The Emacs Widget Library

Version: 1.9914

Introduction

Most graphical user interface toolkits, such as Motif and XView, provide a number of standard user interface controls (sometimes known as 'widgets' or 'gadgets'). Emacs doesn't really support anything like this, except for an incredible powerful text "widget". On the other hand, Emacs does provide the necessary primitives to implement many other widgets within a text buffer. The `widget` package simplifies this task.

The basic widgets are:

`link`
: Areas of text with an associated action. Intended for hypertext links embedded in text.

`push-button`
: Like link, but intended for stand-alone buttons.

`editable-field`
: An editable text field. It can be either variable or fixed length.

`menu-choice`
: Allows the user to choose one of multiple options from a menu, each option is itself a widget. Only the selected option will be visible in the buffer.

`radio-button-choice`
: Allows the user to choose one of multiple options by activating radio buttons. The options are implemented as widgets. All options will be visible in the buffer.

`item`
: A simple constant widget intended to be used in the `menu-choice` and `radio-button-choice` widgets.

`choice-item`
: An button item only intended for use in choices. When invoked, the user will be asked to select another option from the choice widget.

`toggle`
: A simple '`on`'/'`off`' switch.

`checkbox`
: A checkbox ('`[ ]`'/'`[X]`').

`editable-list`
: Create an editable list. The user can insert or delete items in the list. Each list item is itself a widget.

Now of what possible use can support for widgets be in a text editor? I'm glad you asked. The answer is that widgets are useful for implementing forms. A *form* in Emacs is a buffer where the user is supposed to fill out a number of fields, each of which has a specific meaning. The user is not supposed to change or delete any of the text between the fields. Examples of forms in Emacs are the '`forms`' package (of course), the customize buffers, the mail and news compose modes, and the HTML form support in the '`w3`' browser.

The advantages for a programmer of using the `widget` package to implement forms are:

1. More complex field than just editable text are supported.
2. You can give the user immediate feedback if he enters invalid data in a text field, and sometimes prevent entering invalid data.
3. You can have fixed sized fields, thus allowing multiple field to be lined up in columns.
4. It is simple to query or set the value of a field.

5. Editing happens in buffer, not in the mini-buffer.
6. Packages using the library get a uniform look, making them easier for the user to learn.
7. As support for embedded graphics improve, the widget library will extended to support it. This means that your code using the widget library will also use the new graphic features by automatic.

In order to minimize the code that is loaded by users who does not create any widgets, the code has been split in two files:

'`widget.el`'
: This will declare the user variables, define the function `widget-define`, and autoload the function `widget-create`.

'`wid-edit.el`'
: Everything else is here, there is no reason to load it explicitly, as it will be autoloaded when needed.

User Interface

A form consist of read only text for documentation and some fields, where each the fields contain two parts, as tag and a value. The tags are used to identify the fields, so the documentation can refer to the foo field, meaning the field tagged with '`Foo`'. Here is an example form:

```
Here is some documentation.

Name: My Name        Choose: This option
Address:  Some Place
In some City
Some country.

See also _other work_ for more information.

Numbers: count to three below
[INS] [DEL] One
[INS] [DEL] Eh, two?
[INS] [DEL] Five!
[INS]

Select multiple:

[X] This
[ ] That
[X] Thus

Select one:

(*) One
( ) Another One.
( ) A Final One.

[Apply Form] [Reset Form]
```

The top level widgets in is example are tagged '`Name`', '`Choose`', '`Address`', '`_other work_`', '`Numbers`', '`Select multiple`', '`Select one`', '`[Apply Form]`', and '`[Reset Form]`'. There are basically two thing the user can do within a form, namely editing the editable text fields and activating the buttons.

Editable Text Fields

In the example, the value for the '`Name`' is most likely displayed in an editable text field, and so are values for each of the members of the '`Numbers`' list. All the normal Emacs editing operations are available for editing these fields. The only restriction is that each change you make must be contained within a single editable text field. For example, capitalizing all text from the middle of one field to the middle of another field is prohibited.

Editing text fields are created by the `editable-field` widget.

The editing text fields are highlighted with the `widget-field-face` face, making them easy to find.

widget-field-face Face

Face used for other editing fields.

Buttons

Some portions of the buffer have an associated *action*, which can be *invoked* by a standard key or mouse command. These portions are called *buttons*. The default commands for activating a button are:

⟨RET⟩

widget-button-press *pos* &optional *event* Command

Invoke the button at *pos*, defaulting to point. If point is not located on a button, invoke the binding in `widget-global-map` (by default the global map).

mouse-2

widget-button-click *event* Command

Invoke the button at the location of the mouse pointer. If the mouse pointer is located in an editable text field, invoke the binding in `widget-global-map` (by default the global map).

