has no effect. If *window* is `nil`, the selected window is used.

`window-scroll-bars` **&optional** *window* [Function]
: Report the width and type of scroll bars specified for *window*. If *window* is omitted or `nil`, the selected window is used. The value is a list of the form (*width cols*

vertical-type horizontal-type). The value *width* is the value that was specified for the width (which may be `nil`); *cols* is the number of columns that the scroll bar actually occupies.

horizontal-type is not actually meaningful.

If you don't specify these values for a window with `set-window-scroll-bars`, the buffer-local variables `scroll-bar-mode` and `scroll-bar-width` in the buffer being displayed control the window's vertical scroll bars. The function `set-window-buffer` examines these variables. If you change them in a buffer that is already visible in a window, you can make the window take note of the new values by calling `set-window-buffer` specifying the same buffer that is already displayed.

`scroll-bar-mode` [User Option]
This variable, always local in all buffers, controls whether and where to put scroll bars in windows displaying the buffer. The possible values are `nil` for no scroll bar, `left` to put a scroll bar on the left, and `right` to put a scroll bar on the right.

`window-current-scroll-bars` **&optional** *window* [Function]
This function reports the scroll bar type for window *window*. If *window* is omitted or `nil`, the selected window is used. The value is a cons cell (*vertical-type* . *horizontal-type*). Unlike `window-scroll-bars`, this reports the scroll bar type actually used, once frame defaults and `scroll-bar-mode` are taken into account.

`scroll-bar-width` [Variable]
This variable, always local in all buffers, specifies the width of the buffer's scroll bars, measured in pixels. A value of `nil` means to use the value specified by the frame.

38.15 The `display` Property

The `display` text property (or overlay property) is used to insert images into text, and to control other aspects of how text displays. The value of the `display` property should be a display specification, or a list or vector containing several display specifications. Display specifications in the same `display` property value generally apply in parallel to the text they cover.

If several sources (overlays and/or a text property) specify values for the `display` property, only one of the values takes effect, following the rules of `get-char-property`. See Section 32.19.1 [Examining Properties], page 157.

The rest of this section describes several kinds of display specifications and what they mean.

38.15.1 Display Specs That Replace The Text

Some kinds of display specifications specify something to display instead of the text that has the property. These are called *replacing* display specifications. Emacs does not allow the user to interactively move point into the middle of buffer text that is replaced in this way.

If a list of display specifications includes more than one replacing display specification, the first overrides the rest. Replacing display specifications make most other display specifications irrelevant, since those don't apply to the replacement.

For replacing display specifications, "the text that has the property" means all the consecutive characters that have the same Lisp object as their `display` property; these characters are replaced as a single unit. If two characters have different Lisp objects as their `display` properties (i.e. objects which are not `eq`), they are handled separately.

Here is an example which illustrates this point. A string serves as a replacing display specification, which replaces the text that has the property with the specified string (see Section 38.15.4 [Other Display Specs], page 353). Consider the following function:

```
(defun foo ()
  (dotimes (i 5)
    (let ((string (concat "A"))
          (start (+ i i (point-min))))
      (put-text-property start (1+ start) 'display string)
      (put-text-property start (+ 2 start) 'display string))))
```

This function gives each of the first ten characters in the buffer a `display` property which is a string `"A"`, but they don't all get the same string object. The first two characters get the same string object, so they are replaced with one '`A`'; the fact that the display property was assigned in two separate calls to `put-text-property` is irrelevant. Similarly, the next two characters get a second string (`concat` creates a new string object), so they are replaced with one '`A`'; and so on. Thus, the ten characters appear as five A's.

38.15.2 Specified Spaces

To display a space of specified width and/or height, use a display specification of the form `(space . `*`props`*`)`, where *props* is a property list (a list of alternating properties and values). You can put this property on one or more consecutive characters; a space of the specified height and width is displayed in place of *all* of those characters. These are the properties you can use in *props* to specify the weight of the space:

`:width` *width*
: If *width* is an integer or floating point number, it specifies that the space width should be *width* times the normal character width. *width* can also be a *pixel width* specification (see Section 38.15.3 [Pixel Specification], page 352).

`:relative-width` *factor*
: Specifies that the width of the stretch should be computed from the first character in the group of consecutive characters that have the same `display` property. The space width is the width of that character, multiplied by *factor*.

`:align-to` *hpos*
: Specifies that the space should be wide enough to reach *hpos*. If *hpos* is a number, it is measured in units of the normal character width. *hpos* can also be a *pixel width* specification (see Section 38.15.3 [Pixel Specification], page 352).

You should use one and only one of the above properties. You can also specify the height of the space, with these properties:

`:height` *height*
: Specifies the height of the space. If *height* is an integer or floating point number, it specifies that the space height should be *height* times the normal character height. The *height* may also be a *pixel height* specification (see Section 38.15.3 [Pixel Specification], page 352).

`:relative-height` *factor*
: Specifies the height of the space, multiplying the ordinary height of the text having this display specification by *factor*.

`:ascent` *ascent*
: If the value of *ascent* is a non-negative number no greater than 100, it specifies that *ascent* percent of the height of the space should be considered as the ascent of the space—that is, the part above the baseline. The ascent may also be specified in pixel units with a *pixel ascent* specification (see Section 38.15.3 [Pixel Specification], page 352).

Don't use both `:height` and `:relative-height` together.

The `:width` and `:align-to` properties are supported on non-graphic terminals, but the other space properties in this section are not.

Note that space properties are treated as paragraph separators for the purposes of reordering bidirectional text for display. See Section 38.23 [Bidirectional Display], page 382, for the details.

38.15.3 Pixel Specification for Spaces

The value of the `:width`, `:align-to`, `:height`, and `:ascent` properties can be a special kind of expression that is evaluated during redisplay. The result of the evaluation is used as an absolute number of pixels.

The following expressions are supported:

```
expr ::= num | (num) | unit | elem | pos | image | form
num  ::= integer | float | symbol
unit ::= in | mm | cm | width | height
elem ::= left-fringe | right-fringe | left-margin | right-margin
       | scroll-bar | text
pos  ::= left | center | right
form ::= (num . expr) | (op expr ...)
op   ::= + | -
```

The form *num* specifies a fraction of the default frame font height or width. The form `(`*num*`)` specifies an absolute number of pixels. If *num* is a symbol, *symbol*, its buffer-local variable binding is used.

The `in`, `mm`, and `cm` units specify the number of pixels per inch, millimeter, and centimeter, respectively. The `width` and `height` units correspond to the default width and height of the current face. An image specification `image` corresponds to the width or height of the image.

The elements `left-fringe`, `right-fringe`, `left-margin`, `right-margin`, `scroll-bar`, and `text` specify to the width of the corresponding area of the window.

The `left`, `center`, and `right` positions can be used with `:align-to` to specify a position relative to the left edge, center, or right edge of the text area.

Any of the above window elements (except `text`) can also be used with `:align-to` to specify that the position is relative to the left edge of the given area. Once the base offset for a relative position has been set (by the first occurrence of one of these symbols), further occurrences of these symbols are interpreted as the width of the specified area. For example, to align to the center of the left-margin, use

```
:align-to (+ left-margin (0.5 . left-margin))
```

If no specific base offset is set for alignment, it is always relative to the left edge of the text area. For example, '`:align-to 0`' in a header-line aligns with the first text column in the text area.

A value of the form `(num . expr)` stands for the product of the values of *num* and *expr*. For example, `(2 . in)` specifies a width of 2 inches, while `(0.5 . image)` specifies half the width (or height) of the specified image.

The form `(+ expr ...)` adds up the value of the expressions. The form `(- expr ...)` negates or subtracts the value of the expressions.

38.15.4 Other Display Specifications

Here are the other sorts of display specifications that you can use in the `display` text property.

string Display *string* instead of the text that has this property.

Recursive display specifications are not supported—*string*'s `display` properties, if any, are not used.

`(image . image-props)`
This kind of display specification is an image descriptor (see Section 38.16 [Images], page 355). When used as a display specification, it means to display the image instead of the text that has the display specification.

`(slice x y width height)`
This specification together with `image` specifies a *slice* (a partial area) of the image to display. The elements *y* and *x* specify the top left corner of the slice, within the image; *width* and *height* specify the width and height of the slice. Integer values are numbers of pixels. A floating point number in the range 0.0–1.0 stands for that fraction of the width or height of the entire image.

`((margin nil) string)`
A display specification of this form means to display *string* instead of the text that has the display specification, at the same position as that text. It is equivalent to using just *string*, but it is done as a special case of marginal display (see Section 38.15.5 [Display Margins], page 354).

`(left-fringe bitmap [face])`
`(right-fringe bitmap [face])`
This display specification on any character of a line of text causes the specified *bitmap* be displayed in the left or right fringes for that line, instead of the characters that have the display specification. The optional *face* specifies the colors to be used for the bitmap. See Section 38.13.4 [Fringe Bitmaps], page 346, for the details.

`(space-width factor)`
This display specification affects all the space characters within the text that has the specification. It displays all of these spaces *factor* times as wide as normal. The element *factor* should be an integer or float. Characters other than spaces are not affected at all; in particular, this has no effect on tab characters.

`(height height)`
: This display specification makes the text taller or shorter. Here are the possibilities for *height*:

  `(+ n)`
  : This means to use a font that is *n* steps larger. A "step" is defined by the set of available fonts—specifically, those that match what was otherwise specified for this text, in all attributes except height. Each size for which a suitable font is available counts as another step. *n* should be an integer.

  `(- n)`
  : This means to use a font that is *n* steps smaller.

  a number, *factor*
  : A number, *factor*, means to use a font that is *factor* times as tall as the default font.

  a symbol, *function*
  : A symbol is a function to compute the height. It is called with the current height as argument, and should return the new height to use.

  anything else, *form*
  : If the *height* value doesn't fit the previous possibilities, it is a form. Emacs evaluates it to get the new height, with the symbol `height` bound to the current specified font height.

`(raise factor)`
: This kind of display specification raises or lowers the text it applies to, relative to the baseline of the line.

  factor must be a number, which is interpreted as a multiple of the height of the affected text. If it is positive, that means to display the characters raised. If it is negative, that means to display them lower down.

  If the text also has a `height` display specification, that does not affect the amount of raising or lowering, which is based on the faces used for the text.

You can make any display specification conditional. To do that, package it in another list of the form `(when condition . spec)`. Then the specification *spec* applies only when *condition* evaluates to a non-`nil` value. During the evaluation, `object` is bound to the string or buffer having the conditional `display` property. `position` and `buffer-position` are bound to the position within `object` and the buffer position where the `display` property was found, respectively. Both positions can be different when `object` is a string.

38.15.5 Displaying in the Margins

A buffer can have blank areas called *display margins* on the left and on the right. Ordinary text never appears in these areas, but you can put things into the display margins using the `display` property. There is currently no way to make text or images in the margin mouse-sensitive.

The way to display something in the margins is to specify it in a margin display specification in the `display` property of some text. This is a replacing display specification,

meaning that the text you put it on does not get displayed; the margin display appears, but that text does not.

A margin display specification looks like `((margin right-margin) spec)` or `((margin left-margin) spec)`. Here, *spec* is another display specification that says what to display in the margin. Typically it is a string of text to display, or an image descriptor.

To display something in the margin *in association with* certain buffer text, without altering or preventing the display of that text, put a `before-string` property on the text and put the margin display specification on the contents of the before-string.

Before the display margins can display anything, you must give them a nonzero width. The usual way to do that is to set these variables:

`left-margin-width` [Variable]
: This variable specifies the width of the left margin. It is buffer-local in all buffers.

`right-margin-width` [Variable]
: This variable specifies the width of the right margin. It is buffer-local in all buffers.

Setting these variables does not immediately affect the window. These variables are checked when a new buffer is displayed in the window. Thus, you can make changes take effect by calling `set-window-buffer`.

You can also set the margin widths immediately.

`set-window-margins` *window left* **&optional** *right* [Function]
: This function specifies the margin widths for window *window*. The argument *left* controls the left margin and *right* controls the right margin (default `0`).

`window-margins` **&optional** *window* [Function]
: This function returns the left and right margins of *window* as a cons cell of the form `(left . right)`. If *window* is `nil`, the selected window is used.

38.16 Images

To display an image in an Emacs buffer, you must first create an image descriptor, then use it as a display specifier in the `display` property of text that is displayed (see Section 38.15 [Display Property], page 350).

Emacs is usually able to display images when it is run on a graphical terminal. Images cannot be displayed in a text terminal, on certain graphical terminals that lack the support for this, or if Emacs is compiled without image support. You can use the function `display-images-p` to determine if images can in principle be displayed (see Section 29.23 [Display Feature Testing], page 95).

38.16.1 Image Formats

Emacs can display a number of different image formats. Some of these image formats are supported only if particular support libraries are installed. On some platforms, Emacs can load support libraries on demand; if so, the variable `dynamic-library-alist` can be used to modify the set of known names for these dynamic libraries. See Section 39.19 [Dynamic Libraries], page 417.

Supported image formats (and the required support libraries) include PBM and XBM (which do not depend on support libraries and are always available), XPM (`libXpm`), GIF (`libgif` or `libungif`), PostScript (`gs`), JPEG (`libjpeg`), TIFF (`libtiff`), PNG (`libpng`), and SVG (`librsvg`).

Each of these image formats is associated with an *image type symbol*. The symbols for the above formats are, respectively, `pbm`, `xbm`, `xpm`, `gif`, `postscript`, `jpeg`, `tiff`, `png`, and `svg`.

Furthermore, if you build Emacs with ImageMagick (`libMagickWand`) support, Emacs can display any image format that ImageMagick can. See Section 38.16.8 [ImageMagick Images], page 361. All images displayed via ImageMagick have type symbol `imagemagick`.

`image-types` [Variable]

This variable contains a list of type symbols for image formats which are potentially supported in the current configuration.

"Potentially" means that Emacs knows about the image types, not necessarily that they can be used (for example, they could depend on unavailable dynamic libraries). To know which image types are really available, use `image-type-available-p`.

`image-type-available-p` *type* [Function]

This function returns non-`nil` if images of type *type* can be loaded and displayed. *type* must be an image type symbol.

For image types whose support libraries are statically linked, this function always returns `t`. For image types whose support libraries are dynamically loaded, it returns `t` if the library could be loaded and `nil` otherwise.

38.16.2 Image Descriptors

An *image descriptor* is a list which specifies the underlying data for an image, and how to display it. It is typically used as the value of a `display` overlay or text property (see Section 38.15.4 [Other Display Specs], page 353); but See Section 38.16.11 [Showing Images], page 364, for convenient helper functions to insert images into buffers.

Each image descriptor has the form `(image . props)`, where *props* is a property list of alternating keyword symbols and values, including at least the pair `:type TYPE` which specifies the image type.

The following is a list of properties that are meaningful for all image types (there are also properties which are meaningful only for certain image types, as documented in the following subsections):

`:type` *type*

The image type. Every image descriptor must include this property.

`:file` *file*

This says to load the image from file *file*. If *file* is not an absolute file name, it is expanded in `data-directory`.

`:data` *data*

This specifies the raw image data. Each image descriptor must have either `:data` or `:file`, but not both.

For most image types, the value of a `:data` property should be a string containing the image data. Some image types do not support `:data`; for some others, `:data` alone is not enough, so you need to use other image properties along with `:data`. See the following subsections for details.

`:margin` *margin*
: This specifies how many pixels to add as an extra margin around the image. The value, *margin*, must be a non-negative number, or a pair `(x . y)` of such numbers. If it is a pair, *x* specifies how many pixels to add horizontally, and *y* specifies how many pixels to add vertically. If `:margin` is not specified, the default is zero.

`:ascent` *ascent*
: This specifies the amount of the image's height to use for its ascent—that is, the part above the baseline. The value, *ascent*, must be a number in the range 0 to 100, or the symbol `center`.

  If *ascent* is a number, that percentage of the image's height is used for its ascent.

  If *ascent* is `center`, the image is vertically centered around a centerline which would be the vertical centerline of text drawn at the position of the image, in the manner specified by the text properties and overlays that apply to the image.

  If this property is omitted, it defaults to 50.

`:relief` *relief*
: This adds a shadow rectangle around the image. The value, *relief*, specifies the width of the shadow lines, in pixels. If *relief* is negative, shadows are drawn so that the image appears as a pressed button; otherwise, it appears as an unpressed button.

`:conversion` *algorithm*
: This specifies a conversion algorithm that should be applied to the image before it is displayed; the value, *algorithm*, specifies which algorithm.

  `laplace`
  `emboss`
  : Specifies the Laplace edge detection algorithm, which blurs out small differences in color while highlighting larger differences. People sometimes consider this useful for displaying the image for a "disabled" button.

  `(edge-detection :matrix` *matrix* `:color-adjust` *adjust*`)`
  : Specifies a general edge-detection algorithm. *matrix* must be either a nine-element list or a nine-element vector of numbers. A pixel at position x/y in the transformed image is computed from original pixels around that position. *matrix* specifies, for each pixel in the neighborhood of x/y, a factor with which that pixel will influence the transformed pixel; element 0 specifies the factor for the pixel at $x-1/y-1$, element 1 the factor for the pixel at $x/y-1$ etc., as

shown below:

$$\begin{pmatrix} x-1/y-1 & x/y-1 & x+1/y-1 \\ x-1/y & x/y & x+1/y \\ x-1/y+1 & x/y+1 & x+1/y+1 \end{pmatrix}$$

The resulting pixel is computed from the color intensity of the color resulting from summing up the RGB values of surrounding pixels, multiplied by the specified factors, and dividing that sum by the sum of the factors' absolute values.

Laplace edge-detection currently uses a matrix of

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

Emboss edge-detection uses a matrix of

$$\begin{pmatrix} 2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & -2 \end{pmatrix}$$

`disabled` Specifies transforming the image so that it looks "disabled".

`:mask` *mask*

If *mask* is `heuristic` or (`heuristic` *bg*), build a clipping mask for the image, so that the background of a frame is visible behind the image. If *bg* is not specified, or if *bg* is `t`, determine the background color of the image by looking at the four corners of the image, assuming the most frequently occurring color from the corners is the background color of the image. Otherwise, *bg* must be a list (*red green blue*) specifying the color to assume for the background of the image.

If *mask* is `nil`, remove a mask from the image, if it has one. Images in some formats include a mask which can be removed by specifying `:mask nil`.

`:pointer` *shape*

This specifies the pointer shape when the mouse pointer is over this image. See Section 29.17 [Pointer Shape], page 90, for available pointer shapes.

`:map` *map* This associates an image map of *hot spots* with this image.

An image map is an alist where each element has the format (*area id plist*). An *area* is specified as either a rectangle, a circle, or a polygon.

A rectangle is a cons (`rect . ((`*x0* `.` *y0*`) . (`*x1* `.` *y1*`))`) which specifies the pixel coordinates of the upper left and bottom right corners of the rectangle area.

A circle is a cons (`circle . ((`*x0* `.` *y0*`) .` *r*`))` which specifies the center and the radius of the circle; *r* may be a float or integer.

A polygon is a cons (`poly . [`*x0 y0 x1 y1 ...*`]`) where each pair in the vector describes one corner in the polygon.

When the mouse pointer lies on a hot-spot area of an image, the *plist* of that hot-spot is consulted; if it contains a `help-echo` property, that defines a tool-tip for the hot-spot, and if it contains a `pointer` property, that defines the shape of the mouse cursor when it is on the hot-spot. See Section 29.17 [Pointer Shape], page 90, for available pointer shapes.

When you click the mouse when the mouse pointer is over a hot-spot, an event is composed by combining the *id* of the hot-spot with the mouse event; for instance, `[area4 mouse-1]` if the hot-spot's *id* is `area4`.

`image-mask-p` *spec* **&optional** *frame* [Function]

This function returns `t` if image *spec* has a mask bitmap. *frame* is the frame on which the image will be displayed. *frame* `nil` or omitted means to use the selected frame (see Section 29.9 [Input Focus], page 83).

38.16.3 XBM Images

To use XBM format, specify `xbm` as the image type. This image format doesn't require an external library, so images of this type are always supported.

Additional image properties supported for the `xbm` image type are:

`:foreground` *foreground*

The value, *foreground*, should be a string specifying the image foreground color, or `nil` for the default color. This color is used for each pixel in the XBM that is 1. The default is the frame's foreground color.

`:background` *background*

The value, *background*, should be a string specifying the image background color, or `nil` for the default color. This color is used for each pixel in the XBM that is 0. The default is the frame's background color.

If you specify an XBM image using data within Emacs instead of an external file, use the following three properties:

`:data` *data*

The value, *data*, specifies the contents of the image. There are three formats you can use for *data*:

- A vector of strings or bool-vectors, each specifying one line of the image. Do specify `:height` and `:width`.
- A string containing the same byte sequence as an XBM file would contain. You must not specify `:height` and `:width` in this case, because omitting them is what indicates the data has the format of an XBM file. The file contents specify the height and width of the image.
- A string or a bool-vector containing the bits of the image (plus perhaps some extra bits at the end that will not be used). It should contain at least *width* * `height` bits. In this case, you must specify `:height` and `:width`, both to indicate that the string contains just the bits rather than a whole XBM file, and to specify the size of the image.

`:width` *width*

The value, *width*, specifies the width of the image, in pixels.

`:height` *height*
: The value, *height*, specifies the height of the image, in pixels.

38.16.4 XPM Images

To use XPM format, specify `xpm` as the image type. The additional image property `:color-symbols` is also meaningful with the `xpm` image type:

`:color-symbols` *symbols*
: The value, *symbols*, should be an alist whose elements have the form `(`*name* `.` *color*`)`. In each element, *name* is the name of a color as it appears in the image file, and *color* specifies the actual color to use for displaying that name.

38.16.5 GIF Images

For GIF images, specify image type `gif`.

`:index` *index*
: You can use `:index` to specify image number *index* from a GIF file that contains more than one image. If the GIF file doesn't contain an image with the specified index, the image displays as a hollow box. GIF files with more than one image can be animated, see Section 38.16.12 [Animated Images], page 365.

38.16.6 TIFF Images

For TIFF images, specify image type `tiff`.

`:index` *index*
: You can use `:index` to specify image number *index* from a TIFF file that contains more than one image. If the TIFF file doesn't contain an image with the specified index, the image displays as a hollow box.

38.16.7 PostScript Images

To use PostScript for an image, specify image type `postscript`. This works only if you have Ghostscript installed. You must always use these three properties:

`:pt-width` *width*
: The value, *width*, specifies the width of the image measured in points (1/72 inch). *width* must be an integer.

`:pt-height` *height*
: The value, *height*, specifies the height of the image in points (1/72 inch). *height* must be an integer.

`:bounding-box` *box*
: The value, *box*, must be a list or vector of four integers, which specifying the bounding box of the PostScript image, analogous to the '`BoundingBox`' comment found in PostScript files.

```
%%BoundingBox: 22 171 567 738
```

38.16.8 ImageMagick Images

If you build Emacs with ImageMagick support, you can use the ImageMagick library to load many image formats. The image type symbol for images loaded via ImageMagick is `imagemagick`, regardless of the actual underlying image format.

`imagemagick-types` [Function]
: This function returns a list of image file extensions supported by the current ImageMagick installation.

By default, Emacs does not use ImageMagick to display images in Image mode, e.g. when visiting such files with `C-x C-f`. This feature is enabled by calling `imagemagick-register-types`.

`imagemagick-register-types` [Function]
: This function enables using Image mode to visit image files supported by ImageMagick. See Section "File Conveniences" in *The GNU Emacs Manual*. It also causes `create-image` and other helper functions to associate such file names with the `imagemagick` image type (see Section 38.16.10 [Defining Images], page 362).

  All image file extensions supported by ImageMagick are registered, except those specified in `imagemagick-types-inhibit`. If Emacs was not compiled with ImageMagick support, this function does nothing.

`imagemagick-types-inhibit` [User Option]
: This variable specifies a list of image types that should *not* be registered by `imagemagick-register-types`. Each entry in this list should be one of the symbols returned by `imagemagick-types`. The default value lists several file types that are considered "images" by ImageMagick, but which should not be considered as images by Emacs, including C files and HTML files.

Images loaded with ImageMagick support the following additional image descriptor properties:

`:width, :height`
: The `:width` and `:height` keywords are used for scaling the image. If only one of them is specified, the other one will be calculated so as to preserve the aspect ratio. If both are specified, aspect ratio may not be preserved.

`:rotation`
: Specifies a rotation angle in degrees.

`:index` This has the same meaning as it does for GIF images (see Section 38.16.5 [GIF Images], page 360), i.e. it specifies which image to view inside an image bundle file format such as DJVM. You can use the `image-metadata` function to retrieve the total number of images in an image bundle.

38.16.9 Other Image Types

For PBM images, specify image type `pbm`. Color, gray-scale and monochromatic images are supported. For mono PBM images, two additional image properties are supported.

`:foreground` *foreground*
: The value, *foreground*, should be a string specifying the image foreground color, or `nil` for the default color. This color is used for each pixel in the PBM that is 1. The default is the frame's foreground color.

`:background` *background*
: The value, *background*, should be a string specifying the image background color, or `nil` for the default color. This color is used for each pixel in the PBM that is 0. The default is the frame's background color.

For JPEG images, specify image type `jpeg`.

For TIFF images, specify image type `tiff`.

For PNG images, specify image type `png`.

For SVG images, specify image type `svg`.

38.16.10 Defining Images

The functions `create-image`, `defimage` and `find-image` provide convenient ways to create image descriptors.

`create-image` *file-or-data* **&optional** *type data-p* **&rest** *props* [Function]
: This function creates and returns an image descriptor which uses the data in *file-or-data*. *file-or-data* can be a file name or a string containing the image data; *data-p* should be `nil` for the former case, non-`nil` for the latter case.

  The optional argument *type* is a symbol specifying the image type. If *type* is omitted or `nil`, `create-image` tries to determine the image type from the file's first few bytes, or else from the file's name.

  The remaining arguments, *props*, specify additional image properties—for example,

  ```
  (create-image "foo.xpm" 'xpm nil :heuristic-mask t)
  ```

  The function returns `nil` if images of this type are not supported. Otherwise it returns an image descriptor.

`defimage` *symbol specs* **&optional** *doc* [Macro]
: This macro defines *symbol* as an image name. The arguments *specs* is a list which specifies how to display the image. The third argument, *doc*, is an optional documentation string.

  Each argument in *specs* has the form of a property list, and each one should specify at least the `:type` property and either the `:file` or the `:data` property. The value of `:type` should be a symbol specifying the image type, the value of `:file` is the file to load the image from, and the value of `:data` is a string containing the actual image data. Here is an example:

  ```
  (defimage test-image
    ((:type xpm :file "~/test1.xpm")
     (:type xbm :file "~/test1.xbm")))
  ```

  `defimage` tests each argument, one by one, to see if it is usable—that is, if the type is supported and the file exists. The first usable argument is used to make an image descriptor which is stored in *symbol*.

  If none of the alternatives will work, then *symbol* is defined as `nil`.

`find-image` *specs* [Function]

This function provides a convenient way to find an image satisfying one of a list of image specifications *specs*.

Each specification in *specs* is a property list with contents depending on image type. All specifications must at least contain the properties `:type` *type* and either `:file` *file* or `:data` *DATA*, where *type* is a symbol specifying the image type, e.g. `xbm`, *file* is the file to load the image from, and *data* is a string containing the actual image data. The first specification in the list whose *type* is supported, and *file* exists, is used to construct the image specification to be returned. If no specification is satisfied, `nil` is returned.

The image is looked for in `image-load-path`.

`image-load-path` [Variable]

This variable's value is a list of locations in which to search for image files. If an element is a string or a variable symbol whose value is a string, the string is taken to be the name of a directory to search. If an element is a variable symbol whose value is a list, that is taken to be a list of directory names to search.

The default is to search in the '`images`' subdirectory of the directory specified by `data-directory`, then the directory specified by `data-directory`, and finally in the directories in `load-path`. Subdirectories are not automatically included in the search, so if you put an image file in a subdirectory, you have to supply the subdirectory name explicitly. For example, to find the image '`images/foo/bar.xpm`' within `data-directory`, you should specify the image as follows:

```
(defimage foo-image '((:type xpm :file "foo/bar.xpm")))
```

`image-load-path-for-library` *library image* **&optional** *path no-error* [Function]

This function returns a suitable search path for images used by the Lisp package *library*.

The function searches for *image* first using `image-load-path`, excluding '`data-directory/images`', and then in `load-path`, followed by a path suitable for *library*, which includes '`../../etc/images`' and '`../etc/images`' relative to the library file itself, and finally in '`data-directory/images`'.

Then this function returns a list of directories which contains first the directory in which *image* was found, followed by the value of `load-path`. If *path* is given, it is used instead of `load-path`.

If *no-error* is non-`nil` and a suitable path can't be found, don't signal an error. Instead, return a list of directories as before, except that `nil` appears in place of the image directory.

Here is an example of using `image-load-path-for-library`:

```
(defvar image-load-path) ; shush compiler
(let* ((load-path (image-load-path-for-library
                   "mh-e" "mh-logo.xpm"))
       (image-load-path (cons (car load-path)
                              image-load-path)))
  (mh-tool-bar-folder-buttons-init))
```

38.16.11 Showing Images

You can use an image descriptor by setting up the `display` property yourself, but it is easier to use the functions in this section.

`insert-image` *image* **&optional** *string area slice* [Function]

This function inserts *image* in the current buffer at point. The value *image* should be an image descriptor; it could be a value returned by `create-image`, or the value of a symbol defined with `defimage`. The argument *string* specifies the text to put in the buffer to hold the image. If it is omitted or `nil`, `insert-image` uses `" "` by default.

The argument *area* specifies whether to put the image in a margin. If it is `left-margin`, the image appears in the left margin; `right-margin` specifies the right margin. If *area* is `nil` or omitted, the image is displayed at point within the buffer's text.

The argument *slice* specifies a slice of the image to insert. If *slice* is `nil` or omitted the whole image is inserted. Otherwise, *slice* is a list `(x y width height)` which specifies the *x* and *y* positions and *width* and *height* of the image area to insert. Integer values are in units of pixels. A floating point number in the range 0.0–1.0 stands for that fraction of the width or height of the entire image.

Internally, this function inserts *string* in the buffer, and gives it a `display` property which specifies *image*. See Section 38.15 [Display Property], page 350.

`insert-sliced-image` *image* **&optional** *string area rows cols* [Function]

This function inserts *image* in the current buffer at point, like `insert-image`, but splits the image into *rows*x*cols* equally sized slices.

If an image is inserted "sliced", Emacs displays each slice as a separate image, and allow more intuitive scrolling up/down, instead of jumping up/down the entire image when paging through a buffer that displays (large) images.

`put-image` *image pos* **&optional** *string area* [Function]

This function puts image *image* in front of *pos* in the current buffer. The argument *pos* should be an integer or a marker. It specifies the buffer position where the image should appear. The argument *string* specifies the text that should hold the image as an alternative to the default.

