LispWorks® for Macintosh

Editor User Guide

Version 7.1

Contents

Copyright and Trademarks

LispWorks Editor User Guide (Macintosh version)

Version 7.1

August 2017

Address

LispWorks Ltd
St. John's Innovation Centre
Cowley Road
Cambridge
CB4 0WS
England

Telephone

From North America: 877 759 8839
(toll-free)

From elsewhere: +44 1223 421860

Fax

From North America: 617 812 8283

From elsewhere: +44 870 2206189

www.lispworks.com

1 Introduction

The LispWorks editor is built in the spirit of Emacs. As a matter of policy, the key bindings and the behavior of the LispWorks editor are designed to be as close as possible to the standard key bindings and behavior of GNU Emacs.

For users more familiar with Mac OS keys, an alternate keys and behavior model is provided. This manual however, generally documents the Emacs model.

The LispWorks editor has the following features:

- It is a *screen* editor. This means that text is displayed by the screenful, with a screen normally displaying the text which is currently being edited.
- It is a *real-time* editor. This means that modifications made to text are shown immediately, and any commands issued are executed likewise.
- An *on-line help* facility is provided, which allows the user quick and easy access to command and variable definitions. Various levels of help are provided, depending on the type of information the user currently possesses.
- It is *customizable*. The editor can be customized both for the duration of an editing session, and on a more permanent basis.
- A range of commands are provided to facilitate the editing of Lisp programs.

- The editor is itself written in Lisp.

1.1 Using the editor within LispWorks

The LispWorks editor is fully integrated into the LispWorks programming environment. If you do not currently have an Editor (check the **Window** menu), start one by choosing **Window > Tools > Editor** or clicking on in the LispWorks toolbar.

There are a number of editor operations which are only available in Listener windows (for example, operations using the command history). These operations are covered in the *LispWorks IDE User Guide*.

1.2 About this manual

The *LispWorks Editor User Guide* is divided into chapters, as follows:

Chapter 2, "General Concepts", provides a brief overview of terms and concepts which the user should be familiar with before progressing to the rest of the manual. The section 'Basic editing commands' provides a brief description of commands necessary to edit a file from start to finish. If you are already familiar with Emacs, you should be familiar with most of the information contained in this chapter.

Chapter 3, "Command Reference", contains full details of most of the editor commands. Details of editor variables are also provided where necessary. Not included in this chapter are commands used to facilitate the editing of Lisp programs.

Chapter 4, "Editing Lisp Programs", contains full details of editor commands (and variables where necessary) to allow for easier editing of Lisp programs.

Chapter 5, "Emulation", describes use of Mac OS style key bindings rather than Emacs style.

Chapter 6, "Advanced Features", provides information on customizing and programming the editor. The features described in this chapter allow permanent changes to be made to the editor.

Chapter 7, "Self-contained examples", enumerates the example files which are relevant to the content of this manual and are available in the LispWorks library.

A “Glossary” is also included to provide a quick and easy reference to editor terms and concepts.

Each editor command, variable and function is fully described once in a relevant section (for example, the command `Save File` is described in “File handling” on page 27). It is often worthwhile reading the introductory text at the start of the section, as some useful information is often provided there. The descriptions all follow the same layout convention which should be self-explanatory.

Command description layouts include the name of the command, the default key binding (in Emacs editor emulation unless stated otherwise), details of optional arguments required by the associated defining function (if any) and the mode in which the command can be run (if not global).

1.3 Viewing example files

This manual sometimes refers to example files in the LispWorks library via a Lisp form like this:

```
(example-edit-file "editor/commands/space-show-arglist")
```

These examples are Lisp source files in your LispWorks installation under `lib/7-1-0-0/examples/`. You can simply evaluate the given form to view the example source file.

Example files contain instructions about how to use them at the start of the file.

The examples files are in a read-only directory and therefore you should compile them inside the IDE (by the Editor command `Compile Buffer` or the toolbar button or by choosing **Buffer > Compile** from the context menu), so it does not try to write a fasl file.

If you want to manipulate an example file or compile it on the disk rather than in the IDE, then you need first to copy the file elsewhere (most easily by using the Editor command `Write File` or by choosing **File > Save As** from the context menu).

2

General Concepts

There are a number of terms used throughout this manual which the user should be familiar with. Definitions of these terms are provided in this chapter, along with a section containing just enough information to be able to edit a document from start to finish.

This chapter is not designed to provide precise details of commands. For these see the relevant sections in the following chapters.