There are several different kind of buttons, all of which are present in the example:

The Option Field Tags.
: When you invoke one of these buttons, you will be asked to choose between a number of different options. This is how you edit an option field. Option fields are created by the `menu-choice` widget. In the example, '**Choose**' is an option field tag.

*The '***[INS]***' and '***[DEL]***' buttons.*
: Activating these will insert or delete elements from a editable list. The list is created by the `editable-list` widget.

Embedded Buttons.
: The '**_other work_**' is an example of an embedded button. Embedded buttons are not associated with a fields, but can serve any purpose, such as implementing hypertext references. They are usually created by the `link` widget.

*The '[ ]' and '***[X]***' buttons.*
: Activating one of these will convert it to the other. This is useful for implementing multiple-choice fields. You can create it wit

The '( )' and '(*)' buttons.
: Only one radio button in a `radio-button-choice` widget can be selected at any time. When you invoke one of the unselected radio buttons, it will be selected and the previous selected radio button will become unselected.

*The '[**Apply Form**]' '[**Reset Form**]' buttons.*
: These are explicit buttons made with the `push-button` widget. The main difference from the `link` widget is that the buttons are will be displayed as GUI buttons when possible. enough.

To make them easier to locate, buttons are emphasized in the buffer.

widget-button-face Face
: Face used for buttons.

widget-mouse-face User Option
: Face used for buttons when the mouse pointer is above it.

Navigation

You can use all the normal Emacs commands to move around in a form buffer, plus you will have these additional commands:

⟨TAB⟩

widget-forward &optional *count* Command
: Move point *count* buttons or editing fields forward.

⟨M-TAB⟩

widget-backward &optional *count* Command
: Move point *count* buttons or editing fields backward.

Programming Example

Here is the code to implement the user interface example (see [User Interface], page 2).

```
(require 'widget)

(eval-when-compile
  (require 'wid-edit))

(defvar widget-example-repeat)

(defun widget-example ()
  "Create the widgets from the Widget manual."
  (interactive)
  (switch-to-buffer "*Widget Example*")
  (kill-all-local-variables)
  (make-local-variable 'widget-example-repeat)
  (let ((inhibit-read-only t))
    (erase-buffer))
  (widget-insert "Here is some documentation.\n\nName: ")
  (widget-create 'editable-field
```

```
:size 13
"My Name")
 (widget-create 'menu-choice
:tag "Choose"
:value "This"
:help-echo "Choose me, please!"
:notify (lambda (widget &rest ignore)
  (message "%s is a good choice!"
   (widget-value widget)))
'(item :tag "This option" :value "This")
'(choice-item "That option")
'(editable-field :menu-tag "No option" "Thus option"))
 (widget-insert "Address: ")
 (widget-create 'editable-field
"Some Place\nIn some City\nSome country.")
 (widget-insert "\nSee also ")
 (widget-create 'link
:notify (lambda (&rest ignore)
  (widget-value-set widget-example-repeat
    '("En" "To" "Tre"))
  (widget-setup))
"other work")
 (widget-insert " for more information.\n\nNumbers: count to three below\n")
 (setq widget-example-repeat
(widget-create 'editable-list
       :entry-format "%i %d %v"
       :notify (lambda (widget &rest ignore)
 (let ((old (widget-get widget
':example-length))
       (new (length (widget-value widget))))
   (unless (eq old new)
     (widget-put widget ':example-length new)
     (message "You can count to %d." new))))
       :value '("One" "Eh, two?" "Five!")
       '(editable-field :value "three")))
 (widget-insert "\n\nSelect multiple:\n\n")
 (widget-create 'checkbox t)
 (widget-insert " This\n")
 (widget-create 'checkbox nil)
 (widget-insert " That\n")
 (widget-create 'checkbox
:notify (lambda (&rest ignore) (message "Tickle"))
t)
 (widget-insert " Thus\n\nSelect one:\n\n")
 (widget-create 'radio-button-choice
:value "One"
:notify (lambda (widget &rest ignore)
  (message "You selected %s"
   (widget-value widget)))
'(item "One") '(item "Another One.") '(item "A Final One."))
 (widget-insert "\n")
```

```
 (widget-create 'push-button
:notify (lambda (&rest ignore)
  (if (= (length (widget-value widget-example-repeat))
 3)
      (message "Congratulation!")
    (error "Three was the count!")))
"Apply Form")
 (widget-insert " ")
 (widget-create 'push-button
:notify (lambda (&rest ignore)
  (widget-example))
"Reset Form")
 (widget-insert "\n")
 (use-local-map widget-keymap)
 (widget-setup))
```

Setting Up the Buffer

Widgets are created with `widget-create`, which returns a *widget* object. This object can be queried and manipulated by other widget functions, until it is deleted with `widget-delete`. After the widgets have been created, `widget-setup` must be called to enable them.

widget-create *type* [*keyword argument*]. . . Function

Create and return a widget of type *type*. The syntax for the *type* argument is described in [Basic Types], page 7.