The argument *image* must be an image descriptor, perhaps returned by `create-image` or stored by `defimage`.

The argument *area* specifies whether to put the image in a margin. If it is `left-margin`, the image appears in the left margin; `right-margin` specifies the right margin. If *area* is `nil` or omitted, the image is displayed at point within the buffer's text.

Internally, this function creates an overlay, and gives it a `before-string` property containing text that has a `display` property whose value is the image. (Whew!)

`remove-images` *start end* **&optional** *buffer* [Function]

This function removes images in *buffer* between positions *start* and *end*. If *buffer* is omitted or `nil`, images are removed from the current buffer.

This removes only images that were put into *buffer* the way `put-image` does it, not images that were inserted with `insert-image` or in other ways.

`image-size` *spec* **&optional** *pixels frame* [Function]
: This function returns the size of an image as a pair `(width . height)`. *spec* is an image specification. *pixels* non-`nil` means return sizes measured in pixels, otherwise return sizes measured in canonical character units (fractions of the width/height of the frame's default font). *frame* is the frame on which the image will be displayed. *frame* null or omitted means use the selected frame (see Section 29.9 [Input Focus], page 83).

`max-image-size` [Variable]
: This variable is used to define the maximum size of image that Emacs will load. Emacs will refuse to load (and display) any image that is larger than this limit.

  If the value is an integer, it directly specifies the maximum image height and width, measured in pixels. If it is a floating point number, it specifies the maximum image height and width as a ratio to the frame height and width. If the value is non-numeric, there is no explicit limit on the size of images.

  The purpose of this variable is to prevent unreasonably large images from accidentally being loaded into Emacs. It only takes effect the first time an image is loaded. Once an image is placed in the image cache, it can always be displayed, even if the value of *max-image-size* is subsequently changed (see Section 38.16.13 [Image Cache], page 365).

38.16.12 Animated Images

Some image files can contain more than one image. This can be used to create animation. Currently, Emacs only supports animated GIF files. The following functions related to animated images are available.

`image-animated-p` *image* [Function]
: This function returns non-`nil` if *image* can be animated. The actual return value is a cons `(nimages . delay)`, where *nimages* is the number of frames and *delay* is the delay in seconds between them.

`image-animate` *image* **&optional** *index limit* [Function]
: This function animates *image*. The optional integer *index* specifies the frame from which to start (default 0). The optional argument *limit* controls the length of the animation. If omitted or `nil`, the image animates once only; if `t` it loops forever; if a number animation stops after that many seconds.

Animation operates by means of a timer. Note that Emacs imposes a minimum frame delay of 0.01 seconds.

`image-animate-timer` *image* [Function]
: This function returns the timer responsible for animating *image*, if there is one.

38.16.13 Image Cache

Emacs caches images so that it can display them again more efficiently. When Emacs displays an image, it searches the image cache for an existing image specification `equal` to the desired specification. If a match is found, the image is displayed from the cache. Otherwise, Emacs loads the image normally.

`image-flush` *spec* **&optional** *frame* [Function]
: This function removes the image with specification *spec* from the image cache of frame *frame*. Image specifications are compared using `equal`. If *frame* is `nil`, it defaults to the selected frame. If *frame* is `t`, the image is flushed on all existing frames.

  In Emacs's current implementation, each graphical terminal possesses an image cache, which is shared by all the frames on that terminal (see Section 29.2 [Multiple Terminals], page 67). Thus, refreshing an image in one frame also refreshes it in all other frames on the same terminal.

One use for `image-flush` is to tell Emacs about a change in an image file. If an image specification contains a `:file` property, the image is cached based on the file's contents when the image is first displayed. Even if the file subsequently changes, Emacs continues displaying the old version of the image. Calling `image-flush` flushes the image from the cache, forcing Emacs to re-read the file the next time it needs to display that image.

Another use for `image-flush` is for memory conservation. If your Lisp program creates a large number of temporary images over a period much shorter than `image-cache-eviction-delay` (see below), you can opt to flush unused images yourself, instead of waiting for Emacs to do it automatically.

`clear-image-cache` **&optional** *filter* [Function]
: This function clears an image cache, removing all the images stored in it. If *filter* is omitted or `nil`, it clears the cache for the selected frame. If *filter* is a frame, it clears the cache for that frame. If *filter* is `t`, all image caches are cleared. Otherwise, *filter* is taken to be a file name, and all images associated with that file name are removed from all image caches.

If an image in the image cache has not been displayed for a specified period of time, Emacs removes it from the cache and frees the associated memory.

`image-cache-eviction-delay` [Variable]
: This variable specifies the number of seconds an image can remain in the cache without being displayed. When an image is not displayed for this length of time, Emacs removes it from the image cache.

  Under some circumstances, if the number of images in the cache grows too large, the actual eviction delay may be shorter than this.

  If the value is `nil`, Emacs does not remove images from the cache except when you explicitly clear it. This mode can be useful for debugging.

38.17 Buttons

The Button package defines functions for inserting and manipulating *buttons* that can be activated with the mouse or via keyboard commands. These buttons are typically used for various kinds of hyperlinks.

A button is essentially a set of text or overlay properties, attached to a stretch of text in a buffer. These properties are called *button properties*. One of these properties, the *action property*, specifies a function which is called when the user invokes the button using the keyboard or the mouse. The action function may examine the button and use its other properties as desired.

In some ways, the Button package duplicates the functionality in the Widget package. See Section "Introduction" in *The Emacs Widget Library*. The advantage of the Button package is that it is faster, smaller, and simpler to program. From the point of view of the user, the interfaces produced by the two packages are very similar.

38.17.1 Button Properties

Each button has an associated list of properties defining its appearance and behavior, and other arbitrary properties may be used for application specific purposes. The following properties have special meaning to the Button package:

`action` The function to call when the user invokes the button, which is passed the single argument *button*. By default this is `ignore`, which does nothing.

`mouse-action`
This is similar to `action`, and when present, will be used instead of `action` for button invocations resulting from mouse-clicks (instead of the user hitting `RET`). If not present, mouse-clicks use `action` instead.

`face` This is an Emacs face controlling how buttons of this type are displayed; by default this is the `button` face.

`mouse-face`
This is an additional face which controls appearance during mouse-overs (merged with the usual button face); by default this is the usual Emacs `highlight` face.

`keymap` The button's keymap, defining bindings active within the button region. By default this is the usual button region keymap, stored in the variable `button-map`, which defines `RET` and `mouse-2` to invoke the button.

`type` The button type. See Section 38.17.2 [Button Types], page 367.

`help-echo`
A string displayed by the Emacs tool-tip help system; by default, `"mouse-2, RET: Push this button"`.

`follow-link`
The follow-link property, defining how a `Mouse-1` click behaves on this button, See Section 32.19.8 [Clickable Text], page 169.

`button` All buttons have a non-`nil` `button` property, which may be useful in finding regions of text that comprise buttons (which is what the standard button functions do).

There are other properties defined for the regions of text in a button, but these are not generally interesting for typical uses.

38.17.2 Button Types

Every button has a *button type*, which defines default values for the button's properties. Button types are arranged in a hierarchy, with specialized types inheriting from more general types, so that it's easy to define special-purpose types of buttons for specific tasks.

`define-button-type` *name* **&rest** *properties* [Function]

Define a 'button type' called *name* (a symbol). The remaining arguments form a sequence of *property value* pairs, specifying default property values for buttons with this type (a button's type may be set by giving it a `type` property when creating the button, using the `:type` keyword argument).

In addition, the keyword argument `:supertype` may be used to specify a button-type from which *name* inherits its default property values. Note that this inheritance happens only when *name* is defined; subsequent changes to a supertype are not reflected in its subtypes.

Using `define-button-type` to define default properties for buttons is not necessary—buttons without any specified type use the built-in button-type `button`—but it is encouraged, since doing so usually makes the resulting code clearer and more efficient.

38.17.3 Making Buttons

Buttons are associated with a region of text, using an overlay or text properties to hold button-specific information, all of which are initialized from the button's type (which defaults to the built-in button type `button`). Like all Emacs text, the appearance of the button is governed by the `face` property; by default (via the `face` property inherited from the `button` button-type) this is a simple underline, like a typical web-page link.

For convenience, there are two sorts of button-creation functions, those that add button properties to an existing region of a buffer, called `make-...button`, and those that also insert the button text, called `insert-...button`.

The button-creation functions all take the `&rest` argument *properties*, which should be a sequence of *property value* pairs, specifying properties to add to the button; see Section 38.17.1 [Button Properties], page 367. In addition, the keyword argument `:type` may be used to specify a button-type from which to inherit other properties; see Section 38.17.2 [Button Types], page 367. Any properties not explicitly specified during creation will be inherited from the button's type (if the type defines such a property).

The following functions add a button using an overlay (see Section 38.9 [Overlays], page 315) to hold the button properties:

`make-button` *beg end* **&rest** *properties* [Function]

This makes a button from *beg* to *end* in the current buffer, and returns it.

`insert-button` *label* **&rest** *properties* [Function]

This insert a button with the label *label* at point, and returns it.

The following functions are similar, but using text properties (see Section 32.19 [Text Properties], page 156) to hold the button properties. Such buttons do not add markers to the buffer, so editing in the buffer does not slow down if there is an extremely large numbers of buttons. However, if there is an existing face text property on the text (e.g. a face assigned by Font Lock mode), the button face may not be visible. Both of these functions return the starting position of the new button.

`make-text-button` *beg end* **&rest** *properties* [Function]

This makes a button from *beg* to *end* in the current buffer, using text properties.

`insert-text-button` *label* **&rest** *properties* [Function]
This inserts a button with the label *label* at point, using text properties.

38.17.4 Manipulating Buttons

These are functions for getting and setting properties of buttons. Often these are used by a button's invocation function to determine what to do.

Where a *button* parameter is specified, it means an object referring to a specific button, either an overlay (for overlay buttons), or a buffer-position or marker (for text property buttons). Such an object is passed as the first argument to a button's invocation function when it is invoked.

`button-start` *button* [Function]
Return the position at which *button* starts.

`button-end` *button* [Function]
Return the position at which *button* ends.

`button-get` *button prop* [Function]
Get the property of button *button* named *prop*.

`button-put` *button prop val* [Function]
Set *button*'s *prop* property to *val*.

`button-activate` *button* **&optional** *use-mouse-action* [Function]
Call *button*'s `action` property (i.e., invoke it). If *use-mouse-action* is non-`nil`, try to invoke the button's `mouse-action` property instead of `action`; if the button has no `mouse-action` property, use `action` as normal.

`button-label` *button* [Function]
Return *button*'s text label.

`button-type` *button* [Function]
Return *button*'s button-type.

`button-has-type-p` *button type* [Function]
Return `t` if *button* has button-type *type*, or one of *type*'s subtypes.

`button-at` *pos* [Function]
Return the button at position *pos* in the current buffer, or `nil`. If the button at *pos* is a text property button, the return value is a marker pointing to *pos*.

`button-type-put` *type prop val* [Function]
Set the button-type *type*'s *prop* property to *val*.

`button-type-get` *type prop* [Function]
Get the property of button-type *type* named *prop*.

`button-type-subtype-p` *type supertype* [Function]
Return `t` if button-type *type* is a subtype of *supertype*.

38.17.5 Button Buffer Commands

These are commands and functions for locating and operating on buttons in an Emacs buffer.

`push-button` is the command that a user uses to actually 'push' a button, and is bound by default in the button itself to `RET` and to `mouse-2` using a local keymap in the button's overlay or text properties. Commands that are useful outside the buttons itself, such as `forward-button` and `backward-button` are additionally available in the keymap stored in `button-buffer-map`; a mode which uses buttons may want to use `button-buffer-map` as a parent keymap for its keymap.

If the button has a non-`nil` `follow-link` property, and *mouse-1-click-follows-link* is set, a quick `Mouse-1` click will also activate the `push-button` command. See Section 32.19.8 [Clickable Text], page 169.

`push-button` **&optional** *pos use-mouse-action* [Command]
: Perform the action specified by a button at location *pos*. *pos* may be either a buffer position or a mouse-event. If *use-mouse-action* is non-`nil`, or *pos* is a mouse-event (see Section 21.7.3 [Mouse Events], page 329, vol. 1), try to invoke the button's `mouse-action` property instead of `action`; if the button has no `mouse-action` property, use `action` as normal. *pos* defaults to point, except when `push-button` is invoked interactively as the result of a mouse-event, in which case, the mouse event's position is used. If there's no button at *pos*, do nothing and return `nil`, otherwise return `t`.

`forward-button` *n* **&optional** *wrap display-message* [Command]
: Move to the *n*th next button, or *n*th previous button if *n* is negative. If *n* is zero, move to the start of any button at point. If *wrap* is non-`nil`, moving past either end of the buffer continues from the other end. If *display-message* is non-`nil`, the button's help-echo string is displayed. Any button with a non-`nil` `skip` property is skipped over. Returns the button found.

`backward-button` *n* **&optional** *wrap display-message* [Command]
: Move to the *n*th previous button, or *n*th next button if *n* is negative. If *n* is zero, move to the start of any button at point. If *wrap* is non-`nil`, moving past either end of the buffer continues from the other end. If *display-message* is non-`nil`, the button's help-echo string is displayed. Any button with a non-`nil` `skip` property is skipped over. Returns the button found.

`next-button` *pos* **&optional** *count-current* [Function]
`previous-button` *pos* **&optional** *count-current* [Function]
: Return the next button after (for `next-button` or before (for `previous-button`) position *pos* in the current buffer. If *count-current* is non-`nil`, count any button at *pos* in the search, instead of starting at the next button.

38.18 Abstract Display

The Ewoc package constructs buffer text that represents a structure of Lisp objects, and updates the text to follow changes in that structure. This is like the "view" component in the "model/view/controller" design paradigm.

An *ewoc* is a structure that organizes information required to construct buffer text that represents certain Lisp data. The buffer text of the ewoc has three parts, in order: first, fixed *header* text; next, textual descriptions of a series of data elements (Lisp objects that you specify); and last, fixed *footer* text. Specifically, an ewoc contains information on:

- The buffer which its text is generated in.
- The text's start position in the buffer.
- The header and footer strings.
- A doubly-linked chain of *nodes*, each of which contains:
  - A *data element*, a single Lisp object.
  - Links to the preceding and following nodes in the chain.
- A *pretty-printer* function which is responsible for inserting the textual representation of a data element value into the current buffer.

Typically, you define an ewoc with `ewoc-create`, and then pass the resulting ewoc structure to other functions in the Ewoc package to build nodes within it, and display it in the buffer. Once it is displayed in the buffer, other functions determine the correspondence between buffer positions and nodes, move point from one node's textual representation to another, and so forth. See Section 38.18.1 [Abstract Display Functions], page 371.

A node *encapsulates* a data element much the way a variable holds a value. Normally, encapsulation occurs as a part of adding a node to the ewoc. You can retrieve the data element value and place a new value in its place, like so:

```
(ewoc-data node)
⇒ value

(ewoc-set-data node new-value)
⇒ new-value
```

You can also use, as the data element value, a Lisp object (list or vector) that is a container for the "real" value, or an index into some other structure. The example (see Section 38.18.2 [Abstract Display Example], page 373) uses the latter approach.

When the data changes, you will want to update the text in the buffer. You can update all nodes by calling `ewoc-refresh`, or just specific nodes using `ewoc-invalidate`, or all nodes satisfying a predicate using `ewoc-map`. Alternatively, you can delete invalid nodes using `ewoc-delete` or `ewoc-filter`, and add new nodes in their place. Deleting a node from an ewoc deletes its associated textual description from buffer, as well.

38.18.1 Abstract Display Functions

In this subsection, *ewoc* and *node* stand for the structures described above (see Section 38.18 [Abstract Display], page 370), while *data* stands for an arbitrary Lisp object used as a data element.

`ewoc-create` *pretty-printer* **&optional** *header footer nosep* [Function]

This constructs and returns a new ewoc, with no nodes (and thus no data elements). *pretty-printer* should be a function that takes one argument, a data element of the sort you plan to use in this ewoc, and inserts its textual description at point using `insert` (and never `insert-before-markers`, because that would interfere with the Ewoc package's internal mechanisms).

Normally, a newline is automatically inserted after the header, the footer and every node's textual description. If *nosep* is non-`nil`, no newline is inserted. This may be useful for displaying an entire ewoc on a single line, for example, or for making nodes "invisible" by arranging for *pretty-printer* to do nothing for those nodes.

An ewoc maintains its text in the buffer that is current when you create it, so switch to the intended buffer before calling `ewoc-create`.

`ewoc-buffer` *ewoc* [Function]
This returns the buffer where *ewoc* maintains its text.

`ewoc-get-hf` *ewoc* [Function]
This returns a cons cell (`header . footer`) made from *ewoc*'s header and footer.

`ewoc-set-hf` *ewoc header footer* [Function]
This sets the header and footer of *ewoc* to the strings *header* and *footer*, respectively.

`ewoc-enter-first` *ewoc data* [Function]
`ewoc-enter-last` *ewoc data* [Function]
These add a new node encapsulating *data*, putting it, respectively, at the beginning or end of *ewoc*'s chain of nodes.

`ewoc-enter-before` *ewoc node data* [Function]
`ewoc-enter-after` *ewoc node data* [Function]
These add a new node encapsulating *data*, adding it to *ewoc* before or after *node*, respectively.

`ewoc-prev` *ewoc node* [Function]
`ewoc-next` *ewoc node* [Function]
These return, respectively, the previous node and the next node of *node* in *ewoc*.

`ewoc-nth` *ewoc n* [Function]
This returns the node in *ewoc* found at zero-based index *n*. A negative *n* means count from the end. `ewoc-nth` returns `nil` if *n* is out of range.

`ewoc-data` *node* [Function]
This extracts the data encapsulated by *node* and returns it.

`ewoc-set-data` *node data* [Function]
This sets the data encapsulated by *node* to *data*.

`ewoc-locate` *ewoc* **&optional** *pos guess* [Function]
This determines the node in *ewoc* which contains point (or *pos* if specified), and returns that node. If *ewoc* has no nodes, it returns `nil`. If *pos* is before the first node, it returns the first node; if *pos* is after the last node, it returns the last node. The optional third arg *guess* should be a node that is likely to be near *pos*; this doesn't alter the result, but makes the function run faster.

`ewoc-location` *node* [Function]
This returns the start position of *node*.

`ewoc-goto-prev` *ewoc arg* [Function]
`ewoc-goto-next` *ewoc arg* [Function]
These move point to the previous or next, respectively, *arg*th node in *ewoc*. `ewoc-goto-prev` does not move if it is already at the first node or if *ewoc* is empty, whereas `ewoc-goto-next` moves past the last node, returning `nil`. Excepting this special case, these functions return the node moved to.

`ewoc-goto-node` *ewoc node* [Function]
This moves point to the start of *node* in *ewoc*.

`ewoc-refresh` *ewoc* [Function]
This function regenerates the text of *ewoc*. It works by deleting the text between the header and the footer, i.e., all the data elements' representations, and then calling the pretty-printer function for each node, one by one, in order.

`ewoc-invalidate` *ewoc* **&rest** *nodes* [Function]
This is similar to `ewoc-refresh`, except that only *nodes* in *ewoc* are updated instead of the entire set.

`ewoc-delete` *ewoc* **&rest** *nodes* [Function]
This deletes each node in *nodes* from *ewoc*.

`ewoc-filter` *ewoc predicate* **&rest** *args* [Function]
This calls *predicate* for each data element in *ewoc* and deletes those nodes for which *predicate* returns `nil`. Any *args* are passed to *predicate*.

`ewoc-collect` *ewoc predicate* **&rest** *args* [Function]
This calls *predicate* for each data element in *ewoc* and returns a list of those elements for which *predicate* returns non-`nil`. The elements in the list are ordered as in the buffer. Any *args* are passed to *predicate*.

`ewoc-map` *map-function ewoc* **&rest** *args* [Function]
This calls *map-function* for each data element in *ewoc* and updates those nodes for which *map-function* returns non-`nil`. Any *args* are passed to *map-function*.

38.18.2 Abstract Display Example

Here is a simple example using functions of the ewoc package to implement a "color components display", an area in a buffer that represents a vector of three integers (itself representing a 24-bit RGB value) in various ways.

```
(setq colorcomp-ewoc nil
      colorcomp-data nil
      colorcomp-mode-map nil
      colorcomp-labels ["Red" "Green" "Blue"])

(defun colorcomp-pp (data)
  (if data
      (let ((comp (aref colorcomp-data data)))
        (insert (aref colorcomp-labels data) "\t: #x"
                (format "%02X" comp) " "
```

```
                (make-string (ash comp -2) ?#) "\n"))
    (let ((cstr (format "#%02X%02X%02X"
                        (aref colorcomp-data 0)
                        (aref colorcomp-data 1)
                        (aref colorcomp-data 2)))
          (samp " (sample text) "))
      (insert "Color\t: "
              (propertize samp 'face
                          `(foreground-color . ,cstr))
              (propertize samp 'face
                          `(background-color . ,cstr))
              "\n"))))

(defun colorcomp (color)
  "Allow fiddling with COLOR in a new buffer.
The buffer is in Color Components mode."
  (interactive "sColor (name or #RGB or #RRGGBB): ")
  (when (string= "" color)
    (setq color "green"))
  (unless (color-values color)
    (error "No such color: %S" color))
  (switch-to-buffer
   (generate-new-buffer (format "originally: %s" color)))
  (kill-all-local-variables)
  (setq major-mode 'colorcomp-mode
        mode-name "Color Components")
  (use-local-map colorcomp-mode-map)
  (erase-buffer)
  (buffer-disable-undo)
  (let ((data (apply 'vector (mapcar (lambda (n) (ash n -8))
                                     (color-values color))))
        (ewoc (ewoc-create 'colorcomp-pp
                           "\nColor Components\n\n"
                           (substitute-command-keys
                            "\n\\{colorcomp-mode-map}"))))
    (set (make-local-variable 'colorcomp-data) data)
    (set (make-local-variable 'colorcomp-ewoc) ewoc)
    (ewoc-enter-last ewoc 0)
    (ewoc-enter-last ewoc 1)
    (ewoc-enter-last ewoc 2)
    (ewoc-enter-last ewoc nil)))
```

This example can be extended to be a "color selection widget" (in other words, the controller part of the "model/view/controller" design paradigm) by defining commands to modify `colorcomp-data` and to "finish" the selection process, and a keymap to tie it all together conveniently.

```
(defun colorcomp-mod (index limit delta)
  (let ((cur (aref colorcomp-data index)))
```

```
    (unless (= limit cur)
      (aset colorcomp-data index (+ cur delta)))
    (ewoc-invalidate
     colorcomp-ewoc
     (ewoc-nth colorcomp-ewoc index)
     (ewoc-nth colorcomp-ewoc -1))))

(defun colorcomp-R-more () (interactive) (colorcomp-mod 0 255 1))
(defun colorcomp-G-more () (interactive) (colorcomp-mod 1 255 1))
(defun colorcomp-B-more () (interactive) (colorcomp-mod 2 255 1))
(defun colorcomp-R-less () (interactive) (colorcomp-mod 0 0 -1))
(defun colorcomp-G-less () (interactive) (colorcomp-mod 1 0 -1))
(defun colorcomp-B-less () (interactive) (colorcomp-mod 2 0 -1))

(defun colorcomp-copy-as-kill-and-exit ()
  "Copy the color components into the kill ring and kill the buffer.
The string is formatted #RRGGBB (hash followed by six hex digits)."
  (interactive)
  (kill-new (format "#%02X%02X%02X"
                    (aref colorcomp-data 0)
                    (aref colorcomp-data 1)
                    (aref colorcomp-data 2)))
  (kill-buffer nil))

(setq colorcomp-mode-map
      (let ((m (make-sparse-keymap)))
        (suppress-keymap m)
        (define-key m "i" 'colorcomp-R-less)
        (define-key m "o" 'colorcomp-R-more)
        (define-key m "k" 'colorcomp-G-less)
        (define-key m "l" 'colorcomp-G-more)
        (define-key m "," 'colorcomp-B-less)
        (define-key m "." 'colorcomp-B-more)
        (define-key m " " 'colorcomp-copy-as-kill-and-exit)
        m))
```

Note that we never modify the data in each node, which is fixed when the ewoc is created to be either `nil` or an index into the vector `colorcomp-data`, the actual color components.

38.19 Blinking Parentheses

This section describes the mechanism by which Emacs shows a matching open parenthesis when the user inserts a close parenthesis.

`blink-paren-function` [Variable]

The value of this variable should be a function (of no arguments) to be called whenever a character with close parenthesis syntax is inserted. The value of `blink-paren-function` may be `nil`, in which case nothing is done.

`blink-matching-paren` [User Option]

If this variable is `nil`, then `blink-matching-open` does nothing.

`blink-matching-paren-distance` [User Option]

This variable specifies the maximum distance to scan for a matching parenthesis before giving up.

`blink-matching-delay` [User Option]

This variable specifies the number of seconds for the cursor to remain at the matching parenthesis. A fraction of a second often gives good results, but the default is 1, which works on all systems.

`blink-matching-open` [Command]

This function is the default value of `blink-paren-function`. It assumes that point follows a character with close parenthesis syntax and moves the cursor momentarily to the matching opening character. If that character is not already on the screen, it displays the character's context in the echo area. To avoid long delays, this function does not search farther than `blink-matching-paren-distance` characters.

Here is an example of calling this function explicitly.

```
(defun interactive-blink-matching-open ()
  "Indicate momentarily the start of sexp before point."
  (interactive)
  (let ((blink-matching-paren-distance
         (buffer-size))
        (blink-matching-paren t))
    (blink-matching-open)))
```

38.20 Character Display

This section describes how characters are actually displayed by Emacs. Typically, a character is displayed as a *glyph* (a graphical symbol which occupies one character position on the screen), whose appearance corresponds to the character itself. For example, the character '`a`' (character code 97) is displayed as '`a`'. Some characters, however, are displayed specially. For example, the formfeed character (character code 12) is usually displayed as a sequence of two glyphs, '`^L`', while the newline character (character code 10) starts a new screen line.

You can modify how each character is displayed by defining a *display table*, which maps each character code into a sequence of glyphs. See Section 38.20.2 [Display Tables], page 377.

38.20.1 Usual Display Conventions

Here are the conventions for displaying each character code (in the absence of a display table, which can override these conventions).

- The *printable ASCII characters*, character codes 32 through 126 (consisting of numerals, English letters, and symbols like '`#`') are displayed literally.
- The tab character (character code 9) displays as whitespace stretching up to the next tab stop column. See Section "Text Display" in *The GNU Emacs Manual*. The variable `tab-width` controls the number of spaces per tab stop (see below).
- The newline character (character code 10) has a special effect: it ends the preceding line and starts a new line.
- The non-printable *ASCII control characters*—character codes 0 through 31, as well as the `DEL` character (character code 127)—display in one of two ways according to the variable `ctl-arrow`. If this variable is non-`nil` (the default), these characters are displayed as sequences of two glyphs, where the first glyph is '`^`' (a display table can specify a glyph to use instead of '`^`'); e.g. the `DEL` character is displayed as '`^?`'.

  If `ctl-arrow` is `nil`, these characters are displayed as octal escapes (see below).

This rule also applies to carriage return (character code 13), if that character appears in the buffer. But carriage returns usually do not appear in buffer text; they are eliminated as part of end-of-line conversion (see Section 33.9.1 [Coding System Basics], page 193).

- *Raw bytes* are non-ASCII characters with codes 128 through 255 (see Section 33.1 [Text Representations], page 182). These characters display as *octal escapes*: sequences of four glyphs, where the first glyph is the ASCII code for '\', and the others are digit characters representing the character code in octal. (A display table can specify a glyph to use instead of '\'.)
- Each non-ASCII character with code above 255 is displayed literally, if the terminal supports it. If the terminal does not support it, the character is said to be *glyphless*, and it is usually displayed using a placeholder glyph. For example, if a graphical terminal has no font for a character, Emacs usually displays a box containing the character code in hexadecimal. See Section 38.20.5 [Glyphless Chars], page 380.

The above display conventions apply even when there is a display table, for any character whose entry in the active display table is `nil`. Thus, when you set up a display table, you need only specify the characters for which you want special behavior.

The following variables affect how certain characters are displayed on the screen. Since they change the number of columns the characters occupy, they also affect the indentation functions. They also affect how the mode line is displayed; if you want to force redisplay of the mode line using the new values, call the function `force-mode-line-update` (see Section 23.4 [Mode Line Format], page 419, vol. 1).

`ctl-arrow` [User Option]
: This buffer-local variable controls how control characters are displayed. If it is non-`nil`, they are displayed as a caret followed by the character: '`^A`'. If it is `nil`, they are displayed as octal escapes: a backslash followed by three octal digits, as in '`\001`'.

`tab-width` [User Option]
: The value of this buffer-local variable is the spacing between tab stops used for displaying tab characters in Emacs buffers. The value is in units of columns, and the default is 8. Note that this feature is completely independent of the user-settable tab stops used by the command `tab-to-tab-stop`. See Section 32.17.5 [Indent Tabs], page 154.

38.20.2 Display Tables

A display table is a special-purpose char-table (see Section 6.6 [Char-Tables], page 92, vol. 1), with `display-table` as its subtype, which is used to override the usual character display conventions. This section describes how to make, inspect, and assign elements to a display table object.

`make-display-table` [Function]
: This creates and returns a display table. The table initially has `nil` in all elements.

The ordinary elements of the display table are indexed by character codes; the element at index *c* says how to display the character code *c*. The value should be `nil` (which means to display the character *c* according to the usual display conventions; see Section 38.20.1

[Usual Display], page 376), or a vector of glyph codes (which means to display the character *c* as those glyphs; see Section 38.20.4 [Glyphs], page 379).

Warning: if you use the display table to change the display of newline characters, the whole buffer will be displayed as one long "line".

The display table also has six "extra slots" which serve special purposes. Here is a table of their meanings; `nil` in any slot means to use the default for that slot, as stated below.

0 The glyph for the end of a truncated screen line (the default for this is '`$`'). See Section 38.20.4 [Glyphs], page 379. On graphical terminals, Emacs uses arrows in the fringes to indicate truncation, so the display table has no effect.

1 The glyph for the end of a continued line (the default is '`\`'). On graphical terminals, Emacs uses curved arrows in the fringes to indicate continuation, so the display table has no effect.

2 The glyph for indicating a character displayed as an octal character code (the default is '`\`').

3 The glyph for indicating a control character (the default is '`^`').

4 A vector of glyphs for indicating the presence of invisible lines (the default is '`...`'). See Section 38.7 [Selective Display], page 312.

