GNU Emacs Lisp Reference Manual

For Emacs Version 27.2

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

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.

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.

The following two functions are useful when a Lisp program needs to map buffer positions to byte offsets in a file visited by the buffer.

`bufferpos-to-filepos` *position* **&optional** *quality coding-system* [Function]
This function is similar to `position-bytes`, but instead of byte position in the current buffer it returns the offset from the beginning of the current buffer's file of the byte that corresponds to the given character *position* in the buffer. The conversion requires to know how the text is encoded in the buffer's file; this is what the *coding-system* argument is for, defaulting to the value of `buffer-file-coding-system`. The optional argument *quality* specifies how accurate the result should be; it should be one of the following:

`exact` The result must be accurate. The function may need to encode and decode a large part of the buffer, which is expensive and can be slow.

`approximate`
The value can be an approximation. The function may avoid expensive processing and return an inexact result.

`nil` If the exact result needs expensive processing, the function will return `nil` rather than an approximation. This is the default if the argument is omitted.

`filepos-to-bufferpos` *byte* **&optional** *quality coding-system* [Function]
This function returns the buffer position corresponding to a file position specified by *byte*, a zero-base byte offset from the file's beginning. The function performs the conversion opposite to what `bufferpos-to-filepos` does. Optional arguments *quality* and *coding-system* have the same meaning and values as for `bufferpos-to-filepos`.

`multibyte-string-p` *string* [Function]
Return `t` if *string* is a multibyte string, `nil` otherwise. This function also returns `nil` if *string* is some object other than a string.

`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 Disabling Multibyte Characters

By default, Emacs starts in multibyte mode: it stores the contents of buffers and strings using an internal encoding that represents non-ASCII characters using multi-byte sequences. Multibyte mode allows you to use all the supported languages and scripts without limitations.

Under very special circumstances, you may want to disable multibyte character support, for a specific buffer. When multibyte characters are disabled in a buffer, we call that *unibyte mode*. In unibyte mode, each character in the buffer has a character code ranging from 0 through 255 (0377 octal); 0 through 127 (0177 octal) represent ASCII characters, and 128 (0200 octal) through 255 (0377 octal) represent non-ASCII characters.

To edit a particular file in unibyte representation, visit it using `find-file-literally`. See Section 25.1.1 [Visiting Functions], page 542. You can convert a multibyte buffer to unibyte by saving it to a file, killing the buffer, and visiting the file again with `find-file-literally`. Alternatively, you can use *C-x RET c* (`universal-coding-system-argument`) and specify '`raw-text`' as the coding system with which to visit or save a file. See Section "Specifying a Coding System for File Text" in *GNU Emacs Manual*. Unlike `find-file-literally`, finding a file as '`raw-text`' doesn't disable format conversion, uncompression, or auto mode selection.

The buffer-local variable `enable-multibyte-characters` is non-`nil` in multibyte buffers, and `nil` in unibyte ones. The mode line also indicates whether a buffer is multibyte or not. With a graphical display, in a multibyte buffer, the portion of the mode line that indicates the character set has a tooltip that (amongst other things) says that the buffer is multibyte. In a unibyte buffer, the character set indicator is absent. Thus, in a unibyte buffer (when using a graphical display) there is normally nothing before the indication of the visited file's end-of-line convention (colon, backslash, etc.), unless you are using an input method.

You can turn off multibyte support in a specific buffer by invoking the command `toggle-enable-multibyte-characters` in that buffer.

33.3 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 872).

`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, *byte*. It signals an error if *byte* 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.

33.4 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.

This function signals an error if the buffer is narrowed, since the narrowing might have occurred in the middle of multibyte character sequences.

This function also signals an error if the buffer is an indirect buffer. An indirect buffer always inherits the representation of its 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.

33.5 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
```

`char-from-name` *string* **&optional** *ignore-case* [Function]

This function returns the character whose Unicode name is *string*. If *ignore-case* is non-`nil`, case is ignored in *string*. This function returns `nil` if *string* does not name a character.