2.1 Window layout

2.1.1 Windows and panes

When the editor is called up an editor *window* is created and displayed (for those already familiar with Emacs running on a tty terminal, note that in this context a window is an object used by the window manager to display data, and not a term used to describe a portion of the editor display). The largest area of the editor window is taken up by an editor *pane*. Each window contains a single pane and therefore the term *window* is used throughout this manual as being synonymous with pane, unless more clarification is required.

Initially only one editor window is displayed. The corresponding editor pane is either blank (ready for text to be entered) or contains text from a file to be edited. The editor window displays text using the font associated with the editor pane.

2.1.2 Files and buffers

It is not technically correct to say that a window displays the contents of a *file*, rather that each window displays the contents of a *buffer*. A buffer is an object that contains data from the point of view of the editor, whereas a file contains data from the point of view of the operating system. A buffer is a temporary storage area used by the editor to hold the contents of a file while the process of editing is taking place. When editing has finished the contents of the buffer can then be written to the appropriate file. When the user exits from the editor, no information concerning buffers or windows is saved.

A buffer is often displayed in its own window, although it is also possible for many buffers to be associated with a single window, and for a single buffer to be displayed in more than one window.

In most cases, there is one buffer for each file that is accessed, but sometimes there is more than one buffer for a single file. There are also some buffers (such as the Echo Area, which is used to communicate with the user) that are not necessarily associated with any file.

2.1.3 The mode line

At the bottom of each editor window is a mode line that provides information concerning the buffer which that window is displaying. The contents of the mode line are as follows:

- "LATIN-1" or "MACOS-ROMAN" or "UNICODE", or other encoding name, indicating the encoding of any file associated with the buffer.
- "----" or "-**-" or "-%%-": the first indicates that the buffer is unchanged since it was last saved; the second that it has been changed; and the third that it is read only.
- the *name of the buffer* (the name of a buffer originating from a file is usually the same as the name of that file).
- the *package* of the current buffer written within braces.

- a *major mode* (such as Fundamental or Lisp). A buffer always operates in a single major mode.
- a *minor mode* (such as Abbrev or Auto-Fill). If no minor mode is in operation then this element is omitted from the mode line. An editor can operate in any number of minor modes.
- a *position indicator* showing the line numbers of the topmost and bottommost lines displayed in the window, and the total number of lines in the buffer. The editor can be changed to count characters rather than lines, and then displays percentages rather than line numbers.
- the *pathname* with which the buffer is associated.

2.2 Buffer positions: points, marks and locations

2.2.1 Points

A *point* is a position in a buffer where editor commands take effect. The *current point* is generally between the character indicated by the cursor and the previous character (that is, it actually lies *between* two characters). Many types of commands (that is, moving, inserting, deleting) operate with respect to the current point, and indeed move that point.

Each buffer has a current point associated with it. A buffer that is not being displayed remembers where its current point is and returns the user to that point when the buffer is redisplayed.

If the same buffer is being displayed in more than one window, there is a point associated with the buffer for each window. These points are independent of each other.

2.2.2 Marks

The position of a point can be saved for later reference by setting a *mark*. Marks may either be set explicitly or as side effects of commands. More than one mark may be associated with a single buffer and saved in what is known as a *mark ring*. As for points, the positions of marks in a buffer are remembered even if that buffer is not currently being displayed.

2.2.3 Regions

A *region* is the area of text between the mark and the current point. Many editor commands affect only a specified region.

2.2.4 Locations

A *location* is the position of the current point in a buffer at some time in the past. Locations are recorded automatically by commands that take you to a different buffer or where you might lose your place within the current buffer. They are designed to be a more comprehensive form of the mark ring but without the interaction with the selected region.

2.3 Modes

Each buffer can be in two kinds of *mode*: a *major mode,* such as Lisp mode, or Fundamental mode (which is the ordinary text processing mode); and a *minor mode,* such as Abbrev mode or Auto-Fill mode. A buffer always has precisely one major mode associated with it, but minor modes are optional. Any number of minor modes can be associated with a buffer.

The major modes govern how certain commands behave. For example, the concept of indentation is radically different between Lisp mode and Fundamental mode. As another example, a Directory mode buffer (which is essentially read-only) lists files and allows you to operate on them with simple keystrokes like **`E`** for edit and **`D`** for delete. The file listing is updated automatically to reflect any changes.

When a file is loaded into a new buffer, the default mode of that buffer can be determined by the file name. For example, a buffer into which a file name that has a **`.lisp`** suffix is loaded defaults to Lisp mode.