5 The glyph used to draw the border between side-by-side windows (the default is '`|`'). See Section 28.5 [Splitting Windows], page 26. This takes effect only when there are no scroll bars; if scroll bars are supported and in use, a scroll bar separates the two windows.

For example, here is how to construct a display table that mimics the effect of setting `ctl-arrow` to a non-`nil` value (see Section 38.20.4 [Glyphs], page 379, for the function `make-glyph-code`):

```
(setq disptab (make-display-table))
(dotimes (i 32)
  (or (= i ?\t)
      (= i ?\n)
      (aset disptab i
            (vector (make-glyph-code ?^ 'escape-glyph)
                    (make-glyph-code (+ i 64) 'escape-glyph)))))
(aset disptab 127
      (vector (make-glyph-code ?^ 'escape-glyph)
              (make-glyph-code ?? 'escape-glyph))))
```

`display-table-slot` *display-table slot* [Function]
This function returns the value of the extra slot *slot* of *display-table*. The argument *slot* may be a number from 0 to 5 inclusive, or a slot name (symbol). Valid symbols are `truncation`, `wrap`, `escape`, `control`, `selective-display`, and `vertical-border`.

`set-display-table-slot` *display-table slot value* [Function]
This function stores *value* in the extra slot *slot* of *display-table*. The argument *slot* may be a number from 0 to 5 inclusive, or a slot name (symbol). Valid symbols are `truncation`, `wrap`, `escape`, `control`, `selective-display`, and `vertical-border`.

`describe-display-table` *display-table* [Function]
This function displays a description of the display table *display-table* in a help buffer.

`describe-current-display-table` [Command]
This command displays a description of the current display table in a help buffer.

38.20.3 Active Display Table

Each window can specify a display table, and so can each buffer. The window's display table, if there is one, takes precedence over the buffer's display table. If neither exists, Emacs tries to use the standard display table; if that is `nil`, Emacs uses the usual character display conventions (see Section 38.20.1 [Usual Display], page 376).

Note that display tables affect how the mode line is displayed, so if you want to force redisplay of the mode line using a new display table, call `force-mode-line-update` (see Section 23.4 [Mode Line Format], page 419, vol. 1).

`window-display-table` **&optional** *window* [Function]
This function returns *window*'s display table, or `nil` if there is none. The default for *window* is the selected window.

`set-window-display-table` *window table* [Function]
This function sets the display table of *window* to *table*. The argument *table* should be either a display table or `nil`.

`buffer-display-table` [Variable]
This variable is automatically buffer-local in all buffers; its value specifies the buffer's display table. If it is `nil`, there is no buffer display table.

`standard-display-table` [Variable]
The value of this variable is the standard display table, which is used when Emacs is displaying a buffer in a window with neither a window display table nor a buffer display table defined. Its default is `nil`.

The '`disp-table`' library defines several functions for changing the standard display table.

38.20.4 Glyphs

A *glyph* is a graphical symbol which occupies a single character position on the screen. Each glyph is represented in Lisp as a *glyph code*, which specifies a character and optionally a face to display it in (see Section 38.12 [Faces], page 325). The main use of glyph codes is as the entries of display tables (see Section 38.20.2 [Display Tables], page 377). The following functions are used to manipulate glyph codes:

`make-glyph-code` *char* **&optional** *face* [Function]
This function returns a glyph code representing char *char* with face *face*. If *face* is omitted or `nil`, the glyph uses the default face; in that case, the glyph code is an integer. If *face* is non-`nil`, the glyph code is not necessarily an integer object.

`glyph-char` *glyph* [Function]
This function returns the character of glyph code *glyph*.

`glyph-face` *glyph* [Function]
This function returns face of glyph code *glyph*, or `nil` if *glyph* uses the default face.

38.20.5 Glyphless Character Display

Glyphless characters are characters which are displayed in a special way, e.g. as a box containing a hexadecimal code, instead of being displayed literally. These include characters which are explicitly defined to be glyphless, as well as characters for which there is no available font (on a graphical display), and characters which cannot be encoded by the terminal's coding system (on a text terminal).

`glyphless-char-display` [Variable]
The value of this variable is a char-table which defines glyphless characters and how they are displayed. Each entry must be one of the following display methods:

`nil` Display the character in the usual way.

`zero-width`
Don't display the character.

`thin-space`
Display a thin space, 1-pixel wide on graphical displays, or 1-character wide on text terminals.

`empty-box`
Display an empty box.

`hex-code` Display a box containing the Unicode codepoint of the character, in hexadecimal notation.

an ASCII string
Display a box containing that string.

a cons cell `(`*graphical* `.` *text*`)`
Display with *graphical* on graphical displays, and with *text* on text terminals. Both *graphical* and *text* must be one of the display methods described above.

The `thin-space`, `empty-box`, `hex-code`, and ASCII string display methods are drawn with the `glyphless-char` face.

The char-table has one extra slot, which determines how to display any character that cannot be displayed with any available font, or cannot be encoded by the terminal's coding system. Its value should be one of the above display methods, except `zero-width` or a cons cell.

If a character has a non-`nil` entry in an active display table, the display table takes effect; in this case, Emacs does not consult `glyphless-char-display` at all.

`glyphless-char-display-control` [User Option]
This user option provides a convenient way to set `glyphless-char-display` for groups of similar characters. Do not set its value directly from Lisp code; the value takes effect only via a custom `:set` function (see Section 14.3 [Variable Definitions], page 193, vol. 1), which updates `glyphless-char-display`.

Its value should be an alist of elements `(group . method)`, where *group* is a symbol specifying a group of characters, and *method* is a symbol specifying how to display them.

group should be one of the following:

`c0-control`
: ASCII control characters `U+0000` to `U+001F`, excluding the newline and tab characters (normally displayed as escape sequences like '`^A`'; see Section "How Text Is Displayed" in *The GNU Emacs Manual*).

`c1-control`
: Non-ASCII, non-printing characters `U+0080` to `U+009F` (normally displayed as octal escape sequences like '`\230`').

`format-control`
: Characters of Unicode General Category 'Cf', such as '`U+200E`' (Left-to-Right Mark), but excluding characters that have graphic images, such as '`U+00AD`' (Soft Hyphen).

`no-font`
: Characters for there is no suitable font, or which cannot be encoded by the terminal's coding system.

The *method* symbol should be one of `zero-width`, `thin-space`, `empty-box`, or `hex-code`. These have the same meanings as in `glyphless-char-display`, above.

38.21 Beeping

This section describes how to make Emacs ring the bell (or blink the screen) to attract the user's attention. Be conservative about how often you do this; frequent bells can become irritating. Also be careful not to use just beeping when signaling an error is more appropriate (see Section 10.5.3 [Errors], page 128, vol. 1).

`ding` **&optional** *do-not-terminate* [Function]
: This function beeps, or flashes the screen (see `visible-bell` below). It also terminates any keyboard macro currently executing unless *do-not-terminate* is non-`nil`.

`beep` **&optional** *do-not-terminate* [Function]
: This is a synonym for `ding`.

`visible-bell` [User Option]
: This variable determines whether Emacs should flash the screen to represent a bell. Non-`nil` means yes, `nil` means no. This is effective on graphical displays, and on text terminals provided the terminal's Termcap entry defines the visible bell capability ('`vb`').

`ring-bell-function` [Variable]
: If this is non-`nil`, it specifies how Emacs should "ring the bell". Its value should be a function of no arguments. If this is non-`nil`, it takes precedence over the `visible-bell` variable.

38.22 Window Systems

Emacs works with several window systems, most notably the X Window System. Both Emacs and X use the term "window", but use it differently. An Emacs frame is a single window as far as X is concerned; the individual Emacs windows are not known to X at all.

`window-system` [Variable]

This terminal-local variable tells Lisp programs what window system Emacs is using for displaying the frame. The possible values are

- `x` Emacs is displaying the frame using X.
- `w32` Emacs is displaying the frame using native MS-Windows GUI.
- `ns` Emacs is displaying the frame using the Nextstep interface (used on GNUstep and Mac OS X).
- `pc` Emacs is displaying the frame using MS-DOS direct screen writes.
- `nil` Emacs is displaying the frame on a character-based terminal.

`initial-window-system` [Variable]

This variable holds the value of `window-system` used for the first frame created by Emacs during startup. (When Emacs is invoked with the '`--daemon`' option, it does not create any initial frames, so `initial-window-system` is `nil`. See Section "Initial Options" in *The GNU Emacs Manual*.)

`window-system` **&optional** *frame* [Function]

This function returns a symbol whose name tells what window system is used for displaying *frame* (which defaults to the currently selected frame). The list of possible symbols it returns is the same one documented for the variable `window-system` above.

Do *not* use `window-system` and `initial-window-system` as predicates or boolean flag variables, if you want to write code that works differently on text terminals and graphic displays. That is because `window-system` is not a good indicator of Emacs capabilities on a given display type. Instead, use `display-graphic-p` or any of the other `display-*-p` predicates described in Section 29.23 [Display Feature Testing], page 95.

`window-setup-hook` [Variable]

This variable is a normal hook which Emacs runs after handling the initialization files. Emacs runs this hook after it has completed loading your init file, the default initialization file (if any), and the terminal-specific Lisp code, and running the hook `term-setup-hook`.

This hook is used for internal purposes: setting up communication with the window system, and creating the initial window. Users should not interfere with it.

38.23 Bidirectional Display

Emacs can display text written in scripts, such as Arabic, Farsi, and Hebrew, whose natural ordering for horizontal text display runs from right to left. Furthermore, segments of Latin script and digits embedded in right-to-left text are displayed left-to-right, while segments of right-to-left script embedded in left-to-right text (e.g. Arabic or Hebrew text in comments

or strings in a program source file) are appropriately displayed right-to-left. We call such mixtures of left-to-right and right-to-left text *bidirectional text*. This section describes the facilities and options for editing and displaying bidirectional text.

Text is stored in Emacs buffers and strings in *logical* (or *reading*) order, i.e. the order in which a human would read each character. In right-to-left and bidirectional text, the order in which characters are displayed on the screen (called *visual order*) is not the same as logical order; the characters' screen positions do not increase monotonically with string or buffer position. In performing this *bidirectional reordering*, Emacs follows the Unicode Bidirectional Algorithm (a.k.a. UBA), which is described in Annex #9 of the Unicode standard (`http://www.unicode.org/reports/tr9/`). Emacs provides a "Full Bidirectionality" class implementation of the UBA.

`bidi-display-reordering` [Variable]

If the value of this buffer-local variable is non-`nil` (the default), Emacs performs bidirectional reordering for display. The reordering affects buffer text, as well as display strings and overlay strings from text and overlay properties in the buffer (see Section 38.9.2 [Overlay Properties], page 318, and see Section 38.15 [Display Property], page 350). If the value is `nil`, Emacs does not perform bidirectional reordering in the buffer.

The default value of `bidi-display-reordering` controls the reordering of strings which are not directly supplied by a buffer, including the text displayed in mode lines (see Section 23.4 [Mode Line Format], page 419, vol. 1) and header lines (see Section 23.4.7 [Header Lines], page 426, vol. 1).

Emacs never reorders the text of a unibyte buffer, even if `bidi-display-reordering` is non-`nil` in the buffer. This is because unibyte buffers contain raw bytes, not characters, and thus lack the directionality properties required for reordering. Therefore, to test whether text in a buffer will be reordered for display, it is not enough to test the value of `bidi-display-reordering` alone. The correct test is this:

```
(if (and enable-multibyte-characters
         bidi-display-reordering)
    ;; Buffer is being reordered for display
  )
```

However, unibyte display and overlay strings *are* reordered if their parent buffer is reordered. This is because plain-ASCII strings are stored by Emacs as unibyte strings. If a unibyte display or overlay string includes non-ASCII characters, these characters are assumed to have left-to-right direction.

Text covered by `display` text properties, by overlays with `display` properties whose value is a string, and by any other properties that replace buffer text, is treated as a single unit when it is reordered for display. That is, the entire chunk of text covered by these properties is reordered together. Moreover, the bidirectional properties of the characters in such a chunk of text are ignored, and Emacs reorders them as if they were replaced with a single character `U+FFFC`, known as the *Object Replacement Character*. This means that placing a display property over a portion of text may change the way that the surrounding text is reordered for display. To prevent this unexpected effect, always place such properties on text whose directionality is identical with text that surrounds it.

Each paragraph of bidirectional text has a *base direction*, either right-to-left or left-to-right. Left-to-right paragraphs are displayed beginning at the left margin of the window, and are truncated or continued when the text reaches the right margin. Right-to-left paragraphs are displayed beginning at the right margin, and are continued or truncated at the left margin.

By default, Emacs determines the base direction of each paragraph by looking at the text at its beginning. The precise method of determining the base direction is specified by the UBA; in a nutshell, the first character in a paragraph that has an explicit directionality determines the base direction of the paragraph. However, sometimes a buffer may need to force a certain base direction for its paragraphs. For example, buffers containing program source code should force all paragraphs to be displayed left-to-right. You can use following variable to do this:

`bidi-paragraph-direction` [Variable]

If the value of this buffer-local variable is the symbol `right-to-left` or `left-to-right`, all paragraphs in the buffer are assumed to have that specified direction. Any other value is equivalent to `nil` (the default), which means to determine the base direction of each paragraph from its contents.

Modes for program source code should set this to `left-to-right`. Prog mode does this by default, so modes derived from Prog mode do not need to set this explicitly (see Section 23.2.5 [Basic Major Modes], page 407, vol. 1).

`current-bidi-paragraph-direction` **&optional** *buffer* [Function]

This function returns the paragraph direction at point in the named *buffer*. The returned value is a symbol, either `left-to-right` or `right-to-left`. If *buffer* is omitted or `nil`, it defaults to the current buffer. If the buffer-local value of the variable `bidi-paragraph-direction` is non-`nil`, the returned value will be identical to that value; otherwise, the returned value reflects the paragraph direction determined dynamically by Emacs. For buffers whose value of `bidi-display-reordering` is `nil` as well as unibyte buffers, this function always returns `left-to-right`.

Bidirectional reordering can have surprising and unpleasant effects when two strings with bidirectional content are juxtaposed in a buffer, or otherwise programmatically concatenated into a string of text. A typical problematic case is when a buffer consists of sequences of text "fields" separated by whitespace or punctuation characters, like Buffer Menu mode or Rmail Summary Mode. Because the punctuation characters used as separators have *weak directionality*, they take on the directionality of surrounding text. As result, a numeric field that follows a field with bidirectional content can be displayed *to the left* of the preceding field, messing up the expected layout. There are several ways to avoid this problem:

– Append the special character `U+200E`, LEFT-TO-RIGHT MARK, or LRM, to the end of each field that may have bidirectional content, or prepend it to the beginning of the following field. The function `bidi-string-mark-left-to-right`, described below, comes in handy for this purpose. (In a right-to-left paragraph, use `U+200F`, RIGHT-TO-LEFT MARK, or RLM, instead.) This is one of the solutions recommended by the UBA.

– Include the tab character in the field separator. The tab character plays the role of *segment separator* in bidirectional reordering, causing the text on either side to be reordered separately.

– Separate fields with a `display` property or overlay with a property value of the form `(space . PROPS)` (see Section 38.15.2 [Specified Space], page 351). Emacs treats this display specification as a *paragraph separator*, and reorders the text on either side separately.

`bidi-string-mark-left-to-right` *string* [Function]
This function returns its argument *string*, possibly modified, such that the result can be safely concatenated with another string, or juxtaposed with another string in a buffer, without disrupting the relative layout of this string and the next one on display. If the string returned by this function is displayed as part of a left-to-right paragraph, it will always appear on display to the left of the text that follows it. The function works by examining the characters of its argument, and if any of those characters could cause reordering on display, the function appends the LRM character to the string. The appended LRM character is made invisible by giving it an `invisible` text property of `t` (see Section 38.6 [Invisible Text], page 309).

The reordering algorithm uses the bidirectional properties of the characters stored as their `bidi-class` property (see Section 33.5 [Character Properties], page 186). Lisp programs can change these properties by calling the `put-char-code-property` function. However, doing this requires a thorough understanding of the UBA, and is therefore not recommended. Any changes to the bidirectional properties of a character have global effect: they affect all Emacs frames and windows.

Similarly, the `mirroring` property is used to display the appropriate mirrored character in the reordered text. Lisp programs can affect the mirrored display by changing this property. Again, any such changes affect all of Emacs display.

39 Operating System Interface

This chapter is about starting and getting out of Emacs, access to values in the operating system environment, and terminal input, output.

See Section E.1 [Building Emacs], page 457, for related information. See Chapter 38 [Display], page 299, for additional operating system status information pertaining to the terminal and the screen.

39.1 Starting Up Emacs

This section describes what Emacs does when it is started, and how you can customize these actions.

39.1.1 Summary: Sequence of Actions at Startup

When Emacs is started up, it performs the following operations (see `normal-top-level` in '`startup.el`'):

1. It adds subdirectories to `load-path`, by running the file named '`subdirs.el`' in each directory in the list. Normally, this file adds the directory's subdirectories to the list, and those are scanned in their turn. The files '`subdirs.el`' are normally generated automatically when Emacs is installed.
2. It registers input methods by loading any '`leim-list.el`' file found in the `load-path`.
3. It sets the variable `before-init-time` to the value of `current-time` (see Section 39.5 [Time of Day], page 399). It also sets `after-init-time` to `nil`, which signals to Lisp programs that Emacs is being initialized.
4. It sets the language environment and the terminal coding system, if requested by environment variables such as `LANG`.
5. It does some basic parsing of the command-line arguments.
6. If not running in batch mode, it initializes the window system that the variable `initial-window-system` specifies (see Section 38.22 [Window Systems], page 382). The initialization function for each supported window system is specified by `window-system-initialization-alist`. If the value of `initial-window-system` is *windowsystem*, then the appropriate initialization function is defined in the file '`term/`*windowsystem*`-win.el`'. This file should have been compiled into the Emacs executable when it was built.
7. It runs the normal hook `before-init-hook`.
8. If appropriate, it creates a graphical frame. This is not done if the options '`--batch`' or '`--daemon`' were specified.
9. It initializes the initial frame's faces, and sets up the menu bar and tool bar if needed. If graphical frames are supported, it sets up the tool bar even if the current frame is not a graphical one, since a graphical frame may be created later on.
10. It use `custom-reevaluate-setting` to re-initialize the members of the list `custom-delayed-init-variables`. These are any pre-loaded user options whose default value depends on the run-time, rather than build-time, context. See Section E.1 [Building Emacs], page 457.

11. It loads the library '`site-start`', if it exists. This is not done if the options '`-Q`' or '`--no-site-file`' were specified.
12. It loads your init file (see Section 39.1.2 [Init File], page 389). This is not done if the options '`-q`', '`-Q`', or '`--batch`' were specified. If the '`-u`' option was specified, Emacs looks for the init file in that user's home directory instead.
13. It loads the library '`default`', if it exists. This is not done if `inhibit-default-init` is non-`nil`, nor if the options '`-q`', '`-Q`', or '`--batch`' were specified.
14. It loads your abbrevs from the file specified by `abbrev-file-name`, if that file exists and can be read (see Section 36.3 [Abbrev Files], page 252). This is not done if the option '`--batch`' was specified.
15. If `package-enable-at-startup` is non-`nil`, it calls the function `package-initialize` to activate any optional Emacs Lisp package that has been installed. See Section 40.1 [Packaging Basics], page 418.
16. It sets the variable `after-init-time` to the value of `current-time`. This variable was set to `nil` earlier; setting it to the current time signals that the initialization phase is over, and, together with `before-init-time`, provides the measurement of how long it took.
17. It runs the normal hook `after-init-hook`.
18. If the buffer '`*scratch*`' exists and is still in Fundamental mode (as it should be by default), it sets its major mode according to `initial-major-mode`.
19. If started on a text terminal, it loads the terminal-specific Lisp library, which is specified by the variable `term-file-prefix` (see Section 39.1.3 [Terminal-Specific], page 390). This is not done in `--batch` mode, nor if `term-file-prefix` is `nil`.
20. It displays the initial echo area message, unless you have suppressed that with `inhibit-startup-echo-area-message`.
21. It processes any command-line options that were not handled earlier.
22. It now exits if the option `--batch` was specified.
23. If `initial-buffer-choice` is a string, it visits the file with that name. If the '`*scratch*`' buffer exists and is empty, it inserts `initial-scratch-message` into that buffer.
24. It runs `emacs-startup-hook` and then `term-setup-hook`.
25. It calls `frame-notice-user-settings`, which modifies the parameters of the selected frame according to whatever the init files specify.
26. It runs `window-setup-hook`. See Section 38.22 [Window Systems], page 382.
27. It displays the *startup screen*, which is a special buffer that contains information about copyleft and basic Emacs usage. This is not done if `inhibit-startup-screen` or `initial-buffer-choice` are non-`nil`, or if the '`--no-splash`' or '`-Q`' command-line options were specified.
28. If the option `--daemon` was specified, it calls `server-start` and detaches from the controlling terminal. See Section "Emacs Server" in *The GNU Emacs Manual*.
29. If started by the X session manager, it calls `emacs-session-restore` passing it as argument the ID of the previous session. See Section 39.17 [Session Management], page 414.

The following options affect some aspects of the startup sequence.

`inhibit-startup-screen` [User Option]
: This variable, if non-`nil`, inhibits the startup screen. In that case, Emacs typically displays the '`*scratch*`' buffer; but see `initial-buffer-choice`, below.

  Do not set this variable in the init file of a new user, or in a way that affects more than one user, as that would prevent new users from receiving information about copyleft and basic Emacs usage.

  `inhibit-startup-message` and `inhibit-splash-screen` are aliases for this variable.

`initial-buffer-choice` [User Option]
: If non-`nil`, this variable is a string that specifies a file or directory for Emacs to display after starting up, instead of the startup screen.

`inhibit-startup-echo-area-message` [User Option]
: This variable controls the display of the startup echo area message. You can suppress the startup echo area message by adding text with this form to your init file:

  ```
  (setq inhibit-startup-echo-area-message
        "your-login-name")
  ```

  Emacs explicitly checks for an expression as shown above in your init file; your login name must appear in the expression as a Lisp string constant. You can also use the Customize interface. Other methods of setting `inhibit-startup-echo-area-message` to the same value do not inhibit the startup message. This way, you can easily inhibit the message for yourself if you wish, but thoughtless copying of your init file will not inhibit the message for someone else.

`initial-scratch-message` [User Option]
: This variable, if non-`nil`, should be a string, which is inserted into the '`*scratch*`' buffer when Emacs starts up. If it is `nil`, the '`*scratch*`' buffer is empty.

The following command-line options affect some aspects of the startup sequence. See Section "Initial Options" in *The GNU Emacs Manual*.

`--no-splash`
: Do not display a splash screen.

`--batch` Run without an interactive terminal. See Section 39.16 [Batch Mode], page 413.

`--daemon` Do not initialize any display; just start a server in the background.

`--no-init-file`
`-Q` Do not load either the init file, or the '`default`' library.

`--no-site-file`
: Do not load the '`site-start`' library.

`--quick`
`-Q` Equivalent to '`-q --no-site-file --no-splash`'.

39.1.2 The Init File

When you start Emacs, it normally attempts to load your *init file*. This is either a file named '`.emacs`' or '`.emacs.el`' in your home directory, or a file named '`init.el`' in a subdirectory named '`.emacs.d`' in your home directory.

The command-line switches '`-q`', '`-Q`', and '`-u`' control whether and where to find the init file; '`-q`' (and the stronger '`-Q`') says not to load an init file, while '`-u user`' says to load *user*'s init file instead of yours. See Section "Entering Emacs" in *The GNU Emacs Manual*. If neither option is specified, Emacs uses the `LOGNAME` environment variable, or the `USER` (most systems) or `USERNAME` (MS systems) variable, to find your home directory and thus your init file; this way, even if you have su'd, Emacs still loads your own init file. If those environment variables are absent, though, Emacs uses your user-id to find your home directory.

An Emacs installation may have a *default init file*, which is a Lisp library named '`default.el`'. Emacs finds this file through the standard search path for libraries (see Section 15.1 [How Programs Do Loading], page 209, vol. 1). The Emacs distribution does not come with this file; it is intended for local customizations. If the default init file exists, it is loaded whenever you start Emacs. But your own personal init file, if any, is loaded first; if it sets `inhibit-default-init` to a non-`nil` value, then Emacs does not subsequently load the '`default.el`' file. In batch mode, or if you specify '`-q`' (or '`-Q`'), Emacs loads neither your personal init file nor the default init file.

Another file for site-customization is '`site-start.el`'. Emacs loads this *before* the user's init file. You can inhibit the loading of this file with the option '`--no-site-file`'.

`site-run-file` [User Option]
: This variable specifies the site-customization file to load before the user's init file. Its normal value is `"site-start"`. The only way you can change it with real effect is to do so before dumping Emacs.

See Section "Init File Examples" in *The GNU Emacs Manual*, for examples of how to make various commonly desired customizations in your '`.emacs`' file.

`inhibit-default-init` [User Option]
: If this variable is non-`nil`, it prevents Emacs from loading the default initialization library file. The default value is `nil`.

`before-init-hook` [Variable]
: This normal hook is run, once, just before loading all the init files ('`site-start.el`', your init file, and '`default.el`'). (The only way to change it with real effect is before dumping Emacs.)

`after-init-hook` [Variable]
: This normal hook is run, once, just after loading all the init files ('`site-start.el`', your init file, and '`default.el`'), before loading the terminal-specific library (if started on a text terminal) and processing the command-line action arguments.

`emacs-startup-hook` [Variable]
: This normal hook is run, once, just after handling the command line arguments, just before `term-setup-hook`. In batch mode, Emacs does not run either of these hooks.

`user-init-file` [Variable]
: This variable holds the absolute file name of the user's init file. If the actual init file loaded is a compiled file, such as '`.emacs.elc`', the value refers to the corresponding source file.

`user-emacs-directory` [Variable]
: This variable holds the name of the '`.emacs.d`' directory. It is '`~/.emacs.d`' on all platforms but MS-DOS.

39.1.3 Terminal-Specific Initialization

Each terminal type can have its own Lisp library that Emacs loads when run on that type of terminal. The library's name is constructed by concatenating the value of the variable `term-file-prefix` and the terminal type (specified by the environment variable `TERM`). Normally, `term-file-prefix` has the value `"term/"`; changing this is not recommended. Emacs finds the file in the normal manner, by searching the `load-path` directories, and trying the '`.elc`' and '`.el`' suffixes.

The usual role of a terminal-specific library is to enable special keys to send sequences that Emacs can recognize. It may also need to set or add to `input-decode-map` if the Termcap or Terminfo entry does not specify all the terminal's function keys. See Section 39.12 [Terminal Input], page 409.

When the name of the terminal type contains a hyphen or underscore, and no library is found whose name is identical to the terminal's name, Emacs strips from the terminal's name the last hyphen or underscore and everything that follows it, and tries again. This process is repeated until Emacs finds a matching library, or until there are no more hyphens or underscores in the name (i.e. there is no terminal-specific library). For example, if the terminal name is '`xterm-256color`' and there is no '`term/xterm-256color.el`' library, Emacs tries to load '`term/xterm.el`'. If necessary, the terminal library can evaluate `(getenv "TERM")` to find the full name of the terminal type.

Your init file can prevent the loading of the terminal-specific library by setting the variable `term-file-prefix` to `nil`. This feature is useful when experimenting with your own peculiar customizations.

You can also arrange to override some of the actions of the terminal-specific library by setting the variable `term-setup-hook`. This is a normal hook that Emacs runs at the end of its initialization, after loading both your init file and any terminal-specific libraries. You could use this hook to define initializations for terminals that do not have their own libraries. See Section 23.1 [Hooks], page 396, vol. 1.

`term-file-prefix` [Variable]
: If the value of this variable is non-`nil`, Emacs loads a terminal-specific initialization file as follows:

```
(load (concat term-file-prefix (getenv "TERM")))
```

: You may set the `term-file-prefix` variable to `nil` in your init file if you do not wish to load the terminal-initialization file.

: On MS-DOS, Emacs sets the `TERM` environment variable to '`internal`'.

`term-setup-hook` [Variable]

This variable is a normal hook that Emacs runs after loading your init file, the default initialization file (if any) and the terminal-specific Lisp file.

You can use `term-setup-hook` to override the definitions made by a terminal-specific file.

For a related feature, see Section 38.22 [Window Systems], page 382.

39.1.4 Command-Line Arguments

You can use command-line arguments to request various actions when you start Emacs. Note that the recommended way of using Emacs is to start it just once, after logging in, and then do all editing in the same Emacs session (see Section "Entering Emacs" in *The GNU Emacs Manual*). For this reason, you might not use command-line arguments very often; nonetheless, they can be useful when invoking Emacs from session scripts or debugging Emacs. This section describes how Emacs processes command-line arguments.

`command-line` [Function]

This function parses the command line that Emacs was called with, processes it, and (amongst other things) loads the user's init file and displays the startup messages.

`command-line-processed` [Variable]

The value of this variable is `t` once the command line has been processed.

If you redump Emacs by calling `dump-emacs`, you may wish to set this variable to `nil` first in order to cause the new dumped Emacs to process its new command-line arguments.

`command-switch-alist` [Variable]

This variable is an alist of user-defined command-line options and associated handler functions. By default it is empty, but you can add elements if you wish.

A *command-line option* is an argument on the command line, which has the form:

```
-option
```

The elements of the `command-switch-alist` look like this:

```
(option . handler-function)
```

The CAR, *option*, is a string, the name of a command-line option (not including the initial hyphen). The *handler-function* is called to handle *option*, and receives the option name as its sole argument.

In some cases, the option is followed in the command line by an argument. In these cases, the *handler-function* can find all the remaining command-line arguments in the variable `command-line-args-left`. (The entire list of command-line arguments is in `command-line-args`.)

The command-line arguments are parsed by the `command-line-1` function in the '`startup.el`' file. See also Section "Command Line Arguments for Emacs Invocation" in *The GNU Emacs Manual*.

`command-line-args` [Variable]

The value of this variable is the list of command-line arguments passed to Emacs.

`command-line-args-left` [Variable]
: The value of this variable is the list of command-line arguments that have not yet been processed.

`command-line-functions` [Variable]
: This variable's value is a list of functions for handling an unrecognized command-line argument. Each time the next argument to be processed has no special meaning, the functions in this list are called, in order of appearance, until one of them returns a non-`nil` value.

  These functions are called with no arguments. They can access the command-line argument under consideration through the variable `argi`, which is bound temporarily at this point. The remaining arguments (not including the current one) are in the variable `command-line-args-left`.

  When a function recognizes and processes the argument in `argi`, it should return a non-`nil` value to say it has dealt with that argument. If it has also dealt with some of the following arguments, it can indicate that by deleting them from `command-line-args-left`.

  If all of these functions return `nil`, then the argument is treated as a file name to visit.