The keyword arguments can be used to overwrite the keyword arguments that are part of *type*.

widget-delete *widget* Function

Delete *widget* and remove it from the buffer.

widget-setup Function

Setup a buffer to support widgets.

This should be called after creating all the widgets and before allowing the user to edit them.

If you want to insert text outside the widgets in the form, the recommended way to do that is with `widget-insert`.

widget-insert Function

Insert the arguments, either strings or characters, at point. The inserted text will be read only.

There is a standard widget keymap which you might find useful.

widget-keymap Const

A keymap with the global keymap as its parent.
⟨TAB⟩ and *C-*⟨TAB⟩ are bound to `widget-forward` and `widget-backward`, respectively.
⟨RET⟩ and *mouse-2* are bound to `widget-button-press` and `widget-button-`.

widget-global-map Variable

Keymap used by `widget-button-press` and `widget-button-click` when not on a button. By default this is `global-map`.

Basic Types

The syntax of a type specification is given below:

```
NAME ::= (NAME [KEYWORD ARGUMENT]... ARGS)
      |   NAME
```

Where, *name* is a widget name, *keyword* is the name of a property, *argument* is the value of the property, and *args* are interpreted in a widget specific way.

There following keyword arguments that apply to all widgets:

`:value` The initial value for widgets of this type.

`:format` This string will be inserted in the buffer when you create a widget. The following '`%`' escapes are available:

'`%[`'
'`%]`' The text inside will be marked as a button.

By default, the text will be shown in `widget-button-face`, and surrounded by brackets.

widget-button-prefix User Option
String to prefix buttons.

widget-button-suffix User Option
String to suffix buttons.

'`%{`'
'`%}`' The text inside will be displayed with the face specified by `:sample-face`.

'`%v`' This will be replaces with the buffer representation of the widgets value. What this is depends on the widget type.

'`%d`' Insert the string specified by `:doc` here.

'`%h`' Like '`%d`', with the following modifications: If the documentation string is more than one line, it will add a button which will toggle between showing only the first line, and showing the full text. Furthermore, if there is no `:doc` property in the widget, it will instead examine the `:documentation-property` property. If it is a lambda expression, it will be called with the widget's value as an argument, and the result will be used as the documentation text.

'`%t`' Insert the string specified by `:tag` here, or the `princ` representation of the value if there is no tag.

'`%%`' Insert a literal '`%`'.

`:button-face`
Face used to highlight text inside %[%] in the format.

`:button-prefix`
`:button-suffix`
Text around %[%] in the format.

These can be

nil No text is inserted.

a string The string is inserted literally.

a symbol The value of the symbol is expanded according to this table.

`:doc` The string inserted by the '`%d`' escape in the format string.

`:tag` The string inserted by the '`%t`' escape in the format string.

`:tag-glyph`
Name of image to use instead of the string specified by ':tag' on Emacsen that supports it.

`:help-echo`
Message displayed whenever you move to the widget with either `widget-forward` or `widget-backward`.

`:indent` An integer indicating the absolute number of spaces to indent children of this widget.

`:offset` An integer indicating how many extra spaces to add to the widget's grandchildren compared to this widget.

`:extra-offset`
An integer indicating how many extra spaces to add to the widget's children compared to this widget.

`:notify` A function called each time the widget or a nested widget is changed. The function is called with two or three arguments. The first argument is the widget itself, the second argument is the widget that was changed, and the third argument is the event leading to the change, if any.

`:menu-tag`
Tag used in the menu when the widget is used as an option in a `menu-choice` widget.

`:menu-tag-get`
Function used for finding the tag when the widget is used as an option in a `menu-choice` widget. By default, the tag used will be either the `:menu-tag` or `:tag` property if present, or the `princ` representation of the `:value` property if not.

`:match` Should be a function called with two arguments, the widget and a value, and returning non-nil if the widget can represent the specified value.

`:validate`
A function which takes a widget as an argument, and return nil if the widgets current value is valid for the widget. Otherwise, it should return the widget containing the invalid data, and set that widgets `:error` property to a string explaining the error.