```
;; U+03A3
(= (char-from-name "GREEK CAPITAL LETTER SIGMA") #x03A3)
     ⇒ t
```

`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.

33.6 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 (`https://www.unicode.org/reports/tr23/`), and the Emacs character property database is derived from the Unicode Character Database (UCD). See the Character Properties chapter of the Unicode Standard (`https://www.unicode.org/versions/Unicode12.1.0/ch04.pdf`), 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 `nil`.

`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. 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 39.26 [Bidirectional Display], page 1115). 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 characters that don't have decomposition sequences, and for unassigned codepoints, the value is a list with a single member, the character itself.

`decimal-digit-value`
: Corresponds to the Unicode `Numeric_Value` property for characters whose `Numeric_Type` is '`Decimal`'. The value is an integer, or `nil` if the character

[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`'.

has no decimal digit value. 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 '`Digit`'. The value is an integer. Examples of such characters include compatibility subscript and superscript digits, for which the value is the corresponding number. For characters that don't have any numeric value, and 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 a 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 characters that don't have any numeric value, and 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, 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 39.26 [Bidirectional Display], page 1115). For unassigned codepoints, the value is `nil`.

`paired-bracket`
: Corresponds to the Unicode `Bidi_Paired_Bracket` property. The value of this property is the codepoint of a character's *paired bracket*, or `nil` if the character is not a bracket character. This establishes a mapping between characters that are treated as bracket pairs by the Unicode Bidirectional Algorithm; Emacs uses this property when it decides how to reorder for display parentheses, braces, and other similar characters (see Section 39.26 [Bidirectional Display], page 1115).

`bracket-type`
: Corresponds to the Unicode `Bidi_Paired_Bracket_Type` property. For characters whose `paired-bracket` property is non-`nil`, the value of this property is a symbol, either `o` (for opening bracket characters) or `c` (for closing bracket characters). For characters whose `paired-bracket` property is `nil`, the value is the symbol `n` (None). Like `paired-bracket`, this property is used for bidirectional display.

`old-name` Corresponds to the Unicode `Unicode_1_Name` property. The value is a string. For unassigned codepoints, and characters that have no value for this property, the value is `nil`.

`iso-10646-comment`
Corresponds to the Unicode `ISO_Comment` property. The value is either a string or `nil`. For unassigned codepoints, the value is `nil`.

`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.

`special-uppercase`
Corresponds to Unicode language- and context-independent special upper-casing rules. The value of this property is a string (which may be empty). For example mapping for U+00DF LATIN SMALL LETTER SHARP S is `"SS"`. For characters with no special mapping, the value is `nil` which means `uppercase` property needs to be consulted instead.

`special-lowercase`
Corresponds to Unicode language- and context-independent special lower-casing rules. The value of this property is a string (which may be empty). For example mapping for U+0130 LATIN CAPITAL LETTER I WITH DOT ABOVE the value is `"i\u0307"` (i.e. 2-character string consisting of LATIN SMALL LETTER I followed by U+0307 COMBINING DOT ABOVE). For characters with no special mapping, the value is `nil` which means `lowercase` property needs to be consulted instead.

`special-titlecase`
Corresponds to Unicode unconditional special title-casing rules. The value of this property is a string (which may be empty). For example mapping for U+FB01 LATIN SMALL LIGATURE FI the value is `"Fi"`. For characters with no special mapping, the value is `nil` which means `titlecase` property needs to be consulted instead.

`get-char-code-property` *char propname* [Function]

This function returns the value of *char*'s *propname* property.

```
(get-char-code-property ?\s 'general-category)
     ⇒ Zs
(get-char-code-property ?1 'general-category)
     ⇒ Nd
```