39.2 Getting Out of Emacs

There are two ways to get out of Emacs: you can kill the Emacs job, which exits permanently, or you can suspend it, which permits you to reenter the Emacs process later. (In a graphical environment, you can of course simply switch to another application without doing anything special to Emacs, then switch back to Emacs when you want.)

39.2.1 Killing Emacs

Killing Emacs means ending the execution of the Emacs process. If you started Emacs from a terminal, the parent process normally resumes control. The low-level primitive for killing Emacs is `kill-emacs`.

`kill-emacs` **&optional** *exit-data* [Command]
: This command calls the hook `kill-emacs-hook`, then exits the Emacs process and kills it.

  If *exit-data* is an integer, that is used as the exit status of the Emacs process. (This is useful primarily in batch operation; see Section 39.16 [Batch Mode], page 413.)

  If *exit-data* is a string, its contents are stuffed into the terminal input buffer so that the shell (or whatever program next reads input) can read them.

The `kill-emacs` function is normally called via the higher-level command `C-x C-c` (`save-buffers-kill-terminal`). See Section "Exiting" in *The GNU Emacs Manual*. It is also called automatically if Emacs receives a `SIGTERM` or `SIGHUP` operating system signal (e.g. when the controlling terminal is disconnected), or if it receives a `SIGINT` signal while running in batch mode (see Section 39.16 [Batch Mode], page 413).

`kill-emacs-hook` [Variable]
: This normal hook is run by `kill-emacs`, before it kills Emacs.

Because `kill-emacs` can be called in situations where user interaction is impossible (e.g. when the terminal is disconnected), functions on this hook should not attempt to interact with the user. If you want to interact with the user when Emacs is shutting down, use `kill-emacs-query-functions`, described below.

When Emacs is killed, all the information in the Emacs process, aside from files that have been saved, is lost. Because killing Emacs inadvertently can lose a lot of work, the `save-buffers-kill-terminal` command queries for confirmation if you have buffers that need saving or subprocesses that are running. It also runs the abnormal hook `kill-emacs-query-functions`:

`kill-emacs-query-functions` [Variable]

When `save-buffers-kill-terminal` is killing Emacs, it calls the functions in this hook, after asking the standard questions and before calling `kill-emacs`. The functions are called in order of appearance, with no arguments. Each function can ask for additional confirmation from the user. If any of them returns `nil`, `save-buffers-kill-emacs` does not kill Emacs, and does not run the remaining functions in this hook. Calling `kill-emacs` directly does not run this hook.

39.2.2 Suspending Emacs

On text terminals, it is possible to *suspend Emacs*, which means stopping Emacs temporarily and returning control to its superior process, which is usually the shell. This allows you to resume editing later in the same Emacs process, with the same buffers, the same kill ring, the same undo history, and so on. To resume Emacs, use the appropriate command in the parent shell—most likely `fg`.

Suspending works only on a terminal device from which the Emacs session was started. We call that device the *controlling terminal* of the session. Suspending is not allowed if the controlling terminal is a graphical terminal. Suspending is usually not relevant in graphical environments, since you can simply switch to another application without doing anything special to Emacs.

Some operating systems (those without `SIGTSTP`, or MS-DOS) do not support suspension of jobs; on these systems, "suspension" actually creates a new shell temporarily as a subprocess of Emacs. Then you would exit the shell to return to Emacs.

`suspend-emacs` **&optional** *string* [Command]

This function stops Emacs and returns control to the superior process. If and when the superior process resumes Emacs, `suspend-emacs` returns `nil` to its caller in Lisp.

This function works only on the controlling terminal of the Emacs session; to relinquish control of other tty devices, use `suspend-tty` (see below). If the Emacs session uses more than one terminal, you must delete the frames on all the other terminals before suspending Emacs, or this function signals an error. See Section 29.2 [Multiple Terminals], page 67.

If *string* is non-`nil`, its characters are sent to Emacs's superior shell, to be read as terminal input. The characters in *string* are not echoed by the superior shell; only the results appear.

Before suspending, `suspend-emacs` runs the normal hook `suspend-hook`. After the user resumes Emacs, `suspend-emacs` runs the normal hook `suspend-resume-hook`. See Section 23.1 [Hooks], page 396, vol. 1.

The next redisplay after resumption will redraw the entire screen, unless the variable `no-redraw-on-reenter` is non-`nil`. See Section 38.1 [Refresh Screen], page 299.

Here is an example of how you could use these hooks:

```
(add-hook 'suspend-hook
          (lambda () (or (y-or-n-p "Really suspend? ")
                         (error "Suspend canceled"))))
(add-hook 'suspend-resume-hook (lambda () (message "Resumed!")
                                  (sit-for 2)))
```

Here is what you would see upon evaluating `(suspend-emacs "pwd")`:

```
---------- Buffer: Minibuffer ----------
Really suspend? y
---------- Buffer: Minibuffer ----------

---------- Parent Shell ----------
bash$ /home/username
bash$ fg

---------- Echo Area ----------
Resumed!
```

Note that '`pwd`' is not echoed after Emacs is suspended. But it is read and executed by the shell.

`suspend-hook` [Variable]
This variable is a normal hook that Emacs runs before suspending.

`suspend-resume-hook` [Variable]
This variable is a normal hook that Emacs runs on resuming after a suspension.

`suspend-tty` **&optional** *tty* [Function]
If *tty* specifies a terminal device used by Emacs, this function relinquishes the device and restores it to its prior state. Frames that used the device continue to exist, but are not updated and Emacs doesn't read input from them. *tty* can be a terminal object, a frame (meaning the terminal for that frame), or `nil` (meaning the terminal for the selected frame). See Section 29.2 [Multiple Terminals], page 67.

If *tty* is already suspended, this function does nothing.

This function runs the hook `suspend-tty-functions`, passing the terminal object as an argument to each function.

`resume-tty` **&optional** *tty* [Function]
This function resumes the previously suspended terminal device *tty*; where *tty* has the same possible values as it does for `suspend-tty`.

This function reopens the terminal device, re-initializes it, and redraws it with that terminal's selected frame. It then runs the hook `resume-tty-functions`, passing the terminal object as an argument to each function.

If the same device is already used by another Emacs terminal, this function signals an error. If *tty* is not suspended, this function does nothing.

`controlling-tty-p` **&optional** *tty* [Function]
: This function returns non-`nil` if *tty* is the controlling terminal of the Emacs session; *tty* can be a terminal object, a frame (meaning the terminal for that frame), or `nil` (meaning the terminal for the selected frame).

`suspend-frame` [Command]
: This command *suspends* a frame. For GUI frames, it calls `iconify-frame` (see Section 29.10 [Visibility of Frames], page 85); for frames on text terminals, it calls either `suspend-emacs` or `suspend-tty`, depending on whether the frame is displayed on the controlling terminal device or not.

39.3 Operating System Environment

Emacs provides access to variables in the operating system environment through various functions. These variables include the name of the system, the user's UID, and so on.

`system-configuration` [Variable]
: This variable holds the standard GNU configuration name for the hardware/software configuration of your system, as a string. For example, a typical value for a 64-bit GNU/Linux system is '`"x86_64-unknown-linux-gnu"`'.

`system-type` [Variable]
: The value of this variable is a symbol indicating the type of operating system Emacs is running on. The possible values are:

  `aix` IBM's AIX.

  `berkeley-unix`
  : Berkeley BSD and its variants.

  `cygwin` Cygwin, a Posix layer on top of MS-Windows.

  `darwin` Darwin (Mac OS X).

  `gnu` The GNU system (using the GNU kernel, which consists of the HURD and Mach).

  `gnu/linux`
  : A GNU/Linux system—that is, a variant GNU system, using the Linux kernel. (These systems are the ones people often call "Linux", but actually Linux is just the kernel, not the whole system.)

  `gnu/kfreebsd`
  : A GNU (glibc-based) system with a FreeBSD kernel.

  `hpux` Hewlett-Packard HPUX operating system.

  `irix` Silicon Graphics Irix system.

  `ms-dos` Microsoft's DOS. Emacs compiled with DJGPP for MS-DOS binds `system-type` to `ms-dos` even when you run it on MS-Windows.

  `usg-unix-v`
  : AT&T Unix System V.

`windows-nt`
: Microsoft Windows NT, 9X and later. The value of `system-type` is always `windows-nt`, e.g. even on Windows 7.

We do not wish to add new symbols to make finer distinctions unless it is absolutely necessary! In fact, we hope to eliminate some of these alternatives in the future. If you need to make a finer distinction than `system-type` allows for, you can test `system-configuration`, e.g. against a regexp.

`system-name` [Function]
: This function returns the name of the machine you are running on, as a string.

The symbol `system-name` is a variable as well as a function. In fact, the function returns whatever value the variable `system-name` currently holds. Thus, you can set the variable `system-name` in case Emacs is confused about the name of your system. The variable is also useful for constructing frame titles (see Section 29.5 [Frame Titles], page 81).

`mail-host-address` [User Option]
: If this variable is non-`nil`, it is used instead of `system-name` for purposes of generating email addresses. For example, it is used when constructing the default value of `user-mail-address`. See Section 39.4 [User Identification], page 398. (Since this is done when Emacs starts up, the value actually used is the one saved when Emacs was dumped. See Section E.1 [Building Emacs], page 457.)

`getenv` *var* **&optional** *frame* [Command]
: This function returns the value of the environment variable *var*, as a string. *var* should be a string. If *var* is undefined in the environment, `getenv` returns `nil`. It returns '`""`' if *var* is set but null. Within Emacs, a list of environment variables and their values is kept in the variable `process-environment`.

```
(getenv "USER")
     ⇒ "lewis"
```

The shell command `printenv` prints all or part of the environment:

```
bash$ printenv
PATH=/usr/local/bin:/usr/bin:/bin
USER=lewis
TERM=xterm
SHELL=/bin/bash
HOME=/home/lewis
...
```

`setenv` *variable* **&optional** *value substitute* [Command]
: This command sets the value of the environment variable named *variable* to *value*. *variable* should be a string. Internally, Emacs Lisp can handle any string. However, normally *variable* should be a valid shell identifier, that is, a sequence of letters, digits and underscores, starting with a letter or underscore. Otherwise, errors may occur if subprocesses of Emacs try to access the value of *variable*. If *value* is omitted or `nil` (or, interactively, with a prefix argument), `setenv` removes *variable* from the environment. Otherwise, *value* should be a string.

If the optional argument *substitute* is non-`nil`, Emacs calls the function `substitute-env-vars` to expand any environment variables in *value*.

`setenv` works by modifying `process-environment`; binding that variable with `let` is also reasonable practice.

`setenv` returns the new value of *variable*, or `nil` if it removed *variable* from the environment.

`process-environment` [Variable]

This variable is a list of strings, each describing one environment variable. The functions `getenv` and `setenv` work by means of this variable.

```
process-environment
⇒ ("PATH=/usr/local/bin:/usr/bin:/bin"
    "USER=lewis"
    "TERM=xterm"
    "SHELL=/bin/bash"
    "HOME=/home/lewis"
    ...)
```

If `process-environment` contains "duplicate" elements that specify the same environment variable, the first of these elements specifies the variable, and the other "duplicates" are ignored.

`initial-environment` [Variable]

This variable holds the list of environment variables Emacs inherited from its parent process when Emacs started.

`path-separator` [Variable]

This variable holds a string that says which character separates directories in a search path (as found in an environment variable). Its value is `":"` for Unix and GNU systems, and `";"` for MS systems.

`parse-colon-path` *path* [Function]

This function takes a search path string such as the value of the `PATH` environment variable, and splits it at the separators, returning a list of directory names. `nil` in this list means the current directory. Although the function's name says "colon", it actually uses the value of `path-separator`.

```
(parse-colon-path ":/foo:/bar")
     ⇒ (nil "/foo/" "/bar/")
```

`invocation-name` [Variable]

This variable holds the program name under which Emacs was invoked. The value is a string, and does not include a directory name.

`invocation-directory` [Variable]

This variable holds the directory from which the Emacs executable was invoked, or `nil` if that directory cannot be determined.

`installation-directory` [Variable]

If non-`nil`, this is a directory within which to look for the '`lib-src`' and '`etc`' subdirectories. In an installed Emacs, it is normally `nil`. It is non-`nil` when Emacs can't find those directories in their standard installed locations, but can find them

in a directory related somehow to the one containing the Emacs executable (i.e., `invocation-directory`).

`load-average` **&optional** *use-float* [Function]
This function returns the current 1-minute, 5-minute, and 15-minute system load averages, in a list. The load average indicates the number of processes trying to run on the system.

By default, the values are integers that are 100 times the system load averages, but if *use-float* is non-`nil`, then they are returned as floating point numbers without multiplying by 100.

If it is impossible to obtain the load average, this function signals an error. On some platforms, access to load averages requires installing Emacs as setuid or setgid so that it can read kernel information, and that usually isn't advisable.

If the 1-minute load average is available, but the 5- or 15-minute averages are not, this function returns a shortened list containing the available averages.

```
(load-average)
     ⇒ (169 48 36)
(load-average t)
     ⇒ (1.69 0.48 0.36)
```

The shell command `uptime` returns similar information.

`emacs-pid` [Function]
This function returns the process ID of the Emacs process, as an integer.

`tty-erase-char` [Variable]
This variable holds the erase character that was selected in the system's terminal driver, before Emacs was started.

39.4 User Identification

`init-file-user` [Variable]
This variable says which user's init files should be used by Emacs—or `nil` if none. `""` stands for the user who originally logged in. The value reflects command-line options such as '`-q`' or '`-u user`'.

Lisp packages that load files of customizations, or any other sort of user profile, should obey this variable in deciding where to find it. They should load the profile of the user name found in this variable. If `init-file-user` is `nil`, meaning that the '`-q`' option was used, then Lisp packages should not load any customization files or user profile.

`user-mail-address` [User Option]
This holds the nominal email address of the user who is using Emacs. Emacs normally sets this variable to a default value after reading your init files, but not if you have already set it. So you can set the variable to some other value in your init file if you do not want to use the default value.

`user-login-name` **&optional** *uid* [Function]
: This function returns the name under which the user is logged in. It uses the environment variables `LOGNAME` or `USER` if either is set. Otherwise, the value is based on the effective UID, not the real UID.

  If you specify *uid* (a number), the result is the user name that corresponds to *uid*, or `nil` if there is no such user.

`user-real-login-name` [Function]
: This function returns the user name corresponding to Emacs's real UID. This ignores the effective UID, and the environment variables `LOGNAME` and `USER`.

`user-full-name` **&optional** *uid* [Function]
: This function returns the full name of the logged-in user—or the value of the environment variable `NAME`, if that is set.

  If the Emacs process's user-id does not correspond to any known user (and provided `NAME` is not set), the result is `"unknown"`.

  If *uid* is non-`nil`, then it should be a number (a user-id) or a string (a login name). Then `user-full-name` returns the full name corresponding to that user-id or login name. If you specify a user-id or login name that isn't defined, it returns `nil`.

The symbols `user-login-name`, `user-real-login-name` and `user-full-name` are variables as well as functions. The functions return the same values that the variables hold. These variables allow you to "fake out" Emacs by telling the functions what to return. The variables are also useful for constructing frame titles (see Section 29.5 [Frame Titles], page 81).

`user-real-uid` [Function]
: This function returns the real UID of the user. The value may be a floating point number, in the (unlikely) event that the UID is too large to fit in a Lisp integer.

`user-uid` [Function]
: This function returns the effective UID of the user. The value may be a floating point number.

39.5 Time of Day

This section explains how to determine the current time and time zone.

Most of these functions represent time as a list of either three integers, `(`*`sec-high sec-low microsec`*`)`, or of two integers, `(`*`sec-high sec-low`*`)`. The integers *sec-high* and *sec-low* give the high and low bits of an integer number of seconds. This integer number, $high*2^{16}+low$, is the number of seconds from the *epoch* (0:00 January 1, 1970 UTC) to the specified time. The third list element *microsec*, if present, gives the number of microseconds from the start of that second to the specified time.

The return value of `current-time` represents time using three integers, while the timestamps in the return value of `file-attributes` use two integers (see [Definition of file-attributes], page 476, vol. 1). In function arguments, e.g. the *time-value* argument to `current-time-string`, both two- and three-integer lists are accepted. You can convert

times from the list representation into standard human-readable strings using `current-time`, or to other forms using the `decode-time` and `format-time-string` functions documented in the following sections.

`current-time-string` **&optional** *time-value* [Function]

This function returns the current time and date as a human-readable string. The format of the string is unvarying; the number of characters used for each part is always the same, so you can reliably use `substring` to extract pieces of it. You should count characters from the beginning of the string rather than from the end, as additional information may some day be added at the end.

The argument *time-value*, if given, specifies a time to format (represented as a list of integers), instead of the current time.

```
(current-time-string)
     ⇒ "Wed Oct 14 22:21:05 1987"
```

`current-time` [Function]

This function returns the current time, represented as a list of three integers `(`*sec-high sec-low microsec*`)`. On systems with only one-second time resolutions, *microsec* is 0.

`float-time` **&optional** *time-value* [Function]

This function returns the current time as a floating-point number of seconds since the epoch. The optional argument *time-value*, if given, specifies a time (represented as a list of integers) to convert instead of the current time.

Warning: Since the result is floating point, it may not be exact. Do not use this function if precise time stamps are required.

`current-time-zone` **&optional** *time-value* [Function]

This function returns a list describing the time zone that the user is in.

The value has the form `(`*offset name*`)`. Here *offset* is an integer giving the number of seconds ahead of UTC (east of Greenwich). A negative value means west of Greenwich. The second element, *name*, is a string giving the name of the time zone. Both elements change when daylight saving time begins or ends; if the user has specified a time zone that does not use a seasonal time adjustment, then the value is constant through time.

If the operating system doesn't supply all the information necessary to compute the value, the unknown elements of the list are `nil`.

The argument *time-value*, if given, specifies a time (represented as a list of integers) to analyze instead of the current time.

The current time zone is determined by the `TZ` environment variable. See Section 39.3 [System Environment], page 395. For example, you can tell Emacs to use universal time with `(setenv "TZ" "UTC0")`. If `TZ` is not in the environment, Emacs uses a platform-dependent default time zone.

39.6 Time Conversion

These functions convert time values (lists of two or three integers, as explained in the previous section) into calendrical information and vice versa.

Many 32-bit operating systems are limited to time values containing 32 bits of information; these systems typically handle only the times from 1901-12-13 20:45:52 UTC through 2038-01-19 03:14:07 UTC. However, 64-bit and some 32-bit operating systems have larger time values, and can represent times far in the past or future.

Time conversion functions always use the Gregorian calendar, even for dates before the Gregorian calendar was introduced. Year numbers count the number of years since the year 1 B.C., and do not skip zero as traditional Gregorian years do; for example, the year number −37 represents the Gregorian year 38 B.C.

`decode-time` **&optional** *time* [Function]

This function converts a time value into calendrical information. If you don't specify *time*, it decodes the current time. The return value is a list of nine elements, as follows:

```
(seconds minutes hour day month year dow dst zone)
```

Here is what the elements mean:

seconds The number of seconds past the minute, as an integer between 0 and 59. On some operating systems, this is 60 for leap seconds.

minutes The number of minutes past the hour, as an integer between 0 and 59.

hour The hour of the day, as an integer between 0 and 23.

day The day of the month, as an integer between 1 and 31.

month The month of the year, as an integer between 1 and 12.

year The year, an integer typically greater than 1900.

dow The day of week, as an integer between 0 and 6, where 0 stands for Sunday.

dst `t` if daylight saving time is effect, otherwise `nil`.

zone An integer indicating the time zone, as the number of seconds east of Greenwich.

Common Lisp Note: Common Lisp has different meanings for *dow* and *zone*.

`encode-time` *seconds minutes hour day month year* **&optional** *zone* [Function]

This function is the inverse of `decode-time`. It converts seven items of calendrical data into a time value. For the meanings of the arguments, see the table above under `decode-time`.

Year numbers less than 100 are not treated specially. If you want them to stand for years above 1900, or years above 2000, you must alter them yourself before you call `encode-time`.

The optional argument *zone* defaults to the current time zone and its daylight saving time rules. If specified, it can be either a list (as you would get from `current-time-zone`), a string as in the `TZ` environment variable, `t` for Universal Time, or an integer

(as you would get from `decode-time`). The specified zone is used without any further alteration for daylight saving time.

If you pass more than seven arguments to `encode-time`, the first six are used as *seconds* through *year*, the last argument is used as *zone*, and the arguments in between are ignored. This feature makes it possible to use the elements of a list returned by `decode-time` as the arguments to `encode-time`, like this:

```
(apply 'encode-time (decode-time ...))
```

You can perform simple date arithmetic by using out-of-range values for the *seconds*, *minutes*, *hour*, *day*, and *month* arguments; for example, day 0 means the day preceding the given month.

The operating system puts limits on the range of possible time values; if you try to encode a time that is out of range, an error results. For instance, years before 1970 do not work on some systems; on others, years as early as 1901 do work.

39.7 Parsing and Formatting Times

These functions convert time values (lists of two or three integers) to text in a string, and vice versa.

`date-to-time` *string* [Function]

This function parses the time-string *string* and returns the corresponding time value.

`format-time-string` *format-string* **&optional** *time universal* [Function]

This function converts *time* (or the current time, if *time* is omitted) to a string according to *format-string*. The argument *format-string* may contain '`%`'-sequences which say to substitute parts of the time. Here is a table of what the '`%`'-sequences mean:

'`%a`' This stands for the abbreviated name of the day of week.

'`%A`' This stands for the full name of the day of week.

'`%b`' This stands for the abbreviated name of the month.

'`%B`' This stands for the full name of the month.

'`%c`' This is a synonym for '`%x %X`'.

'`%C`' This has a locale-specific meaning. In the default locale (named C), it is equivalent to '`%A, %B %e, %Y`'.

'`%d`' This stands for the day of month, zero-padded.

'`%D`' This is a synonym for '`%m/%d/%y`'.

'`%e`' This stands for the day of month, blank-padded.

'`%h`' This is a synonym for '`%b`'.

'`%H`' This stands for the hour (00-23).

'`%I`' This stands for the hour (01-12).

'`%j`' This stands for the day of the year (001-366).

‘%k’ This stands for the hour (0-23), blank padded.

‘%l’ This stands for the hour (1-12), blank padded.

‘%m’ This stands for the month (01-12).

‘%M’ This stands for the minute (00-59).

‘%n’ This stands for a newline.

‘%N’ This stands for the nanoseconds (000000000-999999999). To ask for fewer digits, use ‘%3N’ for milliseconds, ‘%6N’ for microseconds, etc. Any excess digits are discarded, without rounding. Currently Emacs time stamps are at best microsecond resolution so the last three digits generated by plain ‘%N’ are always zero.

‘%p’ This stands for ‘AM’ or ‘PM’, as appropriate.

‘%r’ This is a synonym for ‘%I:%M:%S %p’.

‘%R’ This is a synonym for ‘%H:%M’.

‘%S’ This stands for the seconds (00-59).

‘%t’ This stands for a tab character.

‘%T’ This is a synonym for ‘%H:%M:%S’.

‘%U’ This stands for the week of the year (01-52), assuming that weeks start on Sunday.

‘%w’ This stands for the numeric day of week (0-6). Sunday is day 0.

‘%W’ This stands for the week of the year (01-52), assuming that weeks start on Monday.

‘%x’ This has a locale-specific meaning. In the default locale (named ‘C’), it is equivalent to ‘%D’.

‘%X’ This has a locale-specific meaning. In the default locale (named ‘C’), it is equivalent to ‘%T’.

‘%y’ This stands for the year without century (00-99).

‘%Y’ This stands for the year with century.

‘%Z’ This stands for the time zone abbreviation (e.g., ‘EST’).

‘%z’ This stands for the time zone numerical offset (e.g., ‘-0500’).

You can also specify the field width and type of padding for any of these ‘%’-sequences. This works as in `printf`: you write the field width as digits in the middle of a ‘%’-sequences. If you start the field width with ‘0’, it means to pad with zeros. If you start the field width with ‘_’, it means to pad with spaces.

For example, ‘%S’ specifies the number of seconds since the minute; ‘%03S’ means to pad this with zeros to 3 positions, ‘%_3S’ to pad with spaces to 3 positions. Plain ‘%3S’ pads with zeros, because that is how ‘%S’ normally pads to two positions.

The characters ‘E’ and ‘O’ act as modifiers when used between ‘%’ and one of the letters in the table above. ‘E’ specifies using the current locale’s “alternative” version

of the date and time. In a Japanese locale, for example, `%Ex` might yield a date format based on the Japanese Emperors' reigns. '`E`' is allowed in '`%Ec`', '`%EC`', '`%Ex`', '`%EX`', '`%Ey`', and '`%EY`'.

'`O`' means to use the current locale's "alternative" representation of numbers, instead of the ordinary decimal digits. This is allowed with most letters, all the ones that output numbers.

If *universal* is non-`nil`, that means to describe the time as Universal Time; `nil` means describe it using what Emacs believes is the local time zone (see `current-time-zone`).

This function uses the C library function `strftime` (see Section "Formatting Calendar Time" in *The GNU C Library Reference Manual*) to do most of the work. In order to communicate with that function, it first encodes its argument using the coding system specified by `locale-coding-system` (see Section 33.11 [Locales], page 207); after `strftime` returns the resulting string, `format-time-string` decodes the string using that same coding system.

`seconds-to-time` *seconds* [Function]

This function converts *seconds*, a floating point number of seconds since the epoch, to a time value and returns that. To perform the inverse conversion, use `float-time`.

`format-seconds` *format-string seconds* [Function]

This function converts its argument *seconds* into a string of years, days, hours, etc., according to *format-string*. The argument *format-string* may contain '`%`'-sequences which control the conversion. Here is a table of what the '`%`'-sequences mean:

'`%y`'
'`%Y`' The integer number of 365-day years.

'`%d`'
'`%D`' The integer number of days.

'`%h`'
'`%H`' The integer number of hours.

'`%m`'
'`%M`' The integer number of minutes.

'`%s`'
'`%S`' The integer number of seconds.

'`%z`' Non-printing control flag. When it is used, other specifiers must be given in the order of decreasing size, i.e. years before days, hours before minutes, etc. Nothing will be produced in the result string to the left of '`%z`' until the first non-zero conversion is encountered. For example, the default format used by `emacs-uptime` (see Section 39.8 [Processor Run Time], page 405) `"%Y, %D, %H, %M, %z%S"` means that the number of seconds will always be produced, but years, days, hours, and minutes will only be shown if they are non-zero.

'`%%`' Produces a literal '`%`'.

Upper-case format sequences produce the units in addition to the numbers, lower-case formats produce only the numbers.

You can also specify the field width by following the '`%`' with a number; shorter numbers will be padded with blanks. An optional period before the width requests zero-padding instead. For example, `"%.3Y"` might produce `"004 years"`.

Warning: This function works only with values of *seconds* that don't exceed `most-positive-fixnum` (see Section 3.1 [Integer Basics], page 33, vol. 1).

39.8 Processor Run time

Emacs provides several functions and primitives that return time, both elapsed and processor time, used by the Emacs process.

`emacs-uptime` **&optional** *format* [Command]

This function returns a string representing the Emacs *uptime*—the elapsed wall-clock time this instance of Emacs is running. The string is formatted by `format-seconds` according to the optional argument *format*. For the available format descriptors, see Section 39.7 [Time Parsing], page 402. If *format* is `nil` or omitted, it defaults to `"%Y, %D, %H, %M, %z%S"`.

When called interactively, it prints the uptime in the echo area.

`get-internal-run-time` [Function]

This function returns the processor run time used by Emacs as a list of three integers: `(high low microsec)`. The integers *high* and *low* combine to give the number of seconds, which is $high * 2^{16} + low$.

The third element, *microsec*, gives the microseconds (or 0 for systems that return time with the resolution of only one second).

Note that the time returned by this function excludes the time Emacs was not using the processor, and if the Emacs process has several threads, the returned value is the sum of the processor times used up by all Emacs threads.

If the system doesn't provide a way to determine the processor run time, `get-internal-run-time` returns the same time as `current-time`.

`emacs-init-time` [Command]

This function returns the duration of the Emacs initialization (see Section 39.1.1 [Startup Summary], page 386) in seconds, as a string. When called interactively, it prints the duration in the echo area.

39.9 Time Calculations

These functions perform calendrical computations using time values (the kind of list that `current-time` returns).

`time-less-p` *t1 t2* [Function]

This returns `t` if time value *t1* is less than time value *t2*.

`time-subtract` *t1 t2* [Function]

This returns the time difference $t1 - t2$ between two time values, in the same format as a time value.

`time-add` *t1 t2* [Function]
: This returns the sum of two time values, one of which ought to represent a time difference rather than a point in time. Here is how to add a number of seconds to a time value:

```
(time-add time (seconds-to-time seconds))
```

`time-to-days` *time* [Function]
: This function returns the number of days between the beginning of year 1 and *time*.

`time-to-day-in-year` *time* [Function]
: This returns the day number within the year corresponding to *time*.

`date-leap-year-p` *year* [Function]
: This function returns `t` if *year* is a leap year.

39.10 Timers for Delayed Execution

You can set up a *timer* to call a function at a specified future time or after a certain length of idleness.

Emacs cannot run timers at any arbitrary point in a Lisp program; it can run them only when Emacs could accept output from a subprocess: namely, while waiting or inside certain primitive functions such as `sit-for` or `read-event` which *can* wait. Therefore, a timer's execution may be delayed if Emacs is busy. However, the time of execution is very precise if Emacs is idle.

Emacs binds `inhibit-quit` to `t` before calling the timer function, because quitting out of many timer functions can leave things in an inconsistent state. This is normally unproblematical because most timer functions don't do a lot of work. Indeed, for a timer to call a function that takes substantial time to run is likely to be annoying. If a timer function needs to allow quitting, it should use `with-local-quit` (see Section 21.11 [Quitting], page 351, vol. 1). For example, if a timer function calls `accept-process-output` to receive output from an external process, that call should be wrapped inside `with-local-quit`, to ensure that *C-g* works if the external process hangs.

It is usually a bad idea for timer functions to alter buffer contents. When they do, they usually should call `undo-boundary` both before and after changing the buffer, to separate the timer's changes from user commands' changes and prevent a single undo entry from growing to be quite large.

Timer functions should also avoid calling functions that cause Emacs to wait, such as `sit-for` (see Section 21.10 [Waiting], page 350, vol. 1). This can lead to unpredictable effects, since other timers (or even the same timer) can run while waiting. If a timer function needs to perform an action after a certain time has elapsed, it can do this by scheduling a new timer.

If a timer function calls functions that can change the match data, it should save and restore the match data. See Section 34.6.4 [Saving Match Data], page 229.