The following predefined function can be used:

widget-children-validate *widget* Function

All the `:children` of *widget* must be valid.

`:tab-order`
Specify the order in which widgets are traversed with `widget-forward` or `widget-backward`. This is only partially implemented.

a. Widgets with tabbing order `-1` are ignored.
b. (Unimplemented) When on a widget with tabbing order *n*, go to the next widget in the buffer with tabbing order *n+1* or `nil`, whichever comes first.
c. When on a widget with no tabbing order specified, go to the next widget in the buffer with a positive tabbing order, or `nil`

`:parent` The parent of a nested widget (e.g. a `menu-choice` item or an element of a `editable-list` widget).

`:sibling-args`
: This keyword is only used for members of a `radio-button-choice` or `checklist`. The value should be a list of extra keyword arguments, which will be used when creating the `radio-button` or `checkbox` associated with this item.

widget-glyph-directory User Option
: Directory where glyphs are found. Widget will look here for a file with the same name as specified for the image, with either a '`.xpm`' (if supported) or '`.xbm`' extension.

widget-glyph-enable User Option
: If non-nil, allow glyphs to appear on displays where they are supported.

The `link` Widget

Syntax:

```
TYPE ::= (link [KEYWORD ARGUMENT]...  [ VALUE ])
```

The *value*, if present, is used to initialize the `:value` property. The value should be a string, which will be inserted in the buffer.

By default the link will be shown in brackets.

widget-link-prefix User Option
: String to prefix links.

widget-link-suffix User Option
: String to suffix links.

The `url-link` Widget

Syntax:

```
TYPE ::= (url-link [KEYWORD ARGUMENT]...  URL)
```

When this link is invoked, the WWW browser specified by `browse-url-browser-function` will be called with *url*.

The `info-link` Widget

Syntax:

```
TYPE ::= (info-link [KEYWORD ARGUMENT]...  ADDRESS)
```

When this link is invoked, the build-in info browser is started on *address*.

The `push-button` Widget

Syntax:

```
TYPE ::= (push-button [KEYWORD ARGUMENT]...  [ VALUE ])
```

The *value*, if present, is used to initialize the `:value` property. The value should be a string, which will be inserted in the buffer.

By default the tag will be shown in brackets.

widget-push-button-prefix User Option
: String to prefix push buttons.

widget-push-button-suffix User Option
: String to suffix push buttons.

The `editable-field` Widget

Syntax:

```
TYPE ::= (editable-field [KEYWORD ARGUMENT]... [ VALUE ])
```

The *value*, if present, is used to initialize the `:value` property. The value should be a string, which will be inserted in field. This widget will match all string values.

The following extra properties are recognized.

`:size` The width of the editable field.
By default the field will reach to the end of the line.

`:value-face`
Face used for highlighting the editable field. Default is `widget-field-face`.

`:secret` Character used to display the value. You can set this to e.g. `?*` if the field contains a password or other secret information. By default, the value is not secret.

`:valid-regexp`
By default the `:validate` function will match the content of the field with the value of this attribute. The default value is `""` which matches everything.

`:keymap` Keymap used in the editable field. The default value is `widget-field-keymap`, which allows you to use all the normal editing commands, even if the buffers major mode suppress some of them. Pressing return invokes the function specified by `:action`.

The `text` Widget

This is just like `editable-field`, but intended for multiline text fields. The default `:keymap` is `widget-text-keymap`, which does not rebind the return key.

The `menu-choice` Widget

Syntax:

```
TYPE ::= (menu-choice [KEYWORD ARGUMENT]... TYPE ... )
```

The *type* arguments represents each possible choice. The widgets value of will be the value of the chosen *type* argument. This widget will match any value that matches at least one of the specified *type* arguments.

`:void` Widget type used as a fallback when the value does not match any of the specified *type* arguments.

`:case-fold`
Set this to nil if you don't want to ignore case when prompting for a choice through the minibuffer.

`:children`
A list whose car is the widget representing the currently chosen type in the buffer.

`:choice` The current chosen type

`:args` The list of types.

The `radio-button-choice` Widget

Syntax:

```
TYPE ::= (radio-button-choice [KEYWORD ARGUMENT]...  TYPE ... )
```

The *type* arguments represents each possible choice. The widgets value of will be the value of the chosen *type* argument. This widget will match any value that matches at least one of the specified *type* arguments.

The following extra properties are recognized.

`:entry-format`
: This string will be inserted for each entry in the list. The following '`%`' escapes are available:

  '`%v`' Replaced with the buffer representation of the *type* widget.