The minor modes determine whether or not certain actions take place. For example, when Auto-Fill mode is on, lines are automatically broken at the right hand margin, as the text is being typed, when the line length exceeds a pre-defined limit. Normally the newline has to be entered manually at the end of each line.

2.4 Text handling concepts

2.4.1 Words

A *word* is defined as a continuous string of alphanumeric characters. These are the letters A-Z, a-z, numbers 0-9, and the Latin-1 alphanumeric characters). In most modes, any character which is not alphanumeric is treated as a word delimiter.

2.4.2 Sentences

A *sentence* begins wherever a paragraph or previous sentence ends. The end of a sentence is defined as consisting of a sentence terminating character followed by two spaces or a newline. *Two* spaces are required to prevent abbreviations (such as Mr.) from being taken as the end of a sentence. Such abbreviations at the end of a line are taken as the end of a sentence. There may also be any number of closing delimiter characters between the sentence terminating character and the spaces or newline.

Sentence terminating characters include: `. ? !`

Closing delimiter characters include: `) ] > / | " '`

2.4.3 Paragraphs

A *paragraph* is defined as the text within two paragraph delimiters. A blank line constitutes a paragraph delimiter. The following characters at the beginning of a line are also paragraph delimiters:

```
Space Tab @ - ' )
```

2.5 Executing commands

2.5.1 Keys — Command, Ctrl and Meta

Editor commands are initiated by one or more *key sequences*. A single key sequence usually involves holding down one of two specially defined *modifier* keys, while at the same time pressing another key which is usually a character key.

Mac OS users will be familiar with the use of the **`Command`** key in key sequences such as **`Command+C`**. These keys always work in the standard Mac OS way in the LispWorks editor. The remainder of this section describes the use of other modifier key.

The two modifier keys referred to are the *Control* (**`Ctrl`**) key and the *Meta* key .

When using Emacs emulation on a keyboard without a *Meta* key, the *Escape* (**`Esc`**) key can be used instead. Note that **`Esc`** must be typed *before* pressing the required character key, and not held down.

When using Mac OS editor emulation, **`Esc`** is the cancel gesture and you may not have an Emacs Meta key. Therefore LispWorks provides an alternate gesture to access editor commands: **`Ctrl+M`**. For example, to invoke the command **`Find Source for Dspec`**, type

```
Ctrl+M X Find Source for Dspec
```

and press **`Return`**.

To continue the search, type **`Ctrl+M ,`**.

You can make either the **`Alt`** or the **`Command`** key act as the Emacs Meta key. This setting is independent of whether you are using Emacs or Mac OS editor emulation. See the *LispWorks IDE User Guide* for instructions on changing editor emulation.

An example of a single key sequence command is **`Ctrl+A`** which moves the current point to the start of the line. This command is issued by holding down the **`Control`** key while at the same time pressing **`A`**.

Some key sequences may require more than one key sequence. For example, the key sequence to save the current buffer to a file is **`Ctrl+X Ctrl+S`**. Another multi-key sequence is **`Ctrl+X S`** which saves all buffers to their relevant files. Note that in this case you do not press the **`Control`** key while pressing **`S`**.

A few commands require both the **`Ctrl`** *and* **`Meta`** key to be held down while pressing the character key. **`Meta+Ctrl+L`**, used to select the previous buffer displayed, is one such command. If the **`Esc`** key is being used in place of the **`Meta`** key, then this key should be pressed *before* the **`Ctrl+L`** part of the key sequence.

2.5.2 Two ways to execute commands

The key sequences used to execute commands, as described in the previous section, are only one way to execute an editor command. As a general rule, editor commands that are used frequently should involve as few key strokes as possible to allow for fast editing. The key sequences described above are quick and easy shortcuts for invoking commands.

Most editor commands can also be invoked explicitly by using their full names. For example, in the previous section we met the keystroke **`Ctrl+A`** which moves the current point to the beginning of the line. This keystroke is called a *key binding* and is a shortcut for executing the command **`Beginning of Line`**. To execute this command by name you must type **`Meta+X`** followed by the full command name (**`Meta+X`** itself is only a key binding for the command **`Extended Command`**).

Even though there may seem like a lot of typing to issue the extended version of a command, it is not generally necessary to type in the whole of a command to be executed. The **`Tab`** key can be used to complete a partially typed in extended command. The editor extends the command name as far as possible when **`Tab`** is used, and if the user is not sure of the rest of the command name, then pressing **`Tab`** again provides a list of possible completions. The command can then be selected from this list.