`run-at-time` *time repeat function* **&rest** *args* [Command]
: This sets up a timer that calls the function *function* with arguments *args* at time *time*. If *repeat* is a number (integer or floating point), the timer is scheduled to run again every *repeat* seconds after *time*. If *repeat* is `nil`, the timer runs only once.

time may specify an absolute or a relative time.

Absolute times may be specified using a string with a limited variety of formats, and are taken to be times *today*, even if already in the past. The recognized forms are '*xxxx*', '*x*`:`*xx*', or '*xx*`:`*xx*' (military time), and '*xx*`am`', '*xx*`AM`', '*xx*`pm`', '*xx*`PM`', '*xx*`:`*xx*`am`', '*xx*`:`*xx*`AM`', '*xx*`:`*xx*`pm`', or '*xx*`:`*xx*`PM`'. A period can be used instead of a colon to separate the hour and minute parts.

To specify a relative time as a string, use numbers followed by units. For example:

'`1 min`' denotes 1 minute from now.

'`1 min 5 sec`'
denotes 65 seconds from now.

'`1 min 2 sec 3 hour 4 day 5 week 6 fortnight 7 month 8 year`'
denotes exactly 103 months, 123 days, and 10862 seconds from now.

For relative time values, Emacs considers a month to be exactly thirty days, and a year to be exactly 365.25 days.

Not all convenient formats are strings. If *time* is a number (integer or floating point), that specifies a relative time measured in seconds. The result of `encode-time` can also be used to specify an absolute value for *time*.

In most cases, *repeat* has no effect on when *first* call takes place—*time* alone specifies that. There is one exception: if *time* is `t`, then the timer runs whenever the time is a multiple of *repeat* seconds after the epoch. This is useful for functions like `display-time`.

The function `run-at-time` returns a timer value that identifies the particular scheduled future action. You can use this value to call `cancel-timer` (see below).

A repeating timer nominally ought to run every *repeat* seconds, but remember that any invocation of a timer can be late. Lateness of one repetition has no effect on the scheduled time of the next repetition. For instance, if Emacs is busy computing for long enough to cover three scheduled repetitions of the timer, and then starts to wait, it will immediately call the timer function three times in immediate succession (presuming no other timers trigger before or between them). If you want a timer to run again no less than *n* seconds after the last invocation, don't use the *repeat* argument. Instead, the timer function should explicitly reschedule the timer.

`timer-max-repeats` [Variable]
This variable's value specifies the maximum number of times to repeat calling a timer function in a row, when many previously scheduled calls were unavoidably delayed.

`with-timeout` (*seconds timeout-forms...*) *body...* [Macro]
Execute *body*, but give up after *seconds* seconds. If *body* finishes before the time is up, `with-timeout` returns the value of the last form in *body*. If, however, the execution of *body* is cut short by the timeout, then `with-timeout` executes all the *timeout-forms* and returns the value of the last of them.

This macro works by setting a timer to run after *seconds* seconds. If *body* finishes before that time, it cancels the timer. If the timer actually runs, it terminates execution of *body*, then executes *timeout-forms*.

Since timers can run within a Lisp program only when the program calls a primitive that can wait, `with-timeout` cannot stop executing *body* while it is in the midst of a computation—only when it calls one of those primitives. So use `with-timeout` only with a *body* that waits for input, not one that does a long computation.

The function `y-or-n-p-with-timeout` provides a simple way to use a timer to avoid waiting too long for an answer. See Section 20.7 [Yes-or-No Queries], page 307, vol. 1.

`cancel-timer` *timer* [Function]
This cancels the requested action for *timer*, which should be a timer—usually, one previously returned by `run-at-time` or `run-with-idle-timer`. This cancels the effect of that call to one of these functions; the arrival of the specified time will not cause anything special to happen.

39.11 Idle Timers

Here is how to set up a timer that runs when Emacs is idle for a certain length of time. Aside from how to set them up, idle timers work just like ordinary timers.

`run-with-idle-timer` *secs repeat function* **&rest** *args* [Command]
Set up a timer which runs the next time Emacs is idle for *secs* seconds. The value of *secs* may be an integer or a floating point number; a value of the type returned by `current-idle-time` is also allowed.

If *repeat* is `nil`, the timer runs just once, the first time Emacs remains idle for a long enough time. More often *repeat* is non-`nil`, which means to run the timer *each time* Emacs remains idle for *secs* seconds.

The function `run-with-idle-timer` returns a timer value which you can use in calling `cancel-timer` (see Section 39.10 [Timers], page 406).

Emacs becomes *idle* when it starts waiting for user input, and it remains idle until the user provides some input. If a timer is set for five seconds of idleness, it runs approximately five seconds after Emacs first becomes idle. Even if *repeat* is non-`nil`, this timer will not run again as long as Emacs remains idle, because the duration of idleness will continue to increase and will not go down to five seconds again.

Emacs can do various things while idle: garbage collect, autosave or handle data from a subprocess. But these interludes during idleness do not interfere with idle timers, because they do not reset the clock of idleness to zero. An idle timer set for 600 seconds will run when ten minutes have elapsed since the last user command was finished, even if subprocess output has been accepted thousands of times within those ten minutes, and even if there have been garbage collections and autosaves.

When the user supplies input, Emacs becomes non-idle while executing the input. Then it becomes idle again, and all the idle timers that are set up to repeat will subsequently run another time, one by one.

`current-idle-time` [Function]
If Emacs is idle, this function returns the length of time Emacs has been idle, as a list of three integers: `(`*`sec-high sec-low microsec`*`)`, where *high* and *low* are the

high and low bits for the number of seconds and *microsec* is the additional number of microseconds (see Section 39.5 [Time of Day], page 399).

When Emacs is not idle, `current-idle-time` returns `nil`. This is a convenient way to test whether Emacs is idle.

The main use of this function is when an idle timer function wants to "take a break" for a while. It can set up another idle timer to call the same function again, after a few seconds more idleness. Here's an example:

```
(defvar resume-timer nil
  "Timer that `timer-function' used to reschedule itself, or nil.")

(defun timer-function ()
  ;; If the user types a command while resume-timer
  ;; is active, the next time this function is called from
  ;; its main idle timer, deactivate resume-timer.
  (when resume-timer
    (cancel-timer resume-timer))
  ...do the work for a while...
  (when taking-a-break
    (setq resume-timer
          (run-with-idle-timer
            ;; Compute an idle time break-length
            ;; more than the current value.
            (time-add (current-idle-time)
                      (seconds-to-time break-length))
            nil
            'timer-function))))
```

Do not write an idle timer function containing a loop which does a certain amount of processing each time around, and exits when `(input-pending-p)` is non-`nil`. This approach seems very natural but has two problems:

- It blocks out all process output (since Emacs accepts process output only while waiting).
- It blocks out any idle timers that ought to run during that time.

The correct approach is for the idle timer to reschedule itself after a brief pause, using the method in the `timer-function` example above.

39.12 Terminal Input

This section describes functions and variables for recording or manipulating terminal input. See Chapter 38 [Display], page 299, for related functions.

39.12.1 Input Modes

`set-input-mode` *interrupt flow meta* **&optional** *quit-char* [Function]

This function sets the mode for reading keyboard input. If *interrupt* is non-null, then Emacs uses input interrupts. If it is `nil`, then it uses CBREAK mode. The default setting is system-dependent. Some systems always use CBREAK mode regardless of what is specified.

When Emacs communicates directly with X, it ignores this argument and uses interrupts if that is the way it knows how to communicate.

If *flow* is non-`nil`, then Emacs uses XON/XOFF (`C-q`, `C-s`) flow control for output to the terminal. This has no effect except in CBREAK mode.

The argument *meta* controls support for input character codes above 127. If *meta* is `t`, Emacs converts characters with the 8th bit set into Meta characters. If *meta* is `nil`, Emacs disregards the 8th bit; this is necessary when the terminal uses it as a parity bit. If *meta* is neither `t` nor `nil`, Emacs uses all 8 bits of input unchanged. This is good for terminals that use 8-bit character sets.

If *quit-char* is non-`nil`, it specifies the character to use for quitting. Normally this character is `C-g`. See Section 21.11 [Quitting], page 351, vol. 1.

The `current-input-mode` function returns the input mode settings Emacs is currently using.

`current-input-mode` [Function]

This function returns the current mode for reading keyboard input. It returns a list, corresponding to the arguments of `set-input-mode`, of the form `(`*`interrupt flow meta quit`*`)` in which:

interrupt is non-`nil` when Emacs is using interrupt-driven input. If `nil`, Emacs is using CBREAK mode.

flow is non-`nil` if Emacs uses XON/XOFF (`C-q`, `C-s`) flow control for output to the terminal. This value is meaningful only when *interrupt* is `nil`.

meta is `t` if Emacs treats the eighth bit of input characters as the meta bit; `nil` means Emacs clears the eighth bit of every input character; any other value means Emacs uses all eight bits as the basic character code.

quit is the character Emacs currently uses for quitting, usually `C-g`.

39.12.2 Recording Input

`recent-keys` [Function]

This function returns a vector containing the last 300 input events from the keyboard or mouse. All input events are included, whether or not they were used as parts of key sequences. Thus, you always get the last 100 input events, not counting events generated by keyboard macros. (These are excluded because they are less interesting for debugging; it should be enough to see the events that invoked the macros.)

A call to `clear-this-command-keys` (see Section 21.5 [Command Loop Info], page 324, vol. 1) causes this function to return an empty vector immediately afterward.

`open-dribble-file` *filename* [Command]

This function opens a *dribble file* named *filename*. When a dribble file is open, each input event from the keyboard or mouse (but not those from keyboard macros) is written in that file. A non-character event is expressed using its printed representation surrounded by '`<...>`'.

You close the dribble file by calling this function with an argument of `nil`.

This function is normally used to record the input necessary to trigger an Emacs bug, for the sake of a bug report.

```
(open-dribble-file "~/dribble")
     ⇒ nil
```

See also the `open-termscript` function (see Section 39.13 [Terminal Output], page 411).

39.13 Terminal Output

The terminal output functions send output to a text terminal, or keep track of output sent to the terminal. The variable `baud-rate` tells you what Emacs thinks is the output speed of the terminal.

`baud-rate` [User Option]

This variable's value is the output speed of the terminal, as far as Emacs knows. Setting this variable does not change the speed of actual data transmission, but the value is used for calculations such as padding.

It also affects decisions about whether to scroll part of the screen or repaint on text terminals. See Section 38.2 [Forcing Redisplay], page 299, for the corresponding functionality on graphical terminals.

The value is measured in baud.

If you are running across a network, and different parts of the network work at different baud rates, the value returned by Emacs may be different from the value used by your local terminal. Some network protocols communicate the local terminal speed to the remote machine, so that Emacs and other programs can get the proper value, but others do not. If Emacs has the wrong value, it makes decisions that are less than optimal. To fix the problem, set `baud-rate`.

`send-string-to-terminal` *string* **&optional** *terminal* [Function]

This function sends *string* to *terminal* without alteration. Control characters in *string* have terminal-dependent effects. This function operates only on text terminals. *terminal* may be a terminal object, a frame, or `nil` for the selected frame's terminal. In batch mode, *string* is sent to `stdout` when *terminal* is `nil`.

One use of this function is to define function keys on terminals that have downloadable function key definitions. For example, this is how (on certain terminals) to define function key 4 to move forward four characters (by transmitting the characters `C-u C-f` to the computer):

```
(send-string-to-terminal "\eF4\^U\^F")
     ⇒ nil
```

`open-termscript` *filename* [Command]

This function is used to open a *termscript file* that will record all the characters sent by Emacs to the terminal. It returns `nil`. Termscript files are useful for investigating problems where Emacs garbles the screen, problems that are due to incorrect Termcap entries or to undesirable settings of terminal options more often than to actual Emacs bugs. Once you are certain which characters were actually output, you can determine reliably whether they correspond to the Termcap specifications in use.

You close the termscript file by calling this function with an argument of `nil`.

See also `open-dribble-file` in Section 39.12.2 [Recording Input], page 410.

```
(open-termscript "../junk/termscript")
     ⇒ nil
```

39.14 Sound Output

To play sound using Emacs, use the function `play-sound`. Only certain systems are supported; if you call `play-sound` on a system which cannot really do the job, it gives an error.

The sound must be stored as a file in RIFF-WAVE format (‘`.wav`’) or Sun Audio format (‘`.au`’).

`play-sound` *sound* [Function]
: This function plays a specified sound. The argument, *sound*, has the form `(sound` *`properties...`*`)`, where the *properties* consist of alternating keywords (particular symbols recognized specially) and values corresponding to them.

  Here is a table of the keywords that are currently meaningful in *sound*, and their meanings:

  `:file` *`file`*
  : This specifies the file containing the sound to play. If the file name is not absolute, it is expanded against the directory `data-directory`.

  `:data` *`data`*
  : This specifies the sound to play without need to refer to a file. The value, *data*, should be a string containing the same bytes as a sound file. We recommend using a unibyte string.

  `:volume` *`volume`*
  : This specifies how loud to play the sound. It should be a number in the range of 0 to 1. The default is to use whatever volume has been specified before.

  `:device` *`device`*
  : This specifies the system device on which to play the sound, as a string. The default device is system-dependent.

  Before actually playing the sound, `play-sound` calls the functions in the list `play-sound-functions`. Each function is called with one argument, *sound*.

`play-sound-file` *file* **&optional** *volume device* [Command]
: This function is an alternative interface to playing a sound *file* specifying an optional *volume* and *device*.

`play-sound-functions` [Variable]
: A list of functions to be called before playing a sound. Each function is called with one argument, a property list that describes the sound.

39.15 Operating on X11 Keysyms

To define system-specific X11 keysyms, set the variable `system-key-alist`.

`system-key-alist` [Variable]

This variable's value should be an alist with one element for each system-specific keysym. Each element has the form `(code . symbol)`, where *code* is the numeric keysym code (not including the "vendor specific" bit, -2^{28}), and *symbol* is the name for the function key.

For example `(168 . mute-acute)` defines a system-specific key (used by HP X servers) whose numeric code is -2^{28} + 168.

It is not crucial to exclude from the alist the keysyms of other X servers; those do no harm, as long as they don't conflict with the ones used by the X server actually in use.

The variable is always local to the current terminal, and cannot be buffer-local. See Section 29.2 [Multiple Terminals], page 67.

You can specify which keysyms Emacs should use for the Meta, Alt, Hyper, and Super modifiers by setting these variables:

`x-alt-keysym` [Variable]
`x-meta-keysym` [Variable]
`x-hyper-keysym` [Variable]
`x-super-keysym` [Variable]

The name of the keysym that should stand for the Alt modifier (respectively, for Meta, Hyper, and Super). For example, here is how to swap the Meta and Alt modifiers within Emacs:

```
(setq x-alt-keysym 'meta)
(setq x-meta-keysym 'alt)
```

39.16 Batch Mode

The command-line option '`-batch`' causes Emacs to run noninteractively. In this mode, Emacs does not read commands from the terminal, it does not alter the terminal modes, and it does not expect to be outputting to an erasable screen. The idea is that you specify Lisp programs to run; when they are finished, Emacs should exit. The way to specify the programs to run is with '`-l` *file*', which loads the library named *file*, or '`-f` *function*', which calls *function* with no arguments, or '`--eval` *form*'.

Any Lisp program output that would normally go to the echo area, either using `message`, or using `prin1`, etc., with `t` as the stream, goes instead to Emacs's standard error descriptor when in batch mode. Similarly, input that would normally come from the minibuffer is read from the standard input descriptor. Thus, Emacs behaves much like a noninteractive application program. (The echo area output that Emacs itself normally generates, such as command echoing, is suppressed entirely.)

`noninteractive` [Variable]

This variable is non-`nil` when Emacs is running in batch mode.

39.17 Session Management

Emacs supports the X Session Management Protocol, which is used to suspend and restart applications. In the X Window System, a program called the *session manager* is responsible for keeping track of the applications that are running. When the X server shuts down, the session manager asks applications to save their state, and delays the actual shutdown until they respond. An application can also cancel the shutdown.

When the session manager restarts a suspended session, it directs these applications to individually reload their saved state. It does this by specifying a special command-line argument that says what saved session to restore. For Emacs, this argument is '`--smid` *`session`*'.

`emacs-save-session-functions` [Variable]

Emacs supports saving state via a hook called `emacs-save-session-functions`. Emacs runs this hook when the session manager tells it that the window system is shutting down. The functions are called with no arguments, and with the current buffer set to a temporary buffer. Each function can use `insert` to add Lisp code to this buffer. At the end, Emacs saves the buffer in a file, called the *session file*.

Subsequently, when the session manager restarts Emacs, it loads the session file automatically (see Chapter 15 [Loading], page 209, vol. 1). This is performed by a function named `emacs-session-restore`, which is called during startup. See Section 39.1.1 [Startup Summary], page 386.

If a function in `emacs-save-session-functions` returns non-`nil`, Emacs tells the session manager to cancel the shutdown.

Here is an example that just inserts some text into '`*scratch*`' when Emacs is restarted by the session manager.

```
(add-hook 'emacs-save-session-functions 'save-yourself-test)

(defun save-yourself-test ()
  (insert "(save-current-buffer
  (switch-to-buffer \"*scratch*\")
  (insert \"I am restored\"))")
  nil)
```

39.18 Desktop Notifications

Emacs is able to send *notifications* on systems that support the freedesktop.org Desktop Notifications Specification. In order to use this functionality, Emacs must have been compiled with D-Bus support, and the `notifications` library must be loaded.

`notifications-notify` **&rest** *params* [Function]

This function sends a notification to the desktop via D-Bus, consisting of the parameters specified by the *params* arguments. These arguments should consist of alternating keyword and value pairs. The supported keywords and values are as follows:

`:title` *`title`*

The notification title.

`:body` *text*
: The notification body text. Depending on the implementation of the notification server, the text could contain HTML markups, like '`"<b>bold text</b>"`', or hyperlinks.

`:app-name` *name*
: The name of the application sending the notification. The default is `notifications-application-name`.

`:replaces-id` *id*
: The notification *id* that this notification replaces. *id* must be the result of a previous `notifications-notify` call.

`:app-icon` *icon-file*
: The file name of the notification icon. If set to `nil`, no icon is displayed. The default is `notifications-application-icon`.

`:actions (`*key title key title* `...)`
: A list of actions to be applied. *key* and *title* are both strings. The default action (usually invoked by clicking the notification) should have a key named '`"default"`'. The title can be anything, though implementations are free not to display it.

`:timeout` *timeout*
: The timeout time in milliseconds since the display of the notification at which the notification should automatically close. If -1, the notification's expiration time is dependent on the notification server's settings, and may vary for the type of notification. If 0, the notification never expires. Default value is -1.

`:urgency` *urgency*
: The urgency level. It can be `low`, `normal`, or `critical`.

`:category` *category*
: The type of notification this is, a string.

`:desktop-entry` *filename*
: This specifies the name of the desktop filename representing the calling program, like '`"emacs"`'.

`:image-data (`*width height rowstride has-alpha bits channels data*`)`
: This is a raw data image format that describes the width, height, rowstride, whether there is an alpha channel, bits per sample, channels and image data, respectively.

`:image-path` *path*
: This is represented either as a URI ('`file://`' is the only URI schema supported right now) or a name in a freedesktop.org-compliant icon theme from '`$XDG_DATA_DIRS/icons`'.

`:sound-file` *filename*
: The path to a sound file to play when the notification pops up.

`:sound-name` *name*
: A themable named sound from the freedesktop.org sound naming specification from '`$XDG_DATA_DIRS/sounds`', to play when the notification pops up. Similar to the icon name, only for sounds. An example would be '`"message-new-instant"`'.

`:suppress-sound`
: Causes the server to suppress playing any sounds, if it has that ability.

`:x` *position*
`:y` *position*
: Specifies the X, Y location on the screen that the notification should point to. Both arguments must be used together.

`:on-action` *function*
: Function to call when an action is invoked. The notification *id* and the *key* of the action are passed as arguments to the function.

`:on-close` *function*
: Function to call when the notification has been closed by timeout or by the user. The function receive the notification *id* and the closing *reason* as arguments:

  - `expired` if the notification has expired
  - `dismissed` if the notification was dismissed by the user
  - `close-notification` if the notification was closed by a call to `notifications-close-notification`
  - `undefined` if the notification server hasn't provided a reason

This function returns a notification id, an integer, which can be used to manipulate the notification item with `notifications-close-notification` or the `:replaces-id` argument of another `notifications-notify` call. For example:

```
(defun my-on-action-function (id key)
  (message "Message %d, key \"%s\" pressed" id key))
     ⇒ my-on-action-function

(defun my-on-close-function (id reason)
  (message "Message %d, closed due to \"%s\"" id reason))
     ⇒ my-on-close-function

(notifications-notify
 :title "Title"
 :body "This is <b>important</b>."
 :actions '("Confirm" "I agree" "Refuse" "I disagree")
 :on-action 'my-on-action-function
 :on-close 'my-on-close-function)
     ⇒ 22
```

```
A message window opens on the desktop.  Press "I agree"
     ⇒ Message 22, key "Confirm" pressed
       Message 22, closed due to "dismissed"
```

`notifications-close-notification` *id* [Function]

This function closes a notification with identifier *id*.

39.19 Dynamically Loaded Libraries

A *dynamically loaded library* is a library that is loaded on demand, when its facilities are first needed. Emacs supports such on-demand loading of support libraries for some of its features.

`dynamic-library-alist` [Variable]

This is an alist of dynamic libraries and external library files implementing them.

Each element is a list of the form (*`library files...`*), where the `car` is a symbol representing a supported external library, and the rest are strings giving alternate filenames for that library.

Emacs tries to load the library from the files in the order they appear in the list; if none is found, the Emacs session won't have access to that library, and the features it provides will be unavailable.

Image support on some platforms uses this facility. Here's an example of setting this variable for supporting images on MS-Windows:

```
(setq dynamic-library-alist
      '((xpm "libxpm.dll" "xpm4.dll" "libXpm-nox4.dll")
        (png "libpng12d.dll" "libpng12.dll" "libpng.dll"
             "libpng13d.dll" "libpng13.dll")
        (jpeg "jpeg62.dll" "libjpeg.dll" "jpeg-62.dll"
              "jpeg.dll")
        (tiff "libtiff3.dll" "libtiff.dll")
        (gif "giflib4.dll" "libungif4.dll" "libungif.dll")
        (svg "librsvg-2-2.dll")
        (gdk-pixbuf "libgdk_pixbuf-2.0-0.dll")
        (glib "libglib-2.0-0.dll")
(gobject "libgobject-2.0-0.dll")))
```

Note that image types `pbm` and `xbm` do not need entries in this variable because they do not depend on external libraries and are always available in Emacs.

Also note that this variable is not meant to be a generic facility for accessing external libraries; only those already known by Emacs can be loaded through it.

This variable is ignored if the given *library* is statically linked into Emacs.

40 Preparing Lisp code for distribution

Emacs provides a standard way to distribute Emacs Lisp code to users. A *package* is a collection of one or more files, formatted and bundled in such a way that users can easily download, install, uninstall, and upgrade it.

The following sections describe how to create a package, and how to put it in a *package archive* for others to download. See Section "Packages" in *The GNU Emacs Manual*, for a description of user-level features of the packaging system.

40.1 Packaging Basics

A package is either a *simple package* or a *multi-file package*. A simple package is stored in a package archive as a single Emacs Lisp file, while a multi-file package is stored as a tar file (containing multiple Lisp files, and possibly non-Lisp files such as a manual).

In ordinary usage, the difference between simple packages and multi-file packages is relatively unimportant; the Package Menu interface makes no distinction between them. However, the procedure for creating them differs, as explained in the following sections.

Each package (whether simple or multi-file) has certain *attributes*:

Name
: A short word (e.g. '`auctex`'). This is usually also the symbol prefix used in the program (see Section D.1 [Coding Conventions], page 444).

Version
: A version number, in a form that the function `version-to-list` understands (e.g. '`11.86`'). Each release of a package should be accompanied by an increase in the version number.

Brief description
: This is shown when the package is listed in the Package Menu. It should occupy a single line, ideally in 36 characters or less.

Long description
: This is shown in the buffer created by `C-h P` (`describe-package`), following the package's brief description and installation status. It normally spans multiple lines, and should fully describe the package's capabilities and how to begin using it once it is installed.

Dependencies
: A list of other packages (possibly including minimal acceptable version numbers) on which this package depends. The list may be empty, meaning this package has no dependencies. Otherwise, installing this package also automatically installs its dependencies; if any dependency cannot be found, the package cannot be installed.

Installing a package, either via the command `package-install-file`, or via the Package Menu, creates a subdirectory of `package-user-dir` named '*`name-version`*', where *name* is the package's name and *version* its version (e.g. '`~/.emacs.d/elpa/auctex-11.86/`'). We call this the package's *content directory*. It is where Emacs puts the package's contents (the single Lisp file for a simple package, or the files extracted from a multi-file package).

Emacs then searches every Lisp file in the content directory for autoload magic comments (see Section 15.5 [Autoload], page 213, vol. 1). These autoload definitions are saved to a file

named '*name*`-autoloads.el`' in the content directory. They are typically used to autoload the principal user commands defined in the package, but they can also perform other tasks, such as adding an element to `auto-mode-alist` (see Section 23.2.2 [Auto Major Mode], page 403, vol. 1). Note that a package typically does *not* autoload every function and variable defined within it—only the handful of commands typically called to begin using the package. Emacs then byte-compiles every Lisp file in the package.

After installation, the installed package is *loaded*: Emacs adds the package's content directory to `load-path`, and evaluates the autoload definitions in '*name*`-autoloads.el`'.

Whenever Emacs starts up, it automatically calls the function `package-initialize` to load installed packages. This is done after loading the init file and abbrev file (if any) and before running `after-init-hook` (see Section 39.1.1 [Startup Summary], page 386). Automatic package loading is disabled if the user option `package-enable-at-startup` is `nil`.

`package-initialize` **&optional** *no-activate* [Command]

This function initializes Emacs' internal record of which packages are installed, and loads them. The user option `package-load-list` specifies which packages to load; by default, all installed packages are loaded. See Section "Package Installation" in *The GNU Emacs Manual*.

The optional argument *no-activate*, if non-`nil`, causes Emacs to update its record of installed packages without actually loading them; it is for internal use only.

40.2 Simple Packages

A simple package consists of a single Emacs Lisp source file. The file must conform to the Emacs Lisp library header conventions (see Section D.8 [Library Headers], page 454). The package's attributes are taken from the various headers, as illustrated by the following example:

```
;;; superfrobnicator.el --- Frobnicate and bifurcate flanges

;; Copyright (C) 2011 Free Software Foundation, Inc.

;; Author: J. R. Hacker <jrh@example.com>
;; Version: 1.3
;; Package-Requires: ((flange "1.0"))
;; Keywords: frobnicate

...

;;; Commentary:

;; This package provides a minor mode to frobnicate and/or
;; bifurcate any flanges you desire.  To activate it, just type
...

;;;###autoload
(define-minor-mode superfrobnicator-mode
```

```
...
```

The name of the package is the same as the base name of the file, as written on the first line. Here, it is '`superfrobnicator`'.

The brief description is also taken from the first line. Here, it is '`Frobnicate and bifurcate flanges`'.

The version number comes from the '`Package-Version`' header, if it exists, or from the '`Version`' header otherwise. One or the other *must* be present. Here, the version number is 1.3.

If the file has a '`;;; Commentary:`' section, this section is used as the long description. (When displaying the description, Emacs omits the '`;;; Commentary:`' line, as well as the leading comment characters in the commentary itself.)

If the file has a '`Package-Requires`' header, that is used as the package dependencies. In the above example, the package depends on the '`flange`' package, version 1.0 or higher. See Section D.8 [Library Headers], page 454, for a description of the '`Package-Requires`' header. If the header is omitted, the package has no dependencies.

The file ought to also contain one or more autoload magic comments, as explained in Section 40.1 [Packaging Basics], page 418. In the above example, a magic comment autoloads `superfrobnicator-mode`.

See Section 40.4 [Package Archives], page 421, for a explanation of how to add a single-file package to a package archive.

40.3 Multi-file Packages

A multi-file package is less convenient to create than a single-file package, but it offers more features: it can include multiple Emacs Lisp files, an Info manual, and other file types (such as images).

Prior to installation, a multi-file package is stored in a package archive as a tar file. The tar file must be named '*`name`*`-`*`version`*`.tar`', where *name* is the package name and *version* is the version number. Its contents, once extracted, must all appear in a directory named '*`name`*`-`*`version`*', the *content directory* (see Section 40.1 [Packaging Basics], page 418). Files may also extract into subdirectories of the content directory.

One of the files in the content directory must be named '*`name`*`-pkg.el`'. It must contain a single Lisp form, consisting of a call to the function `define-package`, described below. This defines the package's version, brief description, and requirements.

For example, if we distribute version 1.3 of the superfrobnicator as a multi-file package, the tar file would be '`superfrobnicator-1.3.tar`'. Its contents would extract into the directory '`superfrobnicator-1.3`', and one of these would be the file '`superfrobnicator-pkg.el`'.

`define-package` *name version* **&optional** *docstring requirements* [Function]

This function defines a package. *name* is the package name, a string. *version* is the version, as a string of a form that can be understood by the function `version-to-list`. *docstring* is the brief description.

requirements is a list of required packages and their versions. Each element in this list should have the form `(`*`dep-name dep-version`*`)`, where *dep-name* is a symbol whose

name is the dependency's package name, and *dep-version* is the dependency's version (a string).

If the content directory contains a file named '`README`', this file is used as the long description.

If the content directory contains a file named '`dir`', this is assumed to be an Info directory file made with `install-info`. See Section "Invoking install-info" in *Texinfo*. The relevant Info files should also be present in the content directory. In this case, Emacs will automatically add the content directory to `Info-directory-list` when the package is activated.

Do not include any '`.elc`' files in the package. Those are created when the package is installed. Note that there is no way to control the order in which files are byte-compiled.

Do not include any file named '*name*`-autoloads.el`'. This file is reserved for the package's autoload definitions (see Section 40.1 [Packaging Basics], page 418). It is created automatically when the package is installed, by searching all the Lisp files in the package for autoload magic comments.

If the multi-file package contains auxiliary data files (such as images), the package's Lisp code can refer to these files via the variable `load-file-name` (see Chapter 15 [Loading], page 209, vol. 1). Here is an example:

```
(defconst superfrobnicator-base (file-name-directory load-file-name))

(defun superfrobnicator-fetch-image (file)
  (expand-file-name file superfrobnicator-base))
```

40.4 Creating and Maintaining Package Archives

Via the Package Menu, users may download packages from *package archives*. Such archives are specified by the variable `package-archives`, whose default value contains a single entry: the archive hosted by the GNU project at `elpa.gnu.org`. This section describes how to set up and maintain a package archive.

`package-archives` [User Option]

The value of this variable is an alist of package archives recognized by the Emacs package manager.