  '`%b`' Replace with the radio button.

  '`%%`' Insert a literal '`%`'.

`button-args`
: A list of keywords to pass to the radio buttons. Useful for setting e.g. the '`:help-echo`' for each button.

`:buttons` The widgets representing the radio buttons.

`:children`
: The widgets representing each type.

`:choice` The current chosen type

`:args` The list of types.

You can add extra radio button items to a `radio-button-choice` widget after it has been created with the function `widget-radio-add-item`.

widget-radio-add-item *widget type* Function
: Add to `radio-button-choice` widget *widget* a new radio button item of type *type*.

Please note that such items added after the `radio-button-choice` widget has been created will **not** be properly destructed when you call `widget-delete`.

The `item` Widget

Syntax:

```
ITEM ::= (item [KEYWORD ARGUMENT]... VALUE)
```

The *value*, if present, is used to initialize the `:value` property. The value should be a string, which will be inserted in the buffer. This widget will only match the specified value.

The `choice-item` Widget

Syntax:

```
ITEM ::= (choice-item [KEYWORD ARGUMENT]... VALUE)
```

The *value*, if present, is used to initialize the `:value` property. The value should be a string, which will be inserted in the buffer as a button. Activating the button of a `choice-item` is equivalent to activating the parent widget. This widget will only match the specified value.

The `toggle` Widget

Syntax:

```
TYPE ::= (toggle [KEYWORD ARGUMENT]...)
```

The widget has two possible states, 'on' and 'off', which corresponds to a `t` or `nil` value.

The following extra properties are recognized.

`:on` String representing the 'on' state. By default the string '`on`'.

`:off` String representing the 'off' state. By default the string '`off`'.

`:on-glyph`
: Name of a glyph to be used instead of the ':on' text string, on emacsen that supports it.

`:off-glyph`
: Name of a glyph to be used instead of the ':off' text string, on emacsen that supports it.

The `checkbox` Widget

The widget has two possible states, 'selected' and 'unselected', which corresponds to a `t` or `nil` value.

Syntax:

```
TYPE ::= (checkbox [KEYWORD ARGUMENT]...)
```

The `checklist` Widget

Syntax:

```
TYPE ::= (checklist [KEYWORD ARGUMENT]...  TYPE ... )
```

The *type* arguments represents each checklist item. The widgets value of will be a list containing the value of each ticked *type* argument. The checklist widget will match a list whose elements all matches at least one of the specified *type* arguments.

The following extra properties are recognized.

`:entry-format`
: This string will be inserted for each entry in the list. The following '`%`' escapes are available:

  '`%v`' Replaced with the buffer representation of the *type* widget.

  '`%b`' Replace with the checkbox.

  '`%%`' Insert a literal '`%`'.

`:greedy` Usually, a checklist will only match if the items are in the exact sequence given in the specification. By setting `:greedy` to non-nil, it will allow the items to come in any sequence. However, if you extract the value they will be in the sequence given in the checklist. I.e. the original sequence is forgotten.

`button-args`
: A list of keywords to pass to the checkboxes. Useful for setting e.g. the '`:help-echo`' for each checkbox.

`:buttons` The widgets representing the checkboxes.

`:children`
: The widgets representing each type.

`:args` The list of types.

The `editable-list` Widget

Syntax:

```
TYPE ::= (editable-list [KEYWORD ARGUMENT]... TYPE)
```

The value is a list, where each member represents one widget of type *type*.

The following extra properties are recognized.

`:entry-format`
: This string will be inserted for each entry in the list. The following '`%`' escapes are available:

  '`%v`' This will be replaced with the buffer representation of the *type* widget.

  '`%i`' Insert the [**INS**] button.

  '`%d`' Insert the [**DEL**] button.

  '`%%`' Insert a literal '`%`'.

`:insert-button-args`
: A list of keyword arguments to pass to the insert buttons.

`:delete-button-args`
: A list of keyword arguments to pass to the delete buttons.

`:append-button-args`
: A list of keyword arguments to pass to the trailing insert button.

`:buttons` The widgets representing the insert and delete buttons.

`:children`
: The widgets representing the elements of the list.

`:args` List whose car is the type of the list elements.

The `group` Widget

This widget simply group other widget together.

Syntax:

```
TYPE ::= (group [KEYWORD ARGUMENT]... TYPE...)
```

The value is a list, with one member for each *type*.

Sexp Types

A number of widgets for editing s-expressions (lisp types) are also available. These basically fall in the following categories.

The Constant Widgets.