The most commonly used editor commands have a default binding associated with them.

2.5.3 Prefix arguments

An editor command can be supplied with an integer argument *p* which may alter the effect of that command. In most cases it means that the command is repeated *p* times. This argument is known as a *prefix argument* as it is supplied before the command to which it is to be applied. Prefix arguments have no effect on some commands.

See “Using prefix arguments” on page 26 for information about using prefix arguments.

2.6 Basic editing commands

This section contains just enough information to allow you to load a file into the editor, edit that file as required, and then save that file. It is designed to give you enough information to get by and no more.

Only the default bindings are provided. The commands introduced are grouped together as they are in the more detailed command references and under the same headings (except for “Killing and Yanking” on page 14). For further information on the commands described below and other related commands, see the relevant sections in Chapter 3, *Command Reference*.

2.6.1 Aborting commands and processes

See “Aborting commands and processes” on page 19

`Ctrl+G`	Abort the current command which may either be running or just partially typed in. Use **`Esc`** in Mac OS editor emulation.

2.6.2 File handling

See “File handling” on page 27.

`Ctrl+X Ctrl+F` *file*	Load file into a buffer ready for editing. If the name of a non-existent file is given, then an empty buffer is created in to which text can be inserted. Only when a save is done will the file be created.
`Ctrl+X Ctrl+S`	Save the contents of the current buffer to the associated file. If there is no associated file, one is created with the same name as the buffer

2.6.3 Inserting text

See “Inserting text” on page 73 for details of various commands which insert text.

Text which is typed in at the keyboard is automatically inserted to the left of the cursor.

To insert a newline press `Return`.

2.6.4 Movement

See “Movement” on page 54.

`Ctrl+F`	Move the cursor forward one character.
`Ctrl+B`	Move the cursor backward one character.
`Ctrl+N`	Move the cursor down one line.
`Ctrl+P`	Move the cursor up one line.

The above commands can also be executed using the arrow keys.

`Ctrl+A`	Move the cursor to the beginning of the line.
`Ctrl+E`	Move the cursor to the end of the line.
`Ctrl+V`	Scroll one screen forward.
`Meta+V`	Scroll one screen backward.
`Meta+Shift+<`	Move to the beginning of the buffer.
`Meta+Shift+>`	Move to the end of the buffer.

2.6.5 Deleting and killing text

See “Deleting and killing text” on page 67.

`Delete`	Delete the character to the left of the cursor.
`Ctrl+D`	Delete the current character.
`Ctrl+K`	Kill text from the cursor to the end of the line. To delete a whole line (that is, text and newline), type `Ctrl+K` twice at the start of the line.

2.6.6 Undoing

See “Undoing” on page 76.

`Ctrl+Shift+_`	Undo the previous command. If `Ctrl+Shift+_` is typed repeatedly, previously executed commands are undone in a "last executed, first undone" order.

2.6.7 Killing and Yanking

The commands given below are used to copy areas of text and insert them at some other point in the buffer. Note that there is no corresponding "Cut and paste" section in the command references, so direct cross references have been included with each command.

When cutting and pasting, the first thing to do is to copy the region of text to be moved. This is done by taking the cursor to the beginning of the piece of text to be copied and pressing `Ctrl+Space` to set a mark, and then taking the cursor to the end of the text and pressing `Ctrl+W`. This kills the region between the current point and the mark but keeps a copy of the killed text. This copy can then be inserted anywhere in the buffer by putting the cursor at the required position and then pressing `Ctrl+Y` to insert the copied text.

If the original text is to be copied but not killed, use the command `Meta+W` instead of `Ctrl+W.` This copies the text ready for insertion, but does not delete it.

`Ctrl+Space`	Set a mark for a region. See “Marks and regions” on page 63.
`Ctrl+W`	Kill the region between the mark and current point, and save a copy of that region. See “Deleting and killing text” on page 67.
`Meta+W`	Copy the region between the mark and the current point. See “Deleting and killing text” on page 67.
`Ctrl+Y`	Insert (yank) a copied region before the current point. See “Inserting text” on page 73.

2.6.8 Help

See “Help” on page 20.

`Ctrl+H A` *string*	List symbols whose names contain *string* in a Symbol Browser tool.
`Ctrl+H D` *command*	Describe *command*, where *command* is the full command name.
`Ctrl+H K` *key*	Describe the command bound to *key*.

3

Command Reference

This chapter contains full details of most of the editor commands. Details of related editor variables have also been included alongside commands, where appropriate. Not included in this chapter, are commands used to facilitate the editing of Lisp programs. See Chapter 4, *Editing Lisp Programs*.