Each alist element corresponds to one archive, and should have the form `(`*`id`*` . `*`location`*`)`, where *id* is the name of the archive (a string) and *location* is its *base location* (a string).

If the base location starts with '`http:`', it is treated as a HTTP URL, and packages are downloaded from this archive via HTTP (as is the case for the default GNU archive).

Otherwise, the base location should be a directory name. In this case, Emacs retrieves packages from this archive via ordinary file access. Such "local" archives are mainly useful for testing.

A package archive is simply a directory in which the package files, and associated files, are stored. If you want the archive to be reachable via HTTP, this directory must be accessible to a web server. How to accomplish this is beyond the scope of this manual.

A convenient way to set up and update a package archive is via the `package-x` library. This is included with Emacs, but not loaded by default; type *`M-x load-library RET package-x RET`* to load it, or add `(require 'package-x)` to your init file. See Section "Lisp Libraries" in *The GNU Emacs Manual*. Once loaded, you can make use of the following:

`package-archive-upload-base` [User Option]

The value of this variable is the base location of a package archive, as a directory name. The commands in the `package-x` library will use this base location.

The directory name should be absolute. You may specify a remote name, such as '`/ssh:foo@example.com:/var/www/packages/`', if the package archive is on a different machine. See Section "Remote Files" in *The GNU Emacs Manual*.

`package-upload-file` *filename* [Command]

This command prompts for *filename*, a file name, and uploads that file to `package-archive-upload-base`. The file must be either a simple package (a '`.el`' file) or a multi-file package (a '`.tar`' file); otherwise, an error is raised. The package attributes are automatically extracted, and the archive's contents list is updated with this information.

If `package-archive-upload-base` does not specify a valid directory, the function prompts interactively for one. If the directory does not exist, it is created. The directory need not have any initial contents (i.e., you can use this command to populate an initially empty archive).

`package-upload-buffer` [Command]

This command is similar to `package-upload-file`, but instead of prompting for a package file, it uploads the contents of the current buffer. The current buffer must be visiting a simple package (a '`.el`' file) or a multi-file package (a '`.tar`' file); otherwise, an error is raised.

After you create an archive, remember that it is not accessible in the Package Menu interface unless it is in `package-archives`.

Appendix A Emacs 23 Antinews

For those users who live backwards in time, here is information about downgrading to Emacs version 23.4. We hope you will enjoy the greater simplicity that results from the absence of many Emacs 24.1 features.

A.1 Old Lisp Features in Emacs 23

- Support for lexical scoping has been removed; all variables are dynamically scoped. The `lexical-binding` variable has been removed, and so has the *lexical* argument to `eval`. The `defvar` and `defconst` forms no longer mark variables as dynamic, since all variables are dynamic.

  Having only dynamic binding follows the spirit of Emacs extensibility, for it allows any Emacs code to access any defined variable with a minimum of fuss. But See Section 11.9.2 [Dynamic Binding Tips], page 147, vol. 1, for tips to avoid making your programs hard to understand.
- Calling a minor mode function from Lisp with a nil or omitted argument does not enable the minor mode unconditionally; instead, it toggles the minor mode—which is the straightforward thing to do, since that is the behavior when invoked interactively. One downside is that it is more troublesome to enable minor modes from hooks; you have to do something like

  ```
  (add-hook 'foo-hook (lambda () (bar-mode 1)))
  ```

  or define `turn-on-bar-mode` and call that from the hook.
- The `prog-mode` dummy major mode has been removed. Instead of using it as a crutch to meet programming mode conventions, you should explicitly ensure that your mode follows those conventions. See Section 23.2.1 [Major Mode Conventions], page 399, vol. 1.
- Emacs no longer supports bidirectional display and editing. Since there is no need to worry about the insertion of right-to-left text messing up how lines and paragraphs are displayed, the function `bidi-string-mark-left-to-right` has been removed; so have many other functions and variables related to bidirectional display. Unicode directionality characters like `U+200E` ("left-to-right mark") have no special effect on display.
- Emacs windows now have most of their internal state hidden from Lisp. Internal windows are no longer visible to Lisp; functions such as `window-parent`, window parameters related to window arrangement, and window-local buffer lists have all been removed. Functions for resizing windows can delete windows if they become too small.

  The "action function" feature for controlling buffer display has been removed, including `display-buffer-overriding-action` and related variables, as well as the *action* argument to `display-buffer` and other functions. The way to programmatically control how Emacs chooses a window to display a buffer is to bind the right combination of `special-display-regexps`, `pop-up-frames`, and other variables.
- The standard completion interface has been simplified, eliminating the `completion-extra-properties` variable, the `metadata` action flag for completion functions, and the concept of "completion categories". Lisp programmers may now find the choice of

methods for tuning completion less bewildering, but if a package finds the streamlined interface insufficient for its needs, it must implement its own specialized completion feature.

- `copy-directory` now behaves the same whether or not the destination is an existing directory: if the destination exists, the *contents* of the first directory are copied into it (with subdirectories handled recursively), rather than copying the first directory into a subdirectory.
- The *trash* arguments for `delete-file` and `delete-directory` have been removed. The variable `delete-by-moving-to-trash` must now be used with care; whenever it is non-`nil`, all calls to `delete-file` or `delete-directory` use the trash.
- Because Emacs no longer supports SELinux file contexts, the *preserve-selinux-context* argument to `copy-file` has been removed. The return value of `backup-buffer` no longer has an entry for the SELinux file context.
- For mouse click input events in the text area, the Y pixel coordinate in the *position* list (see Section 21.7.4 [Click Events], page 329, vol. 1) now counts from the top of the header line, if there is one, rather than the top of the text area.
- Bindings in menu keymaps (see Section 22.3 [Format of Keymaps], page 361, vol. 1) now sometimes get an additional *cache* entry in their definitions, like this:

  ```
  (type item-name cache . binding)
  ```

  The *cache* entry is used internally by Emacs to record equivalent keyboard key sequences for invoking the same command; Lisp programs should never use it.
- The `gnutls` library has been removed, and the function `open-network-stream` correspondingly simplified. Lisp programs that want an encrypted network connection must now call external utilities such as `starttls` or `gnutls-cli`.
- Tool bars can no longer display separators, which frees up several pixels of space on each graphical frame.
- As part of the ongoing quest for simplicity, many other functions and variables have been eliminated.

Appendix B GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix C GNU General Public License

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.

   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source.

The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or

e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance.

However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so

available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see http://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see http://www.gnu.org/licenses/.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read http://www.gnu.org/philosophy/why-not-lgpl.html.

Appendix D Tips and Conventions

This chapter describes no additional features of Emacs Lisp. Instead it gives advice on making effective use of the features described in the previous chapters, and describes conventions Emacs Lisp programmers should follow.

You can automatically check some of the conventions described below by running the command *M-x checkdoc RET* when visiting a Lisp file. It cannot check all of the conventions, and not all the warnings it gives necessarily correspond to problems, but it is worth examining them all.

D.1 Emacs Lisp Coding Conventions

Here are conventions that you should follow when writing Emacs Lisp code intended for widespread use:

- Simply loading a package should not change Emacs's editing behavior. Include a command or commands to enable and disable the feature, or to invoke it.

  This convention is mandatory for any file that includes custom definitions. If fixing such a file to follow this convention requires an incompatible change, go ahead and make the incompatible change; don't postpone it.

- You should choose a short word to distinguish your program from other Lisp programs. The names of all global variables, constants, and functions in your program should begin with that chosen prefix. Separate the prefix from the rest of the name with a hyphen, '`-`'. This practice helps avoid name conflicts, since all global variables in Emacs Lisp share the same name space, and all functions share another name space[1].

  Occasionally, for a command name intended for users to use, it is more convenient if some words come before the package's name prefix. And constructs that define functions, variables, etc., work better if they start with '`defun`' or '`defvar`', so put the name prefix later on in the name.

  This recommendation applies even to names for traditional Lisp primitives that are not primitives in Emacs Lisp—such as `copy-list`. Believe it or not, there is more than one plausible way to define `copy-list`. Play it safe; append your name prefix to produce a name like `foo-copy-list` or `mylib-copy-list` instead.

  If you write a function that you think ought to be added to Emacs under a certain name, such as `twiddle-files`, don't call it by that name in your program. Call it `mylib-twiddle-files` in your program, and send mail to '`bug-gnu-emacs@gnu.org`' suggesting we add it to Emacs. If and when we do, we can change the name easily enough.

  If one prefix is insufficient, your package can use two or three alternative common prefixes, so long as they make sense.

- Put a call to `provide` at the end of each separate Lisp file. See Section 15.7 [Named Features], page 217, vol. 1.

- If a file requires certain other Lisp programs to be loaded beforehand, then the comments at the beginning of the file should say so. Also, use `require` to make sure they are loaded. See Section 15.7 [Named Features], page 217, vol. 1.

[1] The benefits of a Common Lisp-style package system are considered not to outweigh the costs.

- If a file *foo* uses a macro defined in another file *bar*, but does not use any functions or variables defined in *bar*, then *foo* should contain the following expression:

  ```
  (eval-when-compile (require 'bar))
  ```

  This tells Emacs to load *bar* just before byte-compiling *foo*, so that the macro definition is available during compilation. Using `eval-when-compile` avoids loading *bar* when the compiled version of *foo* is *used*. It should be called before the first use of the macro in the file. See Section 13.3 [Compiling Macros], page 182, vol. 1.
- Avoid loading additional libraries at run time unless they are really needed. If your file simply cannot work without some other library, then just `require` that library at the top-level and be done with it. But if your file contains several independent features, and only one or two require the extra library, then consider putting `require` statements inside the relevant functions rather than at the top-level. Or use `autoload` statements to load the extra library when needed. This way people who don't use those aspects of your file do not need to load the extra library.
- Please don't require the `cl` package of Common Lisp extensions at run time. Use of this package is optional, and it is not part of the standard Emacs namespace. If your package loads `cl` at run time, that could cause name clashes for users who don't use that package.

  However, there is no problem with using the `cl` package at compile time, with `(eval-when-compile (require 'cl))`. That's sufficient for using the macros in the `cl` package, because the compiler expands them before generating the byte-code.
- When defining a major mode, please follow the major mode conventions. See Section 23.2.1 [Major Mode Conventions], page 399, vol. 1.
- When defining a minor mode, please follow the minor mode conventions. See Section 23.3.1 [Minor Mode Conventions], page 414, vol. 1.
- If the purpose of a function is to tell you whether a certain condition is true or false, give the function a name that ends in '`p`' (which stands for "predicate"). If the name is one word, add just '`p`'; if the name is multiple words, add '`-p`'. Examples are `framep` and `frame-live-p`.
- If the purpose of a variable is to store a single function, give it a name that ends in '`-function`'. If the purpose of a variable is to store a list of functions (i.e., the variable is a hook), please follow the naming conventions for hooks. See Section 23.1 [Hooks], page 396, vol. 1.
- If loading the file adds functions to hooks, define a function *feature*`-unload-hook`, where *feature* is the name of the feature the package provides, and make it undo any such changes. Using `unload-feature` to unload the file will run this function. See Section 15.9 [Unloading], page 220, vol. 1.
- It is a bad idea to define aliases for the Emacs primitives. Normally you should use the standard names instead. The case where an alias may be useful is where it facilitates backwards compatibility or portability.
- If a package needs to define an alias or a new function for compatibility with some other version of Emacs, name it with the package prefix, not with the raw name with which it occurs in the other version. Here is an example from Gnus, which provides many examples of such compatibility issues.

```
(defalias 'gnus-point-at-bol
  (if (fboundp 'point-at-bol)
      'point-at-bol
    'line-beginning-position))
```

- Redefining or advising an Emacs primitive is a bad idea. It may do the right thing for a particular program, but there is no telling what other programs might break as a result.
- It is likewise a bad idea for one Lisp package to advise a function in another Lisp package (see Chapter 17 [Advising Functions], page 233, vol. 1).
- Avoid using `eval-after-load` in libraries and packages (see Section 15.10 [Hooks for Loading], page 221, vol. 1). This feature is meant for personal customizations; using it in a Lisp program is unclean, because it modifies the behavior of another Lisp file in a way that's not visible in that file. This is an obstacle for debugging, much like advising a function in the other package.
- If a file does replace any of the standard functions or library programs of Emacs, prominent comments at the beginning of the file should say which functions are replaced, and how the behavior of the replacements differs from that of the originals.
- Constructs that define a function or variable should be macros, not functions, and their names should start with '`define-`'. The macro should receive the name to be defined as the first argument. That will help various tools find the definition automatically. Avoid constructing the names in the macro itself, since that would confuse these tools.
- In some other systems there is a convention of choosing variable names that begin and end with '`*`'. We don't use that convention in Emacs Lisp, so please don't use it in your programs. (Emacs uses such names only for special-purpose buffers.) People will find Emacs more coherent if all libraries use the same conventions.
- If your program contains non-ASCII characters in string or character constants, you should make sure Emacs always decodes these characters the same way, regardless of the user's settings. The easiest way to do this is to use the coding system `utf-8-emacs` (see Section 33.9.1 [Coding System Basics], page 193), and specify that coding in the '`-*-`' line or the local variables list. See Section "Local Variables in Files" in *The GNU Emacs Manual*.

  ```
  ;; XXX.el  -*- coding: utf-8-emacs; -*-
  ```

- Indent the file using the default indentation parameters.
- Don't make a habit of putting close-parentheses on lines by themselves; Lisp programmers find this disconcerting.
- Please put a copyright notice and copying permission notice on the file if you distribute copies. See Section D.8 [Library Headers], page 454.

D.2 Key Binding Conventions

- Many special major modes, like Dired, Info, Compilation, and Occur, are designed to handle read-only text that contains *hyper-links*. Such a major mode should redefine *`mouse-2`* and `RET` to follow the links. It should also set up a `follow-link` condition, so that the link obeys `mouse-1-click-follows-link`. See Section 32.19.8 [Clickable

Text], page 169. See Section 38.17 [Buttons], page 366, for an easy method of implementing such clickable links.

- Don't define `C-c letter` as a key in Lisp programs. Sequences consisting of `C-c` and a letter (either upper or lower case) are reserved for users; they are the **only** sequences reserved for users, so do not block them.

  Changing all the Emacs major modes to respect this convention was a lot of work; abandoning this convention would make that work go to waste, and inconvenience users. Please comply with it.
- Function keys `F5` through `F9` without modifier keys are also reserved for users to define.
- Sequences consisting of `C-c` followed by a control character or a digit are reserved for major modes.
- Sequences consisting of `C-c` followed by `{`, `}`, `<`, `>`, `:` or `;` are also reserved for major modes.
- Sequences consisting of `C-c` followed by any other punctuation character are allocated for minor modes. Using them in a major mode is not absolutely prohibited, but if you do that, the major mode binding may be shadowed from time to time by minor modes.
- Don't bind `C-h` following any prefix character (including `C-c`). If you don't bind `C-h`, it is automatically available as a help character for listing the subcommands of the prefix character.
- Don't bind a key sequence ending in `ESC` except following another `ESC`. (That is, it is OK to bind a sequence ending in *`ESC ESC`*.)

  The reason for this rule is that a non-prefix binding for `ESC` in any context prevents recognition of escape sequences as function keys in that context.
- Similarly, don't bind a key sequence ending in `C-g`, since that is commonly used to cancel a key sequence.
- Anything that acts like a temporary mode or state that the user can enter and leave should define *`ESC ESC`* or *`ESC ESC ESC`* as a way to escape.

  For a state that accepts ordinary Emacs commands, or more generally any kind of state in which `ESC` followed by a function key or arrow key is potentially meaningful, then you must not define *`ESC ESC`*, since that would preclude recognizing an escape sequence after `ESC`. In these states, you should define *`ESC ESC ESC`* as the way to escape. Otherwise, define *`ESC ESC`* instead.

D.3 Emacs Programming Tips

Following these conventions will make your program fit better into Emacs when it runs.

- Don't use `next-line` or `previous-line` in programs; nearly always, `forward-line` is more convenient as well as more predictable and robust. See Section 30.2.4 [Text Lines], page 102.
- Don't call functions that set the mark, unless setting the mark is one of the intended features of your program. The mark is a user-level feature, so it is incorrect to change the mark except to supply a value for the user's benefit. See Section 31.7 [The Mark], page 117.

  In particular, don't use any of these functions:

- `beginning-of-buffer`, `end-of-buffer`
- `replace-string`, `replace-regexp`
- `insert-file`, `insert-buffer`

If you just want to move point, or replace a certain string, or insert a file or buffer's contents, without any of the other features intended for interactive users, you can replace these functions with one or two lines of simple Lisp code.

- Use lists rather than vectors, except when there is a particular reason to use a vector. Lisp has more facilities for manipulating lists than for vectors, and working with lists is usually more convenient.

  Vectors are advantageous for tables that are substantial in size and are accessed in random order (not searched front to back), provided there is no need to insert or delete elements (only lists allow that).
- The recommended way to show a message in the echo area is with the `message` function, not `princ`. See Section 38.4 [The Echo Area], page 302.
- When you encounter an error condition, call the function `error` (or `signal`). The function `error` does not return. See Section 10.5.3.1 [Signaling Errors], page 128, vol. 1.

  Don't use `message`, `throw`, `sleep-for`, or `beep` to report errors.
- An error message should start with a capital letter but should not end with a period.
- A question asked in the minibuffer with `yes-or-no-p` or `y-or-n-p` should start with a capital letter and end with '`?` '.
- When you mention a default value in a minibuffer prompt, put it and the word '`default`' inside parentheses. It should look like this:

```
Enter the answer (default 42):
```

- In `interactive`, if you use a Lisp expression to produce a list of arguments, don't try to provide the "correct" default values for region or position arguments. Instead, provide `nil` for those arguments if they were not specified, and have the function body compute the default value when the argument is `nil`. For instance, write this:

```
(defun foo (pos)
  (interactive
   (list (if specified specified-pos)))
  (unless pos (setq pos default-pos))
  ...)
```

rather than this:

```
(defun foo (pos)
  (interactive
   (list (if specified specified-pos
             default-pos)))
  ...)
```

This is so that repetition of the command will recompute these defaults based on the current circumstances.

You do not need to take such precautions when you use interactive specs '`d`', '`m`' and '`r`', because they make special arrangements to recompute the argument values on repetition of the command.

- Many commands that take a long time to execute display a message that says something like '`Operating...`' when they start, and change it to '`Operating...done`' when they finish. Please keep the style of these messages uniform: *no* space around the ellipsis, and *no* period after '`done`'. See Section 38.4.2 [Progress], page 303, for an easy way to generate such messages.
- Try to avoid using recursive edits. Instead, do what the Rmail `e` command does: use a new local keymap that contains a command defined to switch back to the old local keymap. Or simply switch to another buffer and let the user switch back at will. See Section 21.13 [Recursive Editing], page 355, vol. 1.

D.4 Tips for Making Compiled Code Fast

Here are ways of improving the execution speed of byte-compiled Lisp programs.

- Profile your program with the '`elp`' library. See the file '`elp.el`' for instructions.
- Check the speed of individual Emacs Lisp forms using the '`benchmark`' library. See the functions `benchmark-run` and `benchmark-run-compiled` in '`benchmark.el`'.
- Use iteration rather than recursion whenever possible. Function calls are slow in Emacs Lisp even when a compiled function is calling another compiled function.
- Using the primitive list-searching functions `memq`, `member`, `assq`, or `assoc` is even faster than explicit iteration. It can be worth rearranging a data structure so that one of these primitive search functions can be used.
- Certain built-in functions are handled specially in byte-compiled code, avoiding the need for an ordinary function call. It is a good idea to use these functions rather than alternatives. To see whether a function is handled specially by the compiler, examine its `byte-compile` property. If the property is non-`nil`, then the function is handled specially.

  For example, the following input will show you that `aref` is compiled specially (see Section 6.3 [Array Functions], page 89, vol. 1):

  ```
  (get 'aref 'byte-compile)
       ⇒ byte-compile-two-args
  ```

  Note that in this case (and many others), you must first load the '`bytecomp`' library, which defines the `byte-compile` property.
- If calling a small function accounts for a substantial part of your program's running time, make the function inline. This eliminates the function call overhead. Since making a function inline reduces the flexibility of changing the program, don't do it unless it gives a noticeable speedup in something slow enough that users care about the speed. See Section 12.11 [Inline Functions], page 177, vol. 1.

D.5 Tips for Avoiding Compiler Warnings

- Try to avoid compiler warnings about undefined free variables, by adding dummy `defvar` definitions for these variables, like this:

  ```
  (defvar foo)
  ```

  Such a definition has no effect except to tell the compiler not to warn about uses of the variable `foo` in this file.

- Similarly, to avoid a compiler warning about an undefined function that you know *will* be defined, use a `declare-function` statement (see Section 12.12 [Declaring Functions], page 178, vol. 1).
- If you use many functions and variables from a certain file, you can add a `require` for that package to avoid compilation warnings for them. For instance,

```
(eval-when-compile
  (require 'foo))
```

- If you bind a variable in one function, and use it or set it in another function, the compiler warns about the latter function unless the variable has a definition. But adding a definition would be unclean if the variable has a short name, since Lisp packages should not define short variable names. The right thing to do is to rename this variable to start with the name prefix used for the other functions and variables in your package.
- The last resort for avoiding a warning, when you want to do something that is usually a mistake but you know is not a mistake in your usage, is to put it inside `with-no-warnings`. See Section 16.6 [Compiler Errors], page 228, vol. 1.

D.6 Tips for Documentation Strings

Here are some tips and conventions for the writing of documentation strings. You can check many of these conventions by running the command *`M-x checkdoc-minor-mode`*.

- Every command, function, or variable intended for users to know about should have a documentation string.
- An internal variable or subroutine of a Lisp program might as well have a documentation string. Documentation strings take up very little space in a running Emacs.
- Format the documentation string so that it fits in an Emacs window on an 80-column screen. It is a good idea for most lines to be no wider than 60 characters. The first line should not be wider than 67 characters or it will look bad in the output of `apropos`.

  You can fill the text if that looks good. However, rather than blindly filling the entire documentation string, you can often make it much more readable by choosing certain line breaks with care. Use blank lines between sections if the documentation string is long.
- The first line of the documentation string should consist of one or two complete sentences that stand on their own as a summary. *`M-x apropos`* displays just the first line, and if that line's contents don't stand on their own, the result looks bad. In particular, start the first line with a capital letter and end it with a period.

  For a function, the first line should briefly answer the question, "What does this function do?" For a variable, the first line should briefly answer the question, "What does this value mean?"

  Don't limit the documentation string to one line; use as many lines as you need to explain the details of how to use the function or variable. Please use complete sentences for the rest of the text too.
- When the user tries to use a disabled command, Emacs displays just the first paragraph of its documentation string—everything through the first blank line. If you wish, you

can choose which information to include before the first blank line so as to make this display useful.

- The first line should mention all the important arguments of the function, and should mention them in the order that they are written in a function call. If the function has many arguments, then it is not feasible to mention them all in the first line; in that case, the first line should mention the first few arguments, including the most important arguments.
- When a function's documentation string mentions the value of an argument of the function, use the argument name in capital letters as if it were a name for that value. Thus, the documentation string of the function `eval` refers to its first argument as '`FORM`', because the actual argument name is `form`:

  ```
  Evaluate FORM and return its value.
  ```

  Also write metasyntactic variables in capital letters, such as when you show the decomposition of a list or vector into subunits, some of which may vary. '`KEY`' and '`VALUE`' in the following example illustrate this practice:

  ```
  The argument TABLE should be an alist whose elements
  have the form (KEY . VALUE).  Here, KEY is ...
  ```

- Never change the case of a Lisp symbol when you mention it in a doc string. If the symbol's name is `foo`, write "foo", not "Foo" (which is a different symbol).

  This might appear to contradict the policy of writing function argument values, but there is no real contradiction; the argument *value* is not the same thing as the *symbol* that the function uses to hold the value.

  If this puts a lower-case letter at the beginning of a sentence and that annoys you, rewrite the sentence so that the symbol is not at the start of it.

- Do not start or end a documentation string with whitespace.
- **Do not** indent subsequent lines of a documentation string so that the text is lined up in the source code with the text of the first line. This looks nice in the source code, but looks bizarre when users view the documentation. Remember that the indentation before the starting double-quote is not part of the string!
- When a documentation string refers to a Lisp symbol, write it as it would be printed (which usually means in lower case), with single-quotes around it. For example: '`‘lambda’`'. There are two exceptions: write `t` and `nil` without single-quotes.

  Help mode automatically creates a hyperlink when a documentation string uses a symbol name inside single quotes, if the symbol has either a function or a variable definition. You do not need to do anything special to make use of this feature. However, when a symbol has both a function definition and a variable definition, and you want to refer to just one of them, you can specify which one by writing one of the words '`variable`', '`option`', '`function`', or '`command`', immediately before the symbol name. (Case makes no difference in recognizing these indicator words.) For example, if you write

  ```
  This function sets the variable ‘buffer-file-name’.
  ```

  then the hyperlink will refer only to the variable documentation of `buffer-file-name`, and not to its function documentation.

If a symbol has a function definition and/or a variable definition, but those are irrelevant to the use of the symbol that you are documenting, you can write the words '`symbol`' or '`program`' before the symbol name to prevent making any hyperlink. For example,

```
If the argument KIND-OF-RESULT is the symbol 'list',
this function returns a list of all the objects
that satisfy the criterion.
```

does not make a hyperlink to the documentation, irrelevant here, of the function `list`.

Normally, no hyperlink is made for a variable without variable documentation. You can force a hyperlink for such variables by preceding them with one of the words '`variable`' or '`option`'.

Hyperlinks for faces are only made if the face name is preceded or followed by the word '`face`'. In that case, only the face documentation will be shown, even if the symbol is also defined as a variable or as a function.

To make a hyperlink to Info documentation, write the name of the Info node (or anchor) in single quotes, preceded by '`info node`', '`Info node`', '`info anchor`' or '`Info anchor`'. The Info file name defaults to '`emacs`'. For example,

```
See Info node 'Font Lock' and Info node '(elisp)Font Lock Basics'.
```

Finally, to create a hyperlink to URLs, write the URL in single quotes, preceded by '`URL`'. For example,

```
The home page for the GNU project has more information (see URL
'http://www.gnu.org/').
```

- Don't write key sequences directly in documentation strings. Instead, use the '`\\[...]`' construct to stand for them. For example, instead of writing '`C-f`', write the construct '`\\[forward-char]`'. When Emacs displays the documentation string, it substitutes whatever key is currently bound to `forward-char`. (This is normally '`C-f`', but it may be some other character if the user has moved key bindings.) See Section 24.3 [Keys in Documentation], page 454, vol. 1.
- In documentation strings for a major mode, you will want to refer to the key bindings of that mode's local map, rather than global ones. Therefore, use the construct '`\\<...>`' once in the documentation string to specify which key map to use. Do this before the first use of '`\\[...]`'. The text inside the '`\\<...>`' should be the name of the variable containing the local keymap for the major mode.

  It is not practical to use '`\\[...]`' very many times, because display of the documentation string will become slow. So use this to describe the most important commands in your major mode, and then use '`\\{...}`' to display the rest of the mode's keymap.
- For consistency, phrase the verb in the first sentence of a function's documentation string as an imperative—for instance, use "Return the cons of A and B." in preference to "Returns the cons of A and B." Usually it looks good to do likewise for the rest of the first paragraph. Subsequent paragraphs usually look better if each sentence is indicative and has a proper subject.
- The documentation string for a function that is a yes-or-no predicate should start with words such as "Return t if", to indicate explicitly what constitutes "truth". The word "return" avoids starting the sentence with lower-case "t", which could be somewhat distracting.

- If a line in a documentation string begins with an open-parenthesis, write a backslash before the open-parenthesis, like this:

  ```
  The argument FOO can be either a number
  \(a buffer position) or a string (a file name).
  ```

  This prevents the open-parenthesis from being treated as the start of a defun (see Section "Defuns" in *The GNU Emacs Manual*).
- Write documentation strings in the active voice, not the passive, and in the present tense, not the future. For instance, use "Return a list containing A and B." instead of "A list containing A and B will be returned."
- Avoid using the word "cause" (or its equivalents) unnecessarily. Instead of, "Cause Emacs to display text in boldface", write just "Display text in boldface".
- Avoid using "iff" (a mathematics term meaning "if and only if"), since many people are unfamiliar with it and mistake it for a typo. In most cases, the meaning is clear with just "if". Otherwise, try to find an alternate phrasing that conveys the meaning.
- When a command is meaningful only in a certain mode or situation, do mention that in the documentation string. For example, the documentation of `dired-find-file` is:

  ```
  In Dired, visit the file or directory named on this line.
  ```
- When you define a variable that represents an option users might want to set, use `defcustom`. See Section 11.5 [Defining Variables], page 141, vol. 1.
- The documentation string for a variable that is a yes-or-no flag should start with words such as "Non-nil means", to make it clear that all non-`nil` values are equivalent and indicate explicitly what `nil` and non-`nil` mean.

D.7 Tips on Writing Comments

We recommend these conventions for comments:

'`;`' Comments that start with a single semicolon, '`;`', should all be aligned to the same column on the right of the source code. Such comments usually explain how the code on that line does its job. For example:

```
(setq base-version-list                 ; there was a base
      (assoc (substring fn 0 start-vn)  ; version to which
             file-version-assoc-list))  ; this looks like
                                        ; a subversion
```

'`;;`' Comments that start with two semicolons, '`;;`', should be aligned to the same level of indentation as the code. Such comments usually describe the purpose of the following lines or the state of the program at that point. For example:

```
(prog1 (setq auto-fill-function
             ...
             ...
  ;; Update mode line.
  (force-mode-line-update)))
```

We also normally use two semicolons for comments outside functions.

```
;; This Lisp code is run in Emacs when it is to operate as
;; a server for other processes.
```

If a function has no documentation string, it should instead have a two-semicolon comment right before the function, explaining what the function

does and how to call it properly. Explain precisely what each argument means and how the function interprets its possible values. It is much better to convert such comments to documentation strings, though.

‘;;;’ Comments that start with three semicolons, ‘;;;’, should start at the left margin. These are used, occasionally, for comments within functions that should start at the margin. We also use them sometimes for comments that are between functions—whether to use two or three semicolons depends on whether the comment should be considered a “heading” by Outline minor mode. By default, comments starting with at least three semicolons (followed by a single space and a non-whitespace character) are considered headings, comments starting with two or fewer are not.

Another use for triple-semicolon comments is for commenting out lines within a function. We use three semicolons for this precisely so that they remain at the left margin. By default, Outline minor mode does not consider a comment to be a heading (even if it starts with at least three semicolons) if the semicolons are followed by at least two spaces. Thus, if you add an introductory comment to the commented out code, make sure to indent it by at least two spaces after the three semicolons.

```
(defun foo (a)
;;;  This is no longer necessary.
;;;  (force-mode-line-update)
  (message "Finished with %s" a))
```

When commenting out entire functions, use two semicolons.