The `const` widget can contain any lisp expression, but the user is prohibited from editing edit it, which is mainly useful as a component of one of the composite widgets.

The syntax for the `const` widget is

```
TYPE ::= (const [KEYWORD ARGUMENT]...  [ VALUE ])
```

The *value*, if present, is used to initialize the `:value` property and can be any s-expression.

const Widget

This will display any valid s-expression in an immutable part of the buffer.

There are two variations of the `const` widget, namely `variable-item` and `function-item`. These should contain a symbol with a variable or function binding. The major difference from the `const` widget is that they will allow the user to see the variable or function documentation for the symbol.

variable-item Widget

An immutable symbol that is bound as a variable.

function-item Widget

An immutable symbol that is bound as a function.

Generic Sexp Widget.

The `sexp` widget can contain any lisp expression, and allows the user to edit it inline in the buffer.

The syntax for the `sexp` widget is

```
TYPE ::= (sexp [KEYWORD ARGUMENT]...  [ VALUE ])
```

sexp Widget

This will allow you to edit any valid s-expression in an editable buffer field.

The `sexp` widget takes the same keyword arguments as the `editable-field` widget.

Atomic Sexp Widgets.

The atoms are s-expressions that does not consist of other s-expressions. A string is an atom, while a list is a composite type. You can edit the value of an atom with the following widgets.

The syntax for all the atoms are

```
TYPE ::= (NAME [KEYWORD ARGUMENT]...  [ VALUE ])
```

The *value*, if present, is used to initialize the `:value` property and must be an expression of the same type as the widget. I.e. the string widget can only be initialized with a string.

All the atom widgets take the same keyword arguments as the `editable-field` widget.

string Widget

Allows you to edit a string in an editable field.

regexp Widget

Allows you to edit a regular expression in an editable field.

character Widget

Allows you to enter a character in an editable field.

file Widget

Allows you to edit a file name in an editable field. If you invoke the tag button, you can edit the file name in the mini-buffer with completion.

Keywords:

`:must-match`
: If this is set to non-nil, only existing file names will be allowed in the mini-buffer.

directory Widget
: Allows you to edit a directory name in an editable field. Similar to the `file` widget.

symbol Widget
: Allows you to edit a lisp symbol in an editable field.

function Widget
: Allows you to edit a lambda expression, or a function name with completion.

variable Widget
: Allows you to edit a variable name, with completion.

integer Widget
: Allows you to edit an integer in an editable field.

number Widget
: Allows you to edit a number in an editable field.

boolean Widget
: Allows you to edit a boolean. In lisp this means a variable which is either nil meaning false, or non-nil meaning true.

Composite Sexp Widgets.

The syntax for the composite are

```
TYPE ::= (NAME [KEYWORD ARGUMENT]...  COMPONENT...)
```

Where each *component* must be a widget type. Each component widget will be displayed in the buffer, and be editable to the user.

cons Widget
: The value of a `cons` widget is a cons-cell where the car is the value of the first component and the cdr is the value of the second component. There must be exactly two components.

list Widget
: The value of a `list` widget is a list containing the value of each of its component.

vector Widget
: The value of a `vector` widget is a vector containing the value of each of its component.

The above suffice for specifying fixed size lists and vectors. To get variable length lists and vectors, you can use a `choice`, `set` or `repeat` widgets together with the `:inline` keywords. If any component of a composite widget has the `:inline` keyword set, its value must be a list which will then be spliced into the composite. For example, to specify a list whose first element must be a file name, and whose remaining arguments should either by the symbol `t` or two files, you can use the following widget specification:

```
(list file
      (choice (const t)
              (list :inline t
                    :value ("foo" "bar")
                    string string)))
```

The value of a widget of this type will either have the form '`(file t)`' or `(file string string)`.

This concept of inline is probably hard to understand. It was certainly hard to implement so instead of confuse you more by trying to explain it here, I'll just suggest you meditate over it for a while.

choice Widget

Allows you to edit a sexp which may have one of fixed set of types. It is currently implemented with the `choice-menu` basic widget, and has a similar syntax.

set Widget

Allows you to specify a type which must be a list whose elements all belong to given set. The elements of the list is not significant. This is implemented on top of the `checklist` basic widget, and has a similar syntax.

repeat Widget

Allows you to specify a variable length list whose members are all of the same type. Implemented on top of the 'editable-list' basic widget, and has a similar syntax.

Properties

You can examine or set the value of a widget by using the widget object that was returned by `widget-create`.

widget-value *widget* Function

Return the current value contained in *widget*. It is an error to call this function on an uninitialized widget.

widget-value-set *widget value* Function

Set the value contained in *widget* to *value*. It is an error to call this function with an invalid *value*.