```
;; U+2084
(get-char-code-property ?\N{SUBSCRIPT FOUR}
                        'digit-value)
     ⇒ 4
;; U+2155
(get-char-code-property ?\N{VULGAR FRACTION ONE FIFTH}
                        'numeric-value)
     ⇒ 0.2
;; U+2163
(get-char-code-property ?\N{ROMAN NUMERAL FOUR}
                        'numeric-value)
     ⇒ 4
(get-char-code-property ?\( 'paired-bracket)
     ⇒ 41  ;; closing parenthesis
(get-char-code-property ?\) 'bracket-type)
     ⇒ c
```

`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 114) 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.

33.7 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 function 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 33.10 [Coding Systems], page 885).

`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`.

  For backward compatibility, if *code-point* doesn't fit in a Lisp fixnum (see Section 3.1 [Integer Basics], page 37), 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. This usage is obsolescent.

`encode-char` *char charset* [Function]
: This function returns the code point assigned to the character *char* in *charset*. 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.

33.8 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 33.10 [Coding Systems], page 885) 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 33.9 [Translation of Characters], page 884). 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.

33.9 Translation of Characters

A *translation table* is a char-table (see Section 6.6 [Char-Tables], page 114) 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 33.10.1 [Basic Concepts of Coding Systems], page 885.) 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.

33.10 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.

33.10.1 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. The coding system `prefer-utf-8` is like `undecided`, but it prefers to choose `utf-8` when possible.

In general, a coding system doesn't guarantee roundtrip identity: decoding a byte sequence using a 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. (This was the convention used in Classic Mac OS.)

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 872). 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-unix` (so it forces no conversion of end-of-line, unlike `utf-8-emacs`, which can decode all 3 kinds of end-of-line conventions).

`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*.

33.10.2 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 33.10.6 [Specifying Coding Systems], page 895), or implicitly using a default mechanism (see Section 33.10.5 [Default Coding Systems], page 892). But these methods may not completely specify what to do. For example, they may choose a coding system such as `undecided` 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 33.10.4 [User-Chosen Coding Systems], page 890). 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 33.10.7 [Explicit Encoding], page 896) 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 29.20 [Window System Selections], page 762.

`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.

On Windows 2000 and later, Emacs by default uses Unicode APIs to pass file names to the OS, so the value of `file-name-coding-system` is largely ignored. Lisp applications that need to encode or decode file names on the Lisp level should use `utf-8` coding-system when `system-type` is `windows-nt`; the conversion of UTF-8 encoded file names to the encoding appropriate for communicating with the OS is performed internally by Emacs.

33.10.3 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 11.7.3.1 [Signaling Errors], page 171).

`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 33.10.7 [Explicit Encoding], page 896).

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-nul-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 the encoding of text that contains null bytes to be correctly detected, 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 33.7 [Character Sets], page 882) 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 Emacs characters, the value is `(emacs)`.
- If *coding-system* supports all Unicode characters, the value is `(unicode)`.
- If *coding-system* supports all ISO-2022 charsets, the value is `iso-2022`.
- If *coding-system* supports all the characters in the internal coding system used by Emacs version 21 (prior to the implementation of internal Unicode support), the value is `emacs-mule`.

See [Process Information], page 974, 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.

33.10.4 User-Chosen Coding Systems

`select-safe-coding-system` *from to* **&optional** [Function]
default-coding-system accept-default-p file
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 872), `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. This is handy for checking only the coding systems in the list.

The optional argument *accept-default-p* determines whether a coding system selected without user interaction is acceptable. If it's omitted or `nil`, such a silent selection is always acceptable. If it is non-`nil`, it should be a function; `select-safe-coding-system` calls this function with one argument, the base coding system of the selected coding system. If the function 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.

`select-safe-coding-system-function` [Variable]

This variable names the function to be called to request the user to select a proper coding system for encoding text when the default coding system for an output operation cannot safely encode that text. The default value of this variable is `select-safe-coding-system`. Emacs primitives that write text to files, such as `write-region`, or send text to other processes, such as `process-send-region`, normally call the value

of this variable, unless `coding-system-for-write` is bound to a non-`nil` value (see Section 33.10.6 [Specifying Coding Systems], page 895).

Here are two functions you can use to let the user specify a coding system, with completion. See Section 20.6 [Completion], page 360.

`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 33.10 [Coding Systems], page 885.

33.10.5 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 33.10.6 [Specifying Coding Systems], page 895).

`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 the 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`.

The functions in this list could be called either when the file is visited and Emacs wants to decode its contents, and/or when the file's buffer is about to be saved and Emacs wants to determine how to encode its contents.

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.

33.10.6 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.
(let ((coding-system-for-read 'no-conversion))
  (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. It also applies to encoding command-line arguments with which Emacs invokes subprocesses.

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.

`coding-system-require-warning` [Variable]

Binding `coding-system-for-write` to a non-`nil` value prevents output primitives from calling the function specified by `select-safe-coding-system-function` (see Section 33.10.4 [User-Chosen Coding Systems], page 890). This is because *C-x*

RET c (`universal-coding-system-argument`) works by binding `coding-system-for-write`, and Emacs should obey user selection. If a Lisp program binds `coding-system-for-write` to a value that might not be safe for encoding the text to be written, it can also bind `coding-system-require-warning` to a non-`nil` value, which will force the output primitives to check the encoding by calling the value of `select-safe-coding-system-function` even though `coding-system-for-write` is non-`nil`. Alternatively, call `select-safe-coding-system` explicitly before using the specified encoding.

`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 33.10.7 [Explicit Encoding], page 896).

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 33.10.3 [Lisp and Coding Systems], page 888).

`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 executes *body*, like `progn` does (see Section 11.1 [Sequencing], page 150), 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*.

33.10.7 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 872), 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 548); 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 549), 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 872).

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 33.10.4 [User-Chosen Coding Systems], page 890) 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. If that buffer is a unibyte buffer (see Section 33.4 [Selecting a Representation], page 875), the internal representation of the decoded text (see Section 33.1 [Text Representations], page 872) is inserted into the buffer as individual bytes.

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. If that buffer is a unibyte buffer, the internal representation of the decoded text is inserted into it as individual bytes.

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.

33.10.8 Terminal I/O Encoding

Emacs can use coding systems to decode keyboard input 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` when encoding or decoding terminal I/O.

`keyboard-coding-system` **&optional** *terminal* [Function]

This function returns the coding system used for decoding keyboard input from *terminal*. A value of `no-conversion` means no decoding is done. If *terminal* is omitted or `nil`, it means the selected frame's terminal. See Section 29.2 [Multiple Terminals], page 712.

`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 not to 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 712.

`terminal-coding-system` **&optional** *terminal* [Function]
: This function returns the coding system that is in use for encoding terminal output from *terminal*. A value of `no-conversion` means no encoding is done. 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`, that means not to encode terminal output. If *terminal* is a frame, it means that frame's terminal; if it is `nil`, that means the currently selected frame's terminal.

33.11 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 417, and Section 21.8.4 [Invoking the Input Method], page 421.

33.12 Locales

In POSIX, locales 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 sending batch output to the standard output and error streams, 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`*`)` of 2 integers, for the default paper size measured in millimeters (locale items `_NL_PAPER_WIDTH` and `_NL_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.