‘;;;;’ Comments that start with four semicolons, ‘;;;;’, should be aligned to the left margin and are used for headings of major sections of a program. For example:

```
;;;; The kill ring
```

Generally speaking, the *M-;* (`comment-dwim`) command automatically starts a comment of the appropriate type; or indents an existing comment to the right place, depending on the number of semicolons. See Section “Manipulating Comments” in *The GNU Emacs Manual*.

D.8 Conventional Headers for Emacs Libraries

Emacs has conventions for using special comments in Lisp libraries to divide them into sections and give information such as who wrote them. Using a standard format for these items makes it easier for tools (and people) to extract the relevant information. This section explains these conventions, starting with an example:

```
;;; foo.el --- Support for the Foo programming language

;; Copyright (C) 2010-2012 Your Name

;; Author: Your Name <yourname@example.com>
;; Maintainer: Someone Else <someone@example.com>
;; Created: 14 Jul 2010
```

```
;; Keywords: languages

;; This file is not part of GNU Emacs.

;; This file is free software...
...
;; along with this file.  If not, see <http://www.gnu.org/licenses/>.
```

The very first line should have this format:

```
;;; filename --- description
```

The description should be contained in one line. If the file needs a '`-*-`' specification, put it after *description*. If this would make the first line too long, use a Local Variables section at the end of the file.

The copyright notice usually lists your name (if you wrote the file). If you have an employer who claims copyright on your work, you might need to list them instead. Do not say that the copyright holder is the Free Software Foundation (or that the file is part of GNU Emacs) unless your file has been accepted into the Emacs distribution. For more information on the form of copyright and license notices, see the guide on the GNU website.

After the copyright notice come several *header comment* lines, each beginning with '`;;` *header-name*`:`'. Here is a table of the conventional possibilities for *header-name*:

'`Author`' This line states the name and email address of at least the principal author of the library. If there are multiple authors, list them on continuation lines led by `;;` and whitespace (this is easier for tools to parse than having more than one author on one line). We recommend including a contact email address, of the form '`<...>`'. For example:

```
;; Author: Your Name <yourname@example.com>
;;       Someone Else <someone@example.com>
;;       Another Person <another@example.com>
```

'`Maintainer`'
This header has the same format as the Author header. It lists the person(s) who currently maintain(s) the file (respond to bug reports, etc.).

If there is no maintainer line, the person(s) in the Author field is/are presumed to be the maintainers. Some files in Emacs use '`FSF`' for the maintainer. This means that the original author is no longer responsible for the file, and that it is maintained as part of Emacs.

'`Created`' This optional line gives the original creation date of the file, and is for historical interest only.

'`Version`' If you wish to record version numbers for the individual Lisp program, put them in this line. Lisp files distributed with Emacs generally do not have a '`Version`' header, since the version number of Emacs itself serves the same purpose. If you are distributing a collection of multiple files, we recommend not writing the version in every file, but only the main one.

'`Keywords`'
This line lists keywords for the `finder-by-keyword` help command. Please use that command to see a list of the meaningful keywords.

This field is how people will find your package when they're looking for things by topic. To separate the keywords, you can use spaces, commas, or both.

The name of this field is unfortunate, since people often assume it is the place to write arbitrary keywords that describe their package, rather than just the relevant Finder keywords.

'`Package-Version`'
: If '`Version`' is not suitable for use by the package manager, then a package can define '`Package-Version`'; it will be used instead. This is handy if '`Version`' is an RCS id or something else that cannot be parsed by `version-to-list`. See Section 40.1 [Packaging Basics], page 418.

'`Package-Requires`'
: If this exists, it names packages on which the current package depends for proper operation. See Section 40.1 [Packaging Basics], page 418. This is used by the package manager both at download time (to ensure that a complete set of packages is downloaded) and at activation time (to ensure that a package is only activated if all its dependencies have been).

  Its format is a list of lists. The `car` of each sub-list is the name of a package, as a symbol. The `cadr` of each sub-list is the minimum acceptable version number, as a string. For instance:

```
;; Package-Requires: ((gnus "1.0") (bubbles "2.7.2"))
```

  The package code automatically defines a package named '`emacs`' with the version number of the currently running Emacs. This can be used to require a minimal version of Emacs for a package.

Just about every Lisp library ought to have the '`Author`' and '`Keywords`' header comment lines. Use the others if they are appropriate. You can also put in header lines with other header names—they have no standard meanings, so they can't do any harm.

We use additional stylized comments to subdivide the contents of the library file. These should be separated from anything else by blank lines. Here is a table of them:

'`;;; Commentary:`'
: This begins introductory comments that explain how the library works. It should come right after the copying permissions, terminated by a '`Change Log`', '`History`' or '`Code`' comment line. This text is used by the Finder package, so it should make sense in that context.

'`;;; Change Log:`'
: This begins an optional log of changes to the file over time. Don't put too much information in this section—it is better to keep the detailed logs in a separate '`ChangeLog`' file (as Emacs does), and/or to use a version control system. '`History`' is an alternative to '`Change Log`'.

'`;;; Code:`'
: This begins the actual code of the program.

'`;;; `*`filename`*` ends here`'
: This is the *footer line*; it appears at the very end of the file. Its purpose is to enable people to detect truncated versions of the file from the lack of a footer line.

Appendix E GNU Emacs Internals

This chapter describes how the runnable Emacs executable is dumped with the preloaded Lisp libraries in it, how storage is allocated, and some internal aspects of GNU Emacs that may be of interest to C programmers.

E.1 Building Emacs

This section explains the steps involved in building the Emacs executable. You don't have to know this material to build and install Emacs, since the makefiles do all these things automatically. This information is pertinent to Emacs developers.

Compilation of the C source files in the '`src`' directory produces an executable file called '`temacs`', also called a *bare impure Emacs*. It contains the Emacs Lisp interpreter and I/O routines, but not the editing commands.

The command `temacs -l loadup` would run '`temacs`' and direct it to load '`loadup.el`'. The `loadup` library loads additional Lisp libraries, which set up the normal Emacs editing environment. After this step, the Emacs executable is no longer *bare*.

Because it takes some time to load the standard Lisp files, the '`temacs`' executable usually isn't run directly by users. Instead, as one of the last steps of building Emacs, the command '`temacs -batch -l loadup dump`' is run. The special '`dump`' argument causes `temacs` to dump out an executable program, called '`emacs`', which has all the standard Lisp files preloaded. (The '`-batch`' argument prevents '`temacs`' from trying to initialize any of its data on the terminal, so that the tables of terminal information are empty in the dumped Emacs.)

The dumped '`emacs`' executable (also called a *pure* Emacs) is the one which is installed. The variable `preloaded-file-list` stores a list of the Lisp files preloaded into the dumped Emacs. If you port Emacs to a new operating system, and are not able to implement dumping, then Emacs must load '`loadup.el`' each time it starts.

You can specify additional files to preload by writing a library named '`site-load.el`' that loads them. You may need to rebuild Emacs with an added definition

```
#define SITELOAD_PURESIZE_EXTRA n
```

to make *n* added bytes of pure space to hold the additional files; see '`src/puresize.h`'. (Try adding increments of 20000 until it is big enough.) However, the advantage of preloading additional files decreases as machines get faster. On modern machines, it is usually not advisable.

After '`loadup.el`' reads '`site-load.el`', it finds the documentation strings for primitive and preloaded functions (and variables) in the file '`etc/DOC`' where they are stored, by calling `Snarf-documentation` (see [Accessing Documentation], page 454, vol. 1).

You can specify other Lisp expressions to execute just before dumping by putting them in a library named '`site-init.el`'. This file is executed after the documentation strings are found.

If you want to preload function or variable definitions, there are three ways you can do this and make their documentation strings accessible when you subsequently run Emacs:

- Arrange to scan these files when producing the '`etc/DOC`' file, and load them with '`site-load.el`'.

- Load the files with '`site-init.el`', then copy the files into the installation directory for Lisp files when you install Emacs.
- Specify a `nil` value for `byte-compile-dynamic-docstrings` as a local variable in each of these files, and load them with either '`site-load.el`' or '`site-init.el`'. (This method has the drawback that the documentation strings take up space in Emacs all the time.)

It is not advisable to put anything in '`site-load.el`' or '`site-init.el`' that would alter any of the features that users expect in an ordinary unmodified Emacs. If you feel you must override normal features for your site, do it with '`default.el`', so that users can override your changes if they wish. See Section 39.1.1 [Startup Summary], page 386.

In a package that can be preloaded, it is sometimes necessary (or useful) to delay certain evaluations until Emacs subsequently starts up. The vast majority of such cases relate to the values of customizable variables. For example, `tutorial-directory` is a variable defined in '`startup.el`', which is preloaded. The default value is set based on `data-directory`. The variable needs to access the value of `data-directory` when Emacs starts, not when it is dumped, because the Emacs executable has probably been installed in a different location since it was dumped.

`custom-initialize-delay` *symbol value* [Function]
: This function delays the initialization of *symbol* to the next Emacs start. You normally use this function by specifying it as the `:initialize` property of a customizable variable. (The argument *value* is unused, and is provided only for compatibility with the form Custom expects.)

In the unlikely event that you need a more general functionality than `custom-initialize-delay` provides, you can use `before-init-hook` (see Section 39.1.1 [Startup Summary], page 386).

`dump-emacs` *to-file from-file* [Function]
: This function dumps the current state of Emacs into an executable file *to-file*. It takes symbols from *from-file* (this is normally the executable file '`temacs`').

  If you want to use this function in an Emacs that was already dumped, you must run Emacs with '`-batch`'.

E.2 Pure Storage

Emacs Lisp uses two kinds of storage for user-created Lisp objects: *normal storage* and *pure storage*. Normal storage is where all the new data created during an Emacs session are kept (see Section E.3 [Garbage Collection], page 459). Pure storage is used for certain data in the preloaded standard Lisp files—data that should never change during actual use of Emacs.

Pure storage is allocated only while `temacs` is loading the standard preloaded Lisp libraries. In the file '`emacs`', it is marked as read-only (on operating systems that permit this), so that the memory space can be shared by all the Emacs jobs running on the machine at once. Pure storage is not expandable; a fixed amount is allocated when Emacs is compiled, and if that is not sufficient for the preloaded libraries, '`temacs`' allocates dynamic memory for the part that didn't fit. The resulting image will work, but garbage collection

(see Section E.3 [Garbage Collection], page 459) is disabled in this situation, causing a memory leak. Such an overflow normally won't happen unless you try to preload additional libraries or add features to the standard ones. Emacs will display a warning about the overflow when it starts. If this happens, you should increase the compilation parameter `SYSTEM_PURESIZE_EXTRA` in the file '`src/puresize.h`' and rebuild Emacs.

`purecopy` *object* [Function]

This function makes a copy in pure storage of *object*, and returns it. It copies a string by simply making a new string with the same characters, but without text properties, in pure storage. It recursively copies the contents of vectors and cons cells. It does not make copies of other objects such as symbols, but just returns them unchanged. It signals an error if asked to copy markers.

This function is a no-op except while Emacs is being built and dumped; it is usually called only in preloaded Lisp files.

`pure-bytes-used` [Variable]

The value of this variable is the number of bytes of pure storage allocated so far. Typically, in a dumped Emacs, this number is very close to the total amount of pure storage available—if it were not, we would preallocate less.

`purify-flag` [Variable]

This variable determines whether `defun` should make a copy of the function definition in pure storage. If it is non-`nil`, then the function definition is copied into pure storage.

This flag is `t` while loading all of the basic functions for building Emacs initially (allowing those functions to be shareable and non-collectible). Dumping Emacs as an executable always writes `nil` in this variable, regardless of the value it actually has before and after dumping.

You should not change this flag in a running Emacs.

E.3 Garbage Collection

When a program creates a list or the user defines a new function (such as by loading a library), that data is placed in normal storage. If normal storage runs low, then Emacs asks the operating system to allocate more memory. Different types of Lisp objects, such as symbols, cons cells, markers, etc., are segregated in distinct blocks in memory. (Vectors, long strings, buffers and certain other editing types, which are fairly large, are allocated in individual blocks, one per object, while small strings are packed into blocks of 8k bytes.)

It is quite common to use some storage for a while, then release it by (for example) killing a buffer or deleting the last pointer to an object. Emacs provides a *garbage collector* to reclaim this abandoned storage. The garbage collector operates by finding and marking all Lisp objects that are still accessible to Lisp programs. To begin with, it assumes all the symbols, their values and associated function definitions, and any data presently on the stack, are accessible. Any objects that can be reached indirectly through other accessible objects are also accessible.

When marking is finished, all objects still unmarked are garbage. No matter what the Lisp program or the user does, it is impossible to refer to them, since there is no longer a

way to reach them. Their space might as well be reused, since no one will miss them. The second ("sweep") phase of the garbage collector arranges to reuse them.

The sweep phase puts unused cons cells onto a *free list* for future allocation; likewise for symbols and markers. It compacts the accessible strings so they occupy fewer 8k blocks; then it frees the other 8k blocks. Vectors, buffers, windows, and other large objects are individually allocated and freed using `malloc` and `free`.

> **Common Lisp note:** Unlike other Lisps, GNU Emacs Lisp does not call the garbage collector when the free list is empty. Instead, it simply requests the operating system to allocate more storage, and processing continues until `gc-cons-threshold` bytes have been used.
>
> This means that you can make sure that the garbage collector will not run during a certain portion of a Lisp program by calling the garbage collector explicitly just before it (provided that portion of the program does not use so much space as to force a second garbage collection).

`garbage-collect` [Command]

This command runs a garbage collection, and returns information on the amount of space in use. (Garbage collection can also occur spontaneously if you use more than `gc-cons-threshold` bytes of Lisp data since the previous garbage collection.)

`garbage-collect` returns a list containing the following information:

```
((used-conses . free-conses)
 (used-syms . free-syms)
 (used-miscs . free-miscs)
 used-string-chars
 used-vector-slots
 (used-floats . free-floats)
 (used-intervals . free-intervals)
 (used-strings . free-strings))
```

Here is an example:

```
(garbage-collect)
     ⇒ ((106886 . 13184) (9769 . 0)
                (7731 . 4651) 347543 121628
                (31 . 94) (1273 . 168)
                (25474 . 3569))
```

Here is a table explaining each element:

used-conses
: The number of cons cells in use.

free-conses
: The number of cons cells for which space has been obtained from the operating system, but that are not currently being used.

used-syms The number of symbols in use.

free-syms The number of symbols for which space has been obtained from the operating system, but that are not currently being used.

used-miscs
: The number of miscellaneous objects in use. These include markers and overlays, plus certain objects not visible to users.

free-miscs
: The number of miscellaneous objects for which space has been obtained from the operating system, but that are not currently being used.

used-string-chars
: The total size of all strings, in characters.

used-vector-slots
: The total number of elements of existing vectors.

used-floats
: The number of floats in use.

free-floats
: The number of floats for which space has been obtained from the operating system, but that are not currently being used.

used-intervals
: The number of intervals in use. Intervals are an internal data structure used for representing text properties.

free-intervals
: The number of intervals for which space has been obtained from the operating system, but that are not currently being used.

used-strings
: The number of strings in use.

free-strings
: The number of string headers for which the space was obtained from the operating system, but which are currently not in use. (A string object consists of a header and the storage for the string text itself; the latter is only allocated when the string is created.)

If there was overflow in pure space (see Section E.2 [Pure Storage], page 458), `garbage-collect` returns `nil`, because a real garbage collection cannot be done.

`garbage-collection-messages` [User Option]
If this variable is non-`nil`, Emacs displays a message at the beginning and end of garbage collection. The default value is `nil`.

`post-gc-hook` [Variable]
This is a normal hook that is run at the end of garbage collection. Garbage collection is inhibited while the hook functions run, so be careful writing them.

`gc-cons-threshold` [User Option]
The value of this variable is the number of bytes of storage that must be allocated for Lisp objects after one garbage collection in order to trigger another garbage collection. A cons cell counts as eight bytes, a string as one byte per character plus a few bytes of overhead, and so on; space allocated to the contents of buffers does not count. Note that the subsequent garbage collection does not happen immediately when the threshold is exhausted, but only the next time the Lisp evaluator is called.

The initial threshold value is 800,000. If you specify a larger value, garbage collection will happen less often. This reduces the amount of time spent garbage collecting, but increases total memory use. You may want to do this when running a program that creates lots of Lisp data.

You can make collections more frequent by specifying a smaller value, down to 10,000. A value less than 10,000 will remain in effect only until the subsequent garbage collection, at which time `garbage-collect` will set the threshold back to 10,000.

`gc-cons-percentage` [User Option]

The value of this variable specifies the amount of consing before a garbage collection occurs, as a fraction of the current heap size. This criterion and `gc-cons-threshold` apply in parallel, and garbage collection occurs only when both criteria are satisfied.

As the heap size increases, the time to perform a garbage collection increases. Thus, it can be desirable to do them less frequently in proportion.

The value returned by `garbage-collect` describes the amount of memory used by Lisp data, broken down by data type. By contrast, the function `memory-limit` provides information on the total amount of memory Emacs is currently using.

`memory-limit` [Function]

This function returns the address of the last byte Emacs has allocated, divided by 1024. We divide the value by 1024 to make sure it fits in a Lisp integer.

You can use this to get a general idea of how your actions affect the memory usage.

`memory-full` [Variable]

This variable is `t` if Emacs is nearly out of memory for Lisp objects, and `nil` otherwise.

`memory-use-counts` [Function]

This returns a list of numbers that count the number of objects created in this Emacs session. Each of these counters increments for a certain kind of object. See the documentation string for details.

`gcs-done` [Variable]

This variable contains the total number of garbage collections done so far in this Emacs session.

`gc-elapsed` [Variable]

This variable contains the total number of seconds of elapsed time during garbage collection so far in this Emacs session, as a floating point number.

E.4 Memory Usage

These functions and variables give information about the total amount of memory allocation that Emacs has done, broken down by data type. Note the difference between these and the values returned by `garbage-collect`; those count objects that currently exist, but these count the number or size of all allocations, including those for objects that have since been freed.

`cons-cells-consed` [Variable]

The total number of cons cells that have been allocated so far in this Emacs session.

`floats-consed` [Variable]
The total number of floats that have been allocated so far in this Emacs session.

`vector-cells-consed` [Variable]
The total number of vector cells that have been allocated so far in this Emacs session.

`symbols-consed` [Variable]
The total number of symbols that have been allocated so far in this Emacs session.

`string-chars-consed` [Variable]
The total number of string characters that have been allocated so far in this session.

`misc-objects-consed` [Variable]
The total number of miscellaneous objects that have been allocated so far in this session. These include markers and overlays, plus certain objects not visible to users.

`intervals-consed` [Variable]
The total number of intervals that have been allocated so far in this Emacs session.

`strings-consed` [Variable]
The total number of strings that have been allocated so far in this Emacs session.

E.5 Writing Emacs Primitives

Lisp primitives are Lisp functions implemented in C. The details of interfacing the C function so that Lisp can call it are handled by a few C macros. The only way to really understand how to write new C code is to read the source, but we can explain some things here.

An example of a special form is the definition of `or`, from '`eval.c`'. (An ordinary function would have the same general appearance.)

```
DEFUN ("or", For, Sor, 0, UNEVALLED, 0,
  doc: /* Eval args until one of them yields non-nil, then return
that value.
The remaining args are not evalled at all.
If all args return nil, return nil.
usage: (or CONDITIONS ...)  */)
  (Lisp_Object args)
{
  register Lisp_Object val = Qnil;
  struct gcpro gcpro1;

  GCPRO1 (args);

  while (CONSP (args))
    {
      val = eval_sub (XCAR (args));
      if (!NILP (val))
        break;
      args = XCDR (args);
    }

  UNGCPRO;
  return val;
}
```

Let's start with a precise explanation of the arguments to the `DEFUN` macro. Here is a template for them:

```
DEFUN (lname, fname, sname, min, max, interactive, doc)
```

lname This is the name of the Lisp symbol to define as the function name; in the example above, it is `or`.

fname This is the C function name for this function. This is the name that is used in C code for calling the function. The name is, by convention, '`F`' prepended to the Lisp name, with all dashes ('`-`') in the Lisp name changed to underscores. Thus, to call this function from C code, call `For`.

sname This is a C variable name to use for a structure that holds the data for the subr object that represents the function in Lisp. This structure conveys the Lisp symbol name to the initialization routine that will create the symbol and store the subr object as its definition. By convention, this name is always *fname* with '`F`' replaced with '`S`'.

min This is the minimum number of arguments that the function requires. The function `or` allows a minimum of zero arguments.

max This is the maximum number of arguments that the function accepts, if there is a fixed maximum. Alternatively, it can be `UNEVALLED`, indicating a special form that receives unevaluated arguments, or `MANY`, indicating an unlimited number of evaluated arguments (the equivalent of `&rest`). Both `UNEVALLED` and `MANY` are macros. If *max* is a number, it must be more than *min* but less than 8.

interactive

This is an interactive specification, a string such as might be used as the argument of `interactive` in a Lisp function. In the case of `or`, it is 0 (a null pointer), indicating that `or` cannot be called interactively. A value of `""` indicates a function that should receive no arguments when called interactively. If the value begins with a '`(`', the string is evaluated as a Lisp form. For examples of the last two forms, see `widen` and `narrow-to-region` in '`editfns.c`'.

doc This is the documentation string. It uses C comment syntax rather than C string syntax because comment syntax requires nothing special to include multiple lines. The '`doc:`' identifies the comment that follows as the documentation string. The '`/*`' and '`*/`' delimiters that begin and end the comment are not part of the documentation string.

If the last line of the documentation string begins with the keyword '`usage:`', the rest of the line is treated as the argument list for documentation purposes. This way, you can use different argument names in the documentation string from the ones used in the C code. '`usage:`' is required if the function has an unlimited number of arguments.

All the usual rules for documentation strings in Lisp code (see Section D.6 [Documentation Tips], page 450) apply to C code documentation strings too.

After the call to the `DEFUN` macro, you must write the argument list for the C function, including the types for the arguments. If the primitive accepts a fixed maximum number of Lisp arguments, there must be one C argument for each Lisp argument, and each argument must be of type `Lisp_Object`. (Various macros and functions for creating values of type `Lisp_Object` are declared in the file '`lisp.h`'.) If the primitive has no upper limit on the

number of Lisp arguments, it must have exactly two C arguments: the first is the number of Lisp arguments, and the second is the address of a block containing their values. These have types `int` and `Lisp_Object *` respectively.

Within the function `For` itself, note the use of the macros `GCPRO1` and `UNGCPRO`. These macros are defined for the sake of the few platforms which do not use Emacs' default stack-marking garbage collector. The `GCPRO1` macro "protects" a variable from garbage collection, explicitly informing the garbage collector that that variable and all its contents must be as accessible. GC protection is necessary in any function which can perform Lisp evaluation by calling `eval_sub` or `Feval` as a subroutine, either directly or indirectly.

It suffices to ensure that at least one pointer to each object is GC-protected. Thus, a particular local variable can do without protection if it is certain that the object it points to will be preserved by some other pointer (such as another local variable that has a `GCPRO`). Otherwise, the local variable needs a `GCPRO`.

The macro `GCPRO1` protects just one local variable. If you want to protect two variables, use `GCPRO2` instead; repeating `GCPRO1` will not work. Macros `GCPRO3`, `GCPRO4`, `GCPRO5`, and `GCPRO6` also exist. All these macros implicitly use local variables such as `gcpro1`; you must declare these explicitly, with type `struct gcpro`. Thus, if you use `GCPRO2`, you must declare `gcpro1` and `gcpro2`.

`UNGCPRO` cancels the protection of the variables that are protected in the current function. It is necessary to do this explicitly.

You must not use C initializers for static or global variables unless the variables are never written once Emacs is dumped. These variables with initializers are allocated in an area of memory that becomes read-only (on certain operating systems) as a result of dumping Emacs. See Section E.2 [Pure Storage], page 458.

Defining the C function is not enough to make a Lisp primitive available; you must also create the Lisp symbol for the primitive and store a suitable subr object in its function cell. The code looks like this:

```
defsubr (&sname);
```

Here *sname* is the name you used as the third argument to `DEFUN`.

If you add a new primitive to a file that already has Lisp primitives defined in it, find the function (near the end of the file) named `syms_of_`*`something`*, and add the call to `defsubr` there. If the file doesn't have this function, or if you create a new file, add to it a `syms_of_`*`filename`* (e.g., `syms_of_myfile`). Then find the spot in '`emacs.c`' where all of these functions are called, and add a call to `syms_of_`*`filename`* there.

The function `syms_of_`*`filename`* is also the place to define any C variables that are to be visible as Lisp variables. `DEFVAR_LISP` makes a C variable of type `Lisp_Object` visible in Lisp. `DEFVAR_INT` makes a C variable of type `int` visible in Lisp with a value that is always an integer. `DEFVAR_BOOL` makes a C variable of type `int` visible in Lisp with a value that is either `t` or `nil`. Note that variables defined with `DEFVAR_BOOL` are automatically added to the list `byte-boolean-vars` used by the byte compiler.

If you want to make a Lisp variables that is defined in C behave like one declared with `defcustom`, add an appropriate entry to '`cus-start.el`'.

If you define a file-scope C variable of type `Lisp_Object`, you must protect it from garbage-collection by calling `staticpro` in `syms_of_`*`filename`*, like this:

```
staticpro (&variable);
```

Here is another example function, with more complicated arguments. This comes from the code in 'window.c', and it demonstrates the use of macros and functions to manipulate Lisp objects.

```
DEFUN ("coordinates-in-window-p", Fcoordinates_in_window_p,
  Scoordinates_in_window_p, 2, 2, 0,
  doc: /* Return non-nil if COORDINATES are in WINDOW.
  ...
  or 'right-margin' is returned.  */)
  (register Lisp_Object coordinates, Lisp_Object window)
{
  struct window *w;
  struct frame *f;
  int x, y;
  Lisp_Object lx, ly;

  CHECK_LIVE_WINDOW (window);
  w = XWINDOW (window);
  f = XFRAME (w->frame);
  CHECK_CONS (coordinates);
  lx = Fcar (coordinates);
  ly = Fcdr (coordinates);
  CHECK_NUMBER_OR_FLOAT (lx);
  CHECK_NUMBER_OR_FLOAT (ly);
  x = FRAME_PIXEL_X_FROM_CANON_X (f, lx) + FRAME_INTERNAL_BORDER_WIDTH(f);
  y = FRAME_PIXEL_Y_FROM_CANON_Y (f, ly) + FRAME_INTERNAL_BORDER_WIDTH(f);

  switch (coordinates_in_window (w, x, y))
    {
    case ON_NOTHING:            /* NOT in window at all. */
      return Qnil;

    ...

    case ON_MODE_LINE:          /* In mode line of window. */
      return Qmode_line;

    ...

    case ON_SCROLL_BAR:         /* On scroll-bar of window.  */
      /* Historically we are supposed to return nil in this case.  */
      return Qnil;

    default:
      abort ();
    }
}
```

Note that C code cannot call functions by name unless they are defined in C. The way to call a function written in Lisp is to use `Ffuncall`, which embodies the Lisp function `funcall`. Since the Lisp function `funcall` accepts an unlimited number of arguments, in C it takes two: the number of Lisp-level arguments, and a one-dimensional array containing their values. The first Lisp-level argument is the Lisp function to call, and the rest are the arguments to pass to it. Since `Ffuncall` can call the evaluator, you must protect pointers from garbage collection around the call to `Ffuncall`.

The C functions `call0`, `call1`, `call2`, and so on, provide handy ways to call a Lisp function conveniently with a fixed number of arguments. They work by calling `Ffuncall`.

'`eval.c`' is a very good file to look through for examples; '`lisp.h`' contains the definitions for some important macros and functions.

If you define a function which is side-effect free, update the code in '`byte-opt.el`' that binds `side-effect-free-fns` and `side-effect-and-error-free-fns` so that the compiler optimizer knows about it.

E.6 Object Internals

GNU Emacs Lisp manipulates many different types of data. The actual data are stored in a heap and the only access that programs have to it is through pointers. Each pointer is 32 bits wide on 32-bit machines, and 64 bits wide on 64-bit machines; three of these bits are used for the tag that identifies the object's type, and the remainder are used to address the object.

Because Lisp objects are represented as tagged pointers, it is always possible to determine the Lisp data type of any object. The C data type `Lisp_Object` can hold any Lisp object of any data type. Ordinary variables have type `Lisp_Object`, which means they can hold any type of Lisp value; you can determine the actual data type only at run time. The same is true for function arguments; if you want a function to accept only a certain type of argument, you must check the type explicitly using a suitable predicate (see Section 2.6 [Type Predicates], page 27, vol. 1).

E.6.1 Buffer Internals

Two structures (see '`buffer.h`') are used to represent buffers in C. The `buffer_text` structure contains fields describing the text of a buffer; the `buffer` structure holds other fields. In the case of indirect buffers, two or more `buffer` structures reference the same `buffer_text` structure.

Here are some of the fields in `struct buffer_text`:

`beg` The address of the buffer contents.

`gpt`
`gpt_byte` The character and byte positions of the buffer gap. See Section 27.13 [Buffer Gap], page 16.

`z`
`z_byte` The character and byte positions of the end of the buffer text.

`gap_size` The size of buffer's gap. See Section 27.13 [Buffer Gap], page 16.

`modiff`
`save_modiff`
`chars_modiff`
`overlay_modiff`
These fields count the number of buffer-modification events performed in this buffer. `modiff` is incremented after each buffer-modification event, and is never otherwise changed; `save_modiff` contains the value of `modiff` the last time the buffer was visited or saved; `chars_modiff` counts only modifications to

the characters in the buffer, ignoring all other kinds of changes; and `overlay_modiff` counts only modifications to the overlays.

`beg_unchanged`
`end_unchanged`
: The number of characters at the start and end of the text that are known to be unchanged since the last complete redisplay.

`unchanged_modified`
`overlay_unchanged_modified`
: The values of `modiff` and `overlay_modiff`, respectively, after the last complete redisplay. If their current values match `modiff` or `overlay_modiff`, that means `beg_unchanged` and `end_unchanged` contain no useful information.

`markers`
: The markers that refer to this buffer. This is actually a single marker, and successive elements in its marker `chain` are the other markers referring to this buffer text.

`intervals`
: The interval tree which records the text properties of this buffer.

Some of the fields of `struct buffer` are:

`header`
: A `struct vectorlike_header` structure where `header.next` points to the next buffer, in the chain of all buffers (including killed buffers). This chain is used only for garbage collection, in order to collect killed buffers properly. Note that vectors, and most kinds of objects allocated as vectors, are all on one chain, but buffers are on a separate chain of their own.

`own_text`
: A `struct buffer_text` structure that ordinarily holds the buffer contents. In indirect buffers, this field is not used.