Important: You *must* call `widget-setup` after modifying the value of a widget before the user is allowed to edit the widget again. It is enough to call `widget-setup` once if you modify multiple widgets. This is currently only necessary if the widget contains an editing field, but may be necessary for other widgets in the future.

If your application needs to associate some information with the widget objects, for example a reference to the item being edited, it can be done with `widget-put` and `widget-get`. The property names must begin with a '`:`'.

widget-put *widget property value* Function

In *widget* set *property* to *value*. *property* should be a symbol, while *value* can be anything.

widget-get *widget property* Function

In *widget* return the value for *property*. *property* should be a symbol, the value is what was last set by `widget-put` for *property*.

widget-member *widget property* Function

Non-nil if *widget* has a value (even nil) for property *property*.

Occasionally it can be useful to know which kind of widget you have, i.e. the name of the widget type you gave when the widget was created.

widget-type *widget* Function

Return the name of *widget*, a symbol.

Widgets can be in two states: active, which means they are modifiable by the user, or inactive, which means they cannot be modified by the user. You can query or set the state with the following code:

```
;; Examine if widget is active or not.
(if (widget-apply widget :active)
    (message "Widget is active.")
  (message "Widget is inactive.")

;; Make widget inactive.
(widget-apply widget :deactivate)

;; Make widget active.
(widget-apply widget :activate)
```

A widget is inactive if itself, or any of its ancestors (found by following the `:parent` link) have been deactivated. To make sure a widget is really active, you must therefore activate both itself, and all its ancestors.

```
(while widget
  (widget-apply widget :activate)
  (setq widget (widget-get widget :parent)))
```

You can check if a widget has been made inactive by examining the value of `:inactive` keyword. If this is non-nil, the widget itself has been deactivated. This is different from using the `:active` keyword, in that the later tell you if the widget **or** any of its ancestors have been deactivated. Do not attempt to set the `:inactive` keyword directly. Use the `:activate` `:deactivated` keywords instead.

Defining New Widgets

You can define specialized widgets with `define-widget`. It allows you to create a shorthand for more complex widgets, including specifying component widgets and default new default values for the keyword arguments.

widget-define *name class doc* &rest *args* Function

Define a new widget type named *name* from `class`.

name and class should both be symbols, `class` should be one of the existing widget types.

The third argument *DOC* is a documentation string for the widget.

After the new widget has been defined, the following two calls will create identical widgets:

- ```
  (widget-create name)
  ```
- ```
  (apply widget-create class args)
  ```

Using `widget-define` does just store the definition of the widget type in the `widget-type` property of *name*, which is what `widget-create` uses.

If you just want to specify defaults for keywords with no complex conversions, you can use `identity` as your conversion function.

The following additional keyword arguments are useful when defining new widgets:

`:convert-widget`
: Function to convert a widget type before creating a widget of that type. It takes a widget type as an argument, and returns the converted widget type. When a

widget is created, this function is called for the widget type and all the widgets parent types, most derived first.

The following predefined functions can be used here:

widget-types-convert-widget *widget* Function
Convert `:args` as widget types in *widget*.

widget-value-convert-widget *widget* Function
Initialize `:value` from `:args` in *widget*.

`:value-to-internal`
Function to convert the value to the internal format. The function takes two arguments, a widget and an external value, and returns the internal value. The function is called on the present `:value` when the widget is created, and on any value set later with `widget-value-set`.

`:value-to-external`
Function to convert the value to the external format. The function takes two arguments, a widget and an internal value, and returns the internal value. The function is called on the present `:value` when the widget is created, and on any value set later with `widget-value-set`.

`:create` Function to create a widget from scratch. The function takes one argument, a widget type, and create a widget of that type, insert it in the buffer, and return a widget object.

`:delete` Function to delete a widget. The function takes one argument, a widget, and should remove all traces of the widget from the buffer.

`:value-create`
Function to expand the '`%v`' escape in the format string. It will be called with the widget as its argument. Should insert a representation of the widgets value in the buffer.

`:value-delete`
Should remove the representation of the widgets value from the buffer. It will be called with the widget as its argument. It doesn't have to remove the text, but it should release markers and delete nested widgets if such has been used.

The following predefined function can be used here:

widget-children-value-delete *widget* Function
Delete all `:children` and `:buttons` in *widget*.

`:value-get`
Function to extract the value of a widget, as it is displayed in the buffer.

The following predefined function can be used here:

widget-value-value-get *widget* Function
Return the `:value` property of *widget*.

`:format-handler`
Function to handle unknown '`%`' escapes in the format string. It will be called with the widget and the escape character as arguments. You can set this to allow your widget to handle non-standard escapes.

You should end up calling `widget-default-format-handler` to handle unknown escape sequences, which will handle the '`%h`' and any future escape sequences, as well as give an error for unknown escapes.

`:action` Function to handle user initiated events. By default, `:notify` the parent.

The following predefined function can be used here:

widget-parent-action *widget* &optional *event* Function

Tell `:parent` of *widget* to handle the `:action`. Optional *event* is the event that triggered the action.

`:prompt-value`

Function to prompt for a value in the minibuffer. The function should take four arguments, *widget*, *prompt*, *value*, and *unbound* and should return a value for widget entered by the user. *prompt* is the prompt to use. *value* is the default value to use, unless *unbound* is non-nil in which case there are no default value. The function should read the value using the method most natural for this widget, and does not have to check that it matches.

If you want to define a new widget from scratch, use the `default` widget as its base.

default Widget

Widget used as a base for other widgets.

It provides most of the functionality that is referred to as "by default" in this text.

Widget Browser

There is a separate package to browse widgets. This is intended to help programmers who want to examine the content of a widget. The browser shows the value of each keyword, but uses links for certain keywords such as ':parent', which avoids printing cyclic structures.

widget-browse *WIDGET* Command

Create a widget browser for WIDGET. When called interactively, prompt for WIDGET.

widget-browse-other-window *WIDGET* Command

Create a widget browser for WIDGET and show it in another window. When called interactively, prompt for WIDGET.

widget-browse-at *POS* Command

Create a widget browser for the widget at POS. When called interactively, use the position of point.

Widget Minor Mode

There is a minor mode for manipulating widgets in major modes that doesn't provide any support for widgets themselves. This is mostly intended to be useful for programmers doing experiments.

widget-minor-mode Command

Toggle minor mode for traversing widgets. With arg, turn widget mode on if and only if arg is positive.

widget-minor-mode-keymap Variable

Keymap used in `widget-minor-mode`.

Utilities.

widget-prompt-value *widget prompt* [*value unbound*] Function
Prompt for a value matching *widget*, using *prompt*. The current value is assumed to be *value*, unless *unbound* is non-nil.

widget-get-sibling *widget* Function
Get the item *widget* is assumed to toggle. This is only meaningful for radio buttons or checkboxes in a list.

Wishlist

- It should be possible to add or remove items from a list with *C-k* and *C-o* (suggested by RMS).
- The '[INS]' and '[DEL]' buttons should be replaced by a single dash ('-'). The dash should be a button that, when invoked, ask whether you want to add or delete an item (RMS wanted to git rid of the ugly buttons, the dash is my idea).
- The `menu-choice` tag should be prettier, something like the abbreviated menus in Open Look.
- Finish `:tab-order`.
- Make indentation work with glyphs and proportional fonts.
- Add commands to show overview of object and class hierarchies to the browser.
- Find a way to disable mouse highlight for inactive widgets.
- Find a way to make glyphs look inactive.
- Add `property-list` widget.
- Add `association-list` widget.
- Add `key-binding` widget.
- Add `widget` widget for editing widget specifications.
- Find clean way to implement variable length list. See `TeX-printer-list` for an explanation.
- *C-h* in `widget-prompt-value` should give type specific help.
- A mailto widget.

Table of Contents

The Emacs Widget Library 1
- Introduction 1
- User Interface 2
  - Editable Text Fields 3
  - Buttons 3
  - Navigation 4
- Programming Example 4
- Setting Up the Buffer 6
- Basic Types 7
  - The `link` Widget 9
  - The `url-link` Widget 9
  - The `info-link` Widget 9
  - The `push-button` Widget 9
  - The `editable-field` Widget 10
  - The `text` Widget 10
  - The `menu-choice` Widget 10
  - The `radio-button-choice` Widget 11
  - The `item` Widget 11
  - The `choice-item` Widget 11
  - The `toggle` Widget 12
  - The `checkbox` Widget 12
  - The `checklist` Widget 12
  - The `editable-list` Widget 13
  - The `group` Widget 13
- Sexp Types 13
  - The Constant Widgets. 13
  - Generic Sexp Widget. 14
  - Atomic Sexp Widgets. 14
  - Composite Sexp Widgets. 15
- Properties 16
- Defining New Widgets 17
- Widget Browser 19
- Widget Minor Mode 19
- Utilities. 20
- Wishlist 20