`text`
: A pointer to the `buffer_text` structure for this buffer. In an ordinary buffer, this is the `own_text` field above. In an indirect buffer, this is the `own_text` field of the base buffer.

`pt`
`pt_byte`
: The character and byte positions of point in a buffer.

`begv`
`begv_byte`
: The character and byte positions of the beginning of the accessible range of text in the buffer.

`zv`
`zv_byte`
: The character and byte positions of the end of the accessible range of text in the buffer.

`base_buffer`
: In an indirect buffer, this points to the base buffer. In an ordinary buffer, it is null.

`local_flags`
: This field contains flags indicating that certain variables are local in this buffer. Such variables are declared in the C code using `DEFVAR_PER_BUFFER`, and their

buffer-local bindings are stored in fields in the buffer structure itself. (Some of these fields are described in this table.)

`modtime` The modification time of the visited file. It is set when the file is written or read. Before writing the buffer into a file, this field is compared to the modification time of the file to see if the file has changed on disk. See Section 27.5 [Buffer Modification], page 7.

`auto_save_modified`
: The time when the buffer was last auto-saved.

`last_window_start`
: The `window-start` position in the buffer as of the last time the buffer was displayed in a window.

`clip_changed`
: This flag indicates that narrowing has changed in the buffer. See Section 30.4 [Narrowing], page 109.

`prevent_redisplay_optimizations_p`
: This flag indicates that redisplay optimizations should not be used to display this buffer.

`overlay_center`
: This field holds the current overlay center position. See Section 38.9.1 [Managing Overlays], page 316.

`overlays_before`
`overlays_after`
: These fields hold, respectively, a list of overlays that end at or before the current overlay center, and a list of overlays that end after the current overlay center. See Section 38.9.1 [Managing Overlays], page 316. `overlays_before` is sorted in order of decreasing end position, and `overlays_after` is sorted in order of increasing beginning position.

`name` A Lisp string that names the buffer. It is guaranteed to be unique. See Section 27.3 [Buffer Names], page 4.

`save_length`
: The length of the file this buffer is visiting, when last read or saved. This and other fields concerned with saving are not kept in the `buffer_text` structure because indirect buffers are never saved.

`directory`
: The directory for expanding relative file names. This is the value of the buffer-local variable `default-directory` (see Section 25.8.4 [File Name Expansion], page 486, vol. 1).

`filename` The name of the file visited in this buffer, or `nil`. This is the value of the buffer-local variable `buffer-file-name` (see Section 27.4 [Buffer File Name], page 5).

`undo_list`
`backed_up`
`auto_save_file_name`
`auto_save_file_format`
`read_only`
`file_format`
`file_truename`
`invisibility_spec`
`display_count`
`display_time`
: These fields store the values of Lisp variables that are automatically buffer-local (see Section 11.10 [Buffer-Local Variables], page 150, vol. 1), whose corresponding variable names have the additional prefix `buffer-` and have underscores replaced with dashes. For instance, `undo_list` stores the value of `buffer-undo-list`.

`mark`
: The mark for the buffer. The mark is a marker, hence it is also included on the list `markers`. See Section 31.7 [The Mark], page 117.

`local_var_alist`
: The association list describing the buffer-local variable bindings of this buffer, not including the built-in buffer-local bindings that have special slots in the buffer object. (Those slots are omitted from this table.) See Section 11.10 [Buffer-Local Variables], page 150, vol. 1.

`major_mode`
: Symbol naming the major mode of this buffer, e.g., `lisp-mode`.

`mode_name`
: Pretty name of the major mode, e.g., `"Lisp"`.

`keymap`
`abbrev_table`
`syntax_table`
`category_table`
`display_table`
: These fields store the buffer's local keymap (see Chapter 22 [Keymaps], page 360, vol. 1), abbrev table (see Section 36.1 [Abbrev Tables], page 250), syntax table (see Chapter 35 [Syntax Tables], page 234), category table (see Section 35.9 [Categories], page 247), and display table (see Section 38.20.2 [Display Tables], page 377).

`downcase_table`
`upcase_table`
`case_canon_table`
: These fields store the conversion tables for converting text to lower case, upper case, and for canonicalizing text for case-fold search. See Section 4.9 [Case Tables], page 61, vol. 1.

`minor_modes`
: An alist of the minor modes of this buffer.

`pt_marker`
`begv_marker`
`zv_marker`
: These fields are only used in an indirect buffer, or in a buffer that is the base of an indirect buffer. Each holds a marker that records `pt`, `begv`, and `zv` respectively, for this buffer when the buffer is not current.

`mode_line_format`
`header_line_format`
`case_fold_search`
`tab_width`
`fill_column`
`left_margin`
`auto_fill_function`
`truncate_lines`
`word_wrap`
`ctl_arrow`
`bidi_display_reordering`
`bidi_paragraph_direction`
`selective_display`
`selective_display_ellipses`
`overwrite_mode`
`abbrev_mode`
`mark_active`
`enable_multibyte_characters`
`buffer_file_coding_system`
`cache_long_line_scans`
`point_before_scroll`
`left_fringe_width`
`right_fringe_width`
`fringes_outside_margins`
`scroll_bar_width`
`indicate_empty_lines`
`indicate_buffer_boundaries`
`fringe_indicator_alist`
`fringe_cursor_alist`
`scroll_up_aggressively`
`scroll_down_aggressively`
`cursor_type`
`cursor_in_non_selected_windows`
: These fields store the values of Lisp variables that are automatically buffer-local (see Section 11.10 [Buffer-Local Variables], page 150, vol. 1), whose corresponding variable names have underscores replaced with dashes. For instance, `mode_line_format` stores the value of `mode-line-format`.

`last_selected_window`
: This is the last window that was selected with this buffer in it, or `nil` if that window no longer displays this buffer.

E.6.2 Window Internals

The fields of a window (for a complete list, see the definition of `struct window` in '`window.h`') include:

`frame` The frame that this window is on.

`mini_p` Non-`nil` if this window is a minibuffer window.

`parent` Internally, Emacs arranges windows in a tree; each group of siblings has a parent window whose area includes all the siblings. This field points to a window's parent.

Parent windows do not display buffers, and play little role in display except to shape their child windows. Emacs Lisp programs usually have no access to the parent windows; they operate on the windows at the leaves of the tree, which actually display buffers.

`hchild`
`vchild` These fields contain the window's leftmost child and its topmost child respectively. `hchild` is used if the window is subdivided horizontally by child windows, and `vchild` if it is subdivided vertically. In a live window, only one of `hchild`, `vchild`, and `buffer` (q.v.) is non-`nil`.

`next`
`prev` The next sibling and previous sibling of this window. `next` is `nil` if the window is the right-most or bottom-most in its group; `prev` is `nil` if it is the left-most or top-most in its group.

`left_col` The left-hand edge of the window, measured in columns, relative to the leftmost column in the frame (column 0).

`top_line` The top edge of the window, measured in lines, relative to the topmost line in the frame (line 0).

`total_cols`
`total_lines`
The width and height of the window, measured in columns and lines respectively. The width includes the scroll bar and fringes, and/or the separator line on the right of the window (if any).

`buffer` The buffer that the window is displaying.

`start` A marker pointing to the position in the buffer that is the first character displayed in the window.

`pointm` This is the value of point in the current buffer when this window is selected; when it is not selected, it retains its previous value.

`force_start`
If this flag is non-`nil`, it says that the window has been scrolled explicitly by the Lisp program. This affects what the next redisplay does if point is off the screen: instead of scrolling the window to show the text around point, it moves point to a location that is on the screen.

`frozen_window_start_p`
: This field is set temporarily to 1 to indicate to redisplay that `start` of this window should not be changed, even if point gets invisible.

`start_at_line_beg`
: Non-`nil` means current value of `start` was the beginning of a line when it was chosen.

`use_time`
: This is the last time that the window was selected. The function `get-lru-window` uses this field.

`sequence_number`
: A unique number assigned to this window when it was created.

`last_modified`
: The `modiff` field of the window's buffer, as of the last time a redisplay completed in this window.

`last_overlay_modified`
: The `overlay_modiff` field of the window's buffer, as of the last time a redisplay completed in this window.

`last_point`
: The buffer's value of point, as of the last time a redisplay completed in this window.

`last_had_star`
: A non-`nil` value means the window's buffer was "modified" when the window was last updated.

`vertical_scroll_bar`
: This window's vertical scroll bar.

`left_margin_cols`
`right_margin_cols`
: The widths of the left and right margins in this window. A value of `nil` means no margin.

`left_fringe_width`
`right_fringe_width`
: The widths of the left and right fringes in this window. A value of `nil` or `t` means use the values of the frame.

`fringes_outside_margins`
: A non-`nil` value means the fringes outside the display margins; othersize they are between the margin and the text.

`window_end_pos`
: This is computed as `z` minus the buffer position of the last glyph in the current matrix of the window. The value is only valid if `window_end_valid` is not `nil`.

`window_end_bytepos`
: The byte position corresponding to `window_end_pos`.

`window_end_vpos`
: The window-relative vertical position of the line containing `window_end_pos`.

`window_end_valid`
: This field is set to a non-`nil` value if `window_end_pos` is truly valid. This is `nil` if nontrivial redisplay is pre-empted, since in that case the display that `window_end_pos` was computed for did not get onto the screen.

`cursor`
: A structure describing where the cursor is in this window.

`last_cursor`
: The value of `cursor` as of the last redisplay that finished.

`phys_cursor`
: A structure describing where the cursor of this window physically is.

`phys_cursor_type`
`phys_cursor_height`
`phys_cursor_width`
: The type, height, and width of the cursor that was last displayed on this window.

`phys_cursor_on_p`
: This field is non-zero if the cursor is physically on.

`cursor_off_p`
: Non-zero means the cursor in this window is logically off. This is used for blinking the cursor.

`last_cursor_off_p`
: This field contains the value of `cursor_off_p` as of the time of the last redisplay.

`must_be_updated_p`
: This is set to 1 during redisplay when this window must be updated.

`hscroll`
: This is the number of columns that the display in the window is scrolled horizontally to the left. Normally, this is 0.

`vscroll`
: Vertical scroll amount, in pixels. Normally, this is 0.

`dedicated`
: Non-`nil` if this window is dedicated to its buffer.

`display_table`
: The window's display table, or `nil` if none is specified for it.

`update_mode_line`
: Non-`nil` means this window's mode line needs to be updated.

`base_line_number`
: The line number of a certain position in the buffer, or `nil`. This is used for displaying the line number of point in the mode line.

`base_line_pos`
: The position in the buffer for which the line number is known, or `nil` meaning none is known. If it is a buffer, don't display the line number as long as the window shows that buffer.

`region_showing`
: If the region (or part of it) is highlighted in this window, this field holds the mark position that made one end of that region. Otherwise, this field is `nil`.

`column_number_displayed`
: The column number currently displayed in this window's mode line, or `nil` if column numbers are not being displayed.

`current_matrix`
`desired_matrix`
: Glyph matrices describing the current and desired display of this window.

E.6.3 Process Internals

The fields of a process (for a complete list, see the definition of `struct Lisp_Process` in '`process.h`') include:

`name` A string, the name of the process.

`command` A list containing the command arguments that were used to start this process. For a network or serial process, it is `nil` if the process is running or `t` if the process is stopped.

`filter` If non-`nil`, a function used to accept output from the process instead of a buffer.

`sentinel` If non-`nil`, a function called whenever the state of the process changes.

`buffer` The associated buffer of the process.

`pid` An integer, the operating system's process ID. Pseudo-processes such as network or serial connections use a value of 0.

`childp` A flag, `t` if this is really a child process. For a network or serial connection, it is a plist based on the arguments to `make-network-process` or `make-serial-process`.

`mark` A marker indicating the position of the end of the last output from this process inserted into the buffer. This is often but not always the end of the buffer.

`kill_without_query`
: If this is non-zero, killing Emacs while this process is still running does not ask for confirmation about killing the process.

`raw_status`
: The raw process status, as returned by the `wait` system call.

`status` The process status, as `process-status` should return it.

`tick`
`update_tick`
: If these two fields are not equal, a change in the status of the process needs to be reported, either by running the sentinel or by inserting a message in the process buffer.

`pty_flag` Non-`nil` if communication with the subprocess uses a PTY; `nil` if it uses a pipe.

`infd` The file descriptor for input from the process.

`outfd` The file descriptor for output to the process.

`tty_name` The name of the terminal that the subprocess is using, or `nil` if it is using pipes.

`decode_coding_system`
Coding-system for decoding the input from this process.

`decoding_buf`
A working buffer for decoding.

`decoding_carryover`
Size of carryover in decoding.

`encode_coding_system`
Coding-system for encoding the output to this process.

`encoding_buf`
A working buffer for encoding.

`inherit_coding_system_flag`
Flag to set `coding-system` of the process buffer from the coding system used to decode process output.

`type` Symbol indicating the type of process: `real`, `network`, `serial`.

Appendix F Standard Errors

Here is a list of the more important error symbols in standard Emacs, grouped by concept. The list includes each symbol's message (on the `error-message` property of the symbol) and a cross reference to a description of how the error can occur.

Each error symbol has an `error-conditions` property that is a list of symbols. Normally this list includes the error symbol itself and the symbol `error`. Occasionally it includes additional symbols, which are intermediate classifications, narrower than `error` but broader than a single error symbol. For example, all the errors in accessing files have the condition `file-error`. If we do not say here that a certain error symbol has additional error conditions, that means it has none.

As a special exception, the error symbol `quit` does not have the condition `error`, because quitting is not considered an error.

Most of these error symbols are defined in C (mainly '`data.c`'), but some are defined in Lisp. For example, the file '`userlock.el`' defines the `file-locked` and `file-supersession` errors. Several of the specialized Lisp libraries distributed with Emacs define their own error symbols. We do not attempt to list of all those here.

See Section 10.5.3 [Errors], page 128, vol. 1, for an explanation of how errors are generated and handled.

`error` `"error"`
See Section 10.5.3 [Errors], page 128, vol. 1.

`quit` `"Quit"`
See Section 21.11 [Quitting], page 351, vol. 1.

`args-out-of-range`
`"Args out of range"`
This happens when trying to access an element beyond the range of a sequence or buffer.
See Chapter 6 [Sequences Arrays Vectors], page 86, vol. 1, See Chapter 32 [Text], page 122.

`arith-error`
`"Arithmetic error"`
See Section 3.6 [Arithmetic Operations], page 39, vol. 1.

`beginning-of-buffer`
`"Beginning of buffer"`
See Section 30.2.1 [Character Motion], page 100.

`buffer-read-only`
`"Buffer is read-only"`
See Section 27.7 [Read Only Buffers], page 9.

`circular-list`
`"List contains a loop"`
This happens when some operations (e.g. resolving face names) encounter circular structures.
See Section 2.5 [Circular Objects], page 26, vol. 1.

`cl-assertion-failed`
: `"Assertion failed"`
This happens when the `assert` macro fails a test.
See Section "Assertions" in *Common Lisp Extensions*.

`coding-system-error`
: `"Invalid coding system"`
See Section 33.9.3 [Lisp and Coding Systems], page 195.

`cyclic-function-indirection`
: `"Symbol's chain of function indirections contains a loop"`
See Section 9.1.4 [Function Indirection], page 112, vol. 1.

`cyclic-variable-indirection`
: `"Symbol's chain of variable indirections contains a loop"`
See Section 11.13 [Variable Aliases], page 160, vol. 1.

`dbus-error`
: `"D-Bus error"`
This is only defined if Emacs was compiled with D-Bus support.
See Section "Errors and Events" in *D-Bus integration in Emacs*.

`end-of-buffer`
: `"End of buffer"`
See Section 30.2.1 [Character Motion], page 100.

`end-of-file`
: `"End of file during parsing"`
Note that this is not a subcategory of `file-error`, because it pertains to the Lisp reader, not to file I/O.
See Section 19.3 [Input Functions], page 276, vol. 1.

`file-already-exists`
: This is a subcategory of `file-error`.
See Section 25.4 [Writing to Files], page 468, vol. 1.

`file-date-error`
: This is a subcategory of `file-error`. It occurs when `copy-file` tries and fails to set the last-modification time of the output file.
See Section 25.7 [Changing Files], page 479, vol. 1.

`file-error`
: We do not list the error-strings of this error and its subcategories, because the error message is normally constructed from the data items alone when the error condition `file-error` is present. Thus, the error-strings are not very relevant. However, these error symbols do have `error-message` properties, and if no data is provided, the `error-message` property *is* used.
See Chapter 25 [Files], page 461, vol. 1.

`compression-error`
: This is a subcategory of `file-error`, which results from problems handling a compressed file.
See Section 15.1 [How Programs Do Loading], page 209, vol. 1.

`file-locked`
: This is a subcategory of `file-error`.
See Section 25.5 [File Locks], page 470, vol. 1.

`file-supersession`
: This is a subcategory of `file-error`.
See Section 27.6 [Modification Time], page 8.

`ftp-error`
: This is a subcategory of `file-error`, which results from problems in accessing a remote file using ftp.
See Section "Remote Files" in *The GNU Emacs Manual*.

`invalid-function`
: `"Invalid function"`
See Section 9.1.4 [Function Indirection], page 112, vol. 1.

`invalid-read-syntax`
: `"Invalid read syntax"`
See Section 2.1 [Printed Representation], page 8, vol. 1.

`invalid-regexp`
: `"Invalid regexp"`
See Section 34.3 [Regular Expressions], page 211.

`mark-inactive`
: `"The mark is not active now"`
See Section 31.7 [The Mark], page 117.

`no-catch` `"No catch for tag"`
See Section 10.5.1 [Catch and Throw], page 126, vol. 1.

`scan-error`
: `"Scan error"`
This happens when certain syntax-parsing functions find invalid syntax or mismatched parentheses.
See Section 30.2.6 [List Motion], page 106, and Section 35.6 [Parsing Expressions], page 242.

`search-failed`
: `"Search failed"`
See Chapter 34 [Searching and Matching], page 209.

`setting-constant`
: `"Attempt to set a constant symbol"`
The values of the symbols `nil` and `t`, and any symbols that start with ':', may not be changed.
See Section 11.2 [Variables that Never Change], page 137, vol. 1.

`text-read-only`
: `"Text is read-only"`
This is a subcategory of `buffer-read-only`.
See Section 32.19.4 [Special Properties], page 162.

`undefined-color`
: `"Undefined color"`
See Section 29.20 [Color Names], page 92.

`void-function`
: `"Symbol's function definition is void"`
See Section 12.8 [Function Cells], page 175, vol. 1.

`void-variable`
: `"Symbol's value as variable is void"`
See Section 11.7 [Accessing Variables], page 144, vol. 1.

`wrong-number-of-arguments`
: `"Wrong number of arguments"`
See Section 9.1.3 [Classifying Lists], page 112, vol. 1.

`wrong-type-argument`
: `"Wrong type argument"`
See Section 2.6 [Type Predicates], page 27, vol. 1.

The following kinds of error, which are classified as special cases of `arith-error`, can occur on certain systems for invalid use of mathematical functions. See Section 3.9 [Math Functions], page 46, vol. 1.

`domain-error`
: `"Arithmetic domain error"`

`overflow-error`
: `"Arithmetic overflow error"`
This is a subcategory of `domain-error`.

`range-error`
: `"Arithmetic range error"`

`singularity-error`
: `"Arithmetic singularity error"`
This is a subcategory of `domain-error`.

`underflow-error`
: `"Arithmetic underflow error"`
This is a subcategory of `domain-error`.

Appendix G Standard Keymaps

In this section we list some of the more general keymaps. Many of these exist when Emacs is first started, but some are loaded only when the respective feature is accessed.

There are many other, more specialized, maps than these; in particular those associated with major and minor modes. The minibuffer uses several keymaps (see Section 20.6.3 [Completion Commands], page 296, vol. 1). For more details on keymaps, see Chapter 22 [Keymaps], page 360, vol. 1.

`2C-mode-map`
: A sparse keymap for subcommands of the prefix `C-x 6`.
See Section "Two-Column Editing" in *The GNU Emacs Manual*.

`abbrev-map`
: A sparse keymap for subcommands of the prefix `C-x a`.
See Section "Defining Abbrevs" in *The GNU Emacs Manual*.

`button-buffer-map`
: A sparse keymap useful for buffers containing buffers.
You may want to use this as a parent keymap. See Section 38.17 [Buttons], page 366.

`button-map`
: A sparse keymap used by buttons.

`ctl-x-4-map`
: A sparse keymap for subcommands of the prefix `C-x 4`.

`ctl-x-5-map`
: A sparse keymap for subcommands of the prefix `C-x 5`.

`ctl-x-map`
: A full keymap for `C-x` commands.

`ctl-x-r-map`
: A sparse keymap for subcommands of the prefix `C-x r`.
See Section "Registers" in *The GNU Emacs Manual*.

`esc-map` A full keymap for `ESC` (or `Meta`) commands.

`facemenu-keymap`
: A sparse keymap used for the `M-o` prefix key.

`function-key-map`
: The parent keymap of all `local-function-key-map` (q.v.) instances.

`global-map`
: The full keymap containing default global key bindings.
Modes should not modify the Global map.

`goto-map` A sparse keymap used for the `M-g` prefix key.

`help-map` A sparse keymap for the keys following the help character `C-h`.
See Section 24.5 [Help Functions], page 457, vol. 1.

`Helper-help-map`
: A full keymap used by the help utility package.
It has the same keymap in its value cell and in its function cell.

`input-decode-map`
: The keymap for translating keypad and function keys.
If there are none, then it contains an empty sparse keymap. See Section 22.14 [Translation Keymaps], page 378, vol. 1.

`key-translation-map`
: A keymap for translating keys. This one overrides ordinary key bindings, unlike `local-function-key-map`. See Section 22.14 [Translation Keymaps], page 378, vol. 1.

`kmacro-keymap`
: A sparse keymap for keys that follows the *C-x C-k* prefix search.
See Section "Keyboard Macros" in *The GNU Emacs Manual*.

`local-function-key-map`
: The keymap for translating key sequences to preferred alternatives.
If there are none, then it contains an empty sparse keymap. See Section 22.14 [Translation Keymaps], page 378, vol. 1.

`menu-bar-file-menu`
`menu-bar-edit-menu`
`menu-bar-options-menu`
`global-buffers-menu-map`
`menu-bar-tools-menu`
`menu-bar-help-menu`
: These keymaps display the main, top-level menus in the menu bar.
Some of them contain sub-menus. For example, the Edit menu contains `menu-bar-search-menu`, etc. See Section 22.17.5 [Menu Bar], page 391, vol. 1.

`minibuffer-inactive-mode-map`
: A full keymap used in the minibuffer when it is not active.
See Section "Editing in the Minibuffer" in *The GNU Emacs Manual*.

`mode-line-coding-system-map`
`mode-line-input-method-map`
`mode-line-column-line-number-mode-map`
: These keymaps control various areas of the mode line.
See Section 23.4 [Mode Line Format], page 419, vol. 1.

`mode-specific-map`
: The keymap for characters following *C-c*. Note, this is in the global map. This map is not actually mode-specific: its name was chosen to be informative in *C-h b* (`display-bindings`), where it describes the main use of the *C-c* prefix key.

`mouse-appearance-menu-map`
: A sparse keymap used for the *S-mouse-1* key.

`mule-keymap`
: The global keymap used for the *C-x RET* prefix key.

`narrow-map`
: A sparse keymap for subcommands of the prefix *C-x n*.

`prog-mode-map`
: The keymap used by Prog mode.
See Section 23.2.5 [Basic Major Modes], page 407, vol. 1.

`query-replace-map`
`multi-query-replace-map`
: A sparse keymap used for responses in `query-replace` and related commands; also for `y-or-n-p` and `map-y-or-n-p`. The functions that use this map do not support prefix keys; they look up one event at a time. `multi-query-replace-map` extends `query-replace-map` for multi-buffer replacements. See Section 34.7 [Search and Replace], page 230.

`search-map`
: A sparse keymap that provides global bindings for search-related commands.

`special-mode-map`
: The keymap used by Special mode.
See Section 23.2.5 [Basic Major Modes], page 407, vol. 1.

`tool-bar-map`
: The keymap defining the contents of the tool bar.
See Section 22.17.6 [Tool Bar], page 392, vol. 1.

`universal-argument-map`
: A sparse keymap used while processing *C-u*.
See Section 21.12 [Prefix Command Arguments], page 353, vol. 1.

`vc-prefix-map`
: The global keymap used for the *C-x v* prefix key.

`x-alternatives-map`
: A sparse keymap used to map certain keys under graphical frames.
The function `x-setup-function-keys` uses this.

Appendix H Standard Hooks

The following is a list of some hook variables that let you provide functions to be called from within Emacs on suitable occasions.

Most of these variables have names ending with '`-hook`'. They are *normal hooks*, run by means of `run-hooks`. The value of such a hook is a list of functions; the functions are called with no arguments and their values are completely ignored. The recommended way to put a new function on such a hook is to call `add-hook`. See Section 23.1 [Hooks], page 396, vol. 1, for more information about using hooks.

The variables whose names end in '`-hooks`' or '`-functions`' are usually *abnormal hooks*; their values are lists of functions, but these functions are called in a special way (they are passed arguments, or their values are used). The variables whose names end in '`-function`' have single functions as their values.

This is not an exhaustive list, it only covers the more general hooks. For example, every major mode defines a hook named '*`modename`*`-mode-hook`'. The major mode command runs this normal hook with `run-mode-hooks` as the very last thing it does. See Section 23.2.6 [Mode Hooks], page 408, vol. 1. Most minor modes have mode hooks too.

A special feature allows you to specify expressions to evaluate if and when a file is loaded (see Section 15.10 [Hooks for Loading], page 221, vol. 1). That feature is not exactly a hook, but does a similar job.

`activate-mark-hook`
`deactivate-mark-hook`
: See Section 31.7 [The Mark], page 117.

`after-change-functions`
`before-change-functions`
`first-change-hook`
: See Section 32.27 [Change Hooks], page 180.

`after-change-major-mode-hook`
`change-major-mode-after-body-hook`
: See Section 23.2.6 [Mode Hooks], page 408, vol. 1.

`after-init-hook`
`before-init-hook`
`emacs-startup-hook`
: See Section 39.1.2 [Init File], page 389.

`after-insert-file-functions`
`write-region-annotate-functions`
`write-region-post-annotation-function`
: See Section 25.12 [Format Conversion], page 497, vol. 1.

`after-make-frame-functions`
`before-make-frame-hook`
: See Section 29.1 [Creating Frames], page 67.

`after-save-hook`
`before-save-hook`
`write-contents-functions`
`write-file-functions`
: See Section 25.2 [Saving Buffers], page 465, vol. 1.

`after-setting-font-hook`
: Hook run after a frame's font changes.

`auto-save-hook`
: See Section 26.2 [Auto-Saving], page 507, vol. 1.

`before-hack-local-variables-hook`
`hack-local-variables-hook`
: See Section 11.11 [File Local Variables], page 156, vol. 1.

`buffer-access-fontify-functions`
: See Section 32.19.7 [Lazy Properties], page 169.

`buffer-list-update-hook`
: Hook run when the buffer list changes.

`buffer-quit-function`
: Function to call to "quit" the current buffer.

`change-major-mode-hook`
: See Section 11.10.2 [Creating Buffer-Local], page 152, vol. 1.

`command-line-functions`
: See Section 39.1.4 [Command-Line Arguments], page 391.

`delayed-warnings-hook`
: The command loop runs this soon after `post-command-hook` (q.v.).

`delete-frame-functions`
: See Section 29.6 [Deleting Frames], page 82.

`delete-terminal-functions`
: See Section 29.2 [Multiple Terminals], page 67.

`display-buffer-function`
`pop-up-frame-function`
`special-display-function`
`split-window-preferred-function`
: See Section 28.13 [Choosing Window Options], page 41.

`echo-area-clear-hook`
: See Section 38.4.4 [Echo Area Customization], page 306.

`find-file-hook`
`find-file-not-found-functions`
: See Section 25.1.1 [Visiting Functions], page 461, vol. 1.

`font-lock-extend-after-change-region-function`
: See Section 23.6.9.2 [Region to Refontify], page 440, vol. 1.

`font-lock-extend-region-functions`
: See Section 23.6.9 [Multiline Font Lock], page 438, vol. 1.

`font-lock-fontify-buffer-function`
`font-lock-fontify-region-function`
`font-lock-mark-block-function`
`font-lock-unfontify-buffer-function`
`font-lock-unfontify-region-function`
: See Section 23.6.4 [Other Font Lock Variables], page 435, vol. 1.

`fontification-functions`
: See Section 38.12.7 [Automatic Face Assignment], page 336.

`frame-auto-hide-function`
: See Section 28.16 [Quitting Windows], page 47.

`kill-buffer-hook`
`kill-buffer-query-functions`
: See Section 27.10 [Killing Buffers], page 13.

`kill-emacs-hook`
`kill-emacs-query-functions`
: See Section 39.2.1 [Killing Emacs], page 392.

`menu-bar-update-hook`
: See Section 22.17.5 [Menu Bar], page 391, vol. 1.

`minibuffer-setup-hook`
`minibuffer-exit-hook`
: See Section 20.14 [Minibuffer Misc], page 313, vol. 1.

`mouse-leave-buffer-hook`
: Hook run when about to switch windows with a mouse command.

`mouse-position-function`
: See Section 29.14 [Mouse Position], page 87.

`post-command-hook`
`pre-command-hook`
: See Section 21.1 [Command Overview], page 315, vol. 1.

`post-gc-hook`
: See Section E.3 [Garbage Collection], page 459.

`post-self-insert-hook`
: See Section 23.3.2 [Keymaps and Minor Modes], page 415, vol. 1.

`suspend-hook`
`suspend-resume-hook`
`suspend-tty-functions`
`resume-tty-functions`
: See Section 39.2.2 [Suspending Emacs], page 393.

`syntax-begin-function`
`syntax-propertize-extend-region-functions`
`syntax-propertize-function`
`font-lock-syntactic-face-function`
: See Section 23.6.8 [Syntactic Font Lock], page 437, vol. 1. See Section 35.4 [Syntax Properties], page 240.

`temp-buffer-setup-hook`
`temp-buffer-show-function`
`temp-buffer-show-hook`
: See Section 38.8 [Temporary Displays], page 313.

`term-setup-hook`
: See Section 39.1.3 [Terminal-Specific], page 390.

`window-configuration-change-hook`
`window-scroll-functions`
`window-size-change-functions`
: See Section 28.25 [Window Hooks], page 64.

`window-setup-hook`
: See Section 38.22 [Window Systems], page 382.

`window-text-change-functions`
: Functions to call in redisplay when text in the window might change.

Index

"

#

$

%

&

'

(

)

*

+

,

-

.

/

;

<

=

>

?

@

[

]

^

`

\

|

1

2

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z