Commands are grouped according to functionality as follows:

- "Aborting commands and processes"
- "Executing commands"
- "Help"
- "Using prefix arguments"
- "File handling"
- "Filename completion"
- "Directory mode"
- "Movement"
- "Marks and regions"
- "Locations"
- "Deleting and killing text"

- “Inserting text”
- “Delete Selection”
- “Undoing”
- “Case conversion”
- “Transposition”
- “Overwriting”
- “Indentation”
- “Filling”
- “Buffers”
- “Windows”
- “Pages”
- “Searching and replacing”
- “Comparison”
- “Registers”
- “Modes”
- “Abbreviations”
- “Keyboard macros”
- “Echo area operations”
- “Editor variables”
- “Recursive editing”
- “Key bindings”
- “Running shell commands”
- “Buffers, windows and the mouse”
- “Interaction with the GUI and the IDE”
- “Miscellaneous”
- “Obscure commands”

3.1 Aborting commands and processes

Key Sequence

`Ctrl+G`

Aborts the current command. `Ctrl+G` (or `Esc` in Mac OS X editor emulation) can either be used to abandon a command which has been partially typed in, or to abort the command which is currently running.

Note that, unlike most of the keys described in this manual, this cannot be changed via `editor:bind-key`. Instead, use `editor:set-interrupt-keys` if you wish to change this.

Key Sequence

`Command+Ctrl+,`

Chooses a process that is useful to break, and breaks it. The process to break is chosen as follows:

1. It checks for a busy processes that is essential for LispWorks to work correctly, or that interacts with the user (normally that means that some CAPI interface uses it), or that is flagged as wanting interrupts (currently that means a REPL). If it finds such a busy process, it breaks it.
2. Otherwise, if the LispWorks IDE is running, activate or start the Process Browser.
3. Otherwise, if there is a busy process break it.
4. Otherwise, just break the current process.

Note: This break gesture is supported only on Mac OS X 10.4 and later.

3.2 Executing commands

Some commands (usually those used most frequently) are bound to key combinations or key sequences, which means that fewer keystrokes are necessary to execute these commands. Other commands must be invoked explicitly, using `Extended Command`.

It is also possible to execute shell commands from within the editor. See "Running shell commands" on page 145.

Extended Command *Editor Command*

Key sequence: **`Meta+X`**

Allows the user to type in a command name explicitly. Any editor command can be invoked in this way, and this is the usual method of invoking a command that is not bound to any key sequence. Any prefix argument is passed to the command that is invoked.

It is not generally necessary to type in the whole of a command to be executed. Completion (using **`Tab`**) can be used after the first part of the command has been typed.

3.3 Help

The editor provides a number of on-line help facilities, covering a range of areas.

There is one main help command, accessed by **`Help`** (**`Ctrl+H`**), with many options to give you a wide range of help on editor commands, variables and functions.

There are also further help commands which provide information on Lisp symbols (see “Documentation” on page 190).

3.3.1 The help command

Help *Editor Command*

Options: See below
Key sequence: **`Ctrl+H`** *option*

Provides on-line help. Depending on what information the user has and the type of information required, one of the following options should be selected after invoking the **`Help`** command. In most cases a **`Help`** command plus option can also be invoked by an extended editor command.

A brief summary of the help options is given directly below, with more detailed information following.

`?`	Display a list of help options.

`q` or `n`	Quit help.
`a` *string*	Display a list of symbols whose names match *string*, in a Symbol Browser tool.
`b`	Display a list of key bindings and associated commands.
`c` *key*	Display the command to which key is bound.
`d` *command*	Describe the editor *command*.
`Ctrl+D` *command*	Bring up the on-line version of this manual for *command*.
`g` *object*	Invoke the appropriate describe *object* command.
`k` key	Describe the command to which *key* is bound.
`Ctrl+K` *key*	Bring up the on-line version of this manual for *key*.
`l`	describe the last 60 keys typed.
`v` *variable*	Describe *variable* and show its current value.
`Ctrl+V` *variable*	Bring up the on-line version of this manual for *variable*.
`w` *command*	Display the key sequence to which *command* is bound.

Apropos Command *Editor Command*

Arguments: *string*
Key sequence: None

Displays a list of editor commands, variables, and attributes whose names contain *string*, in a Help window.

Editor command, variable and attribute names tend to follow patterns which becomes apparent as you look through this manual. For example, commands which perform operations on files tend to contain the string `file`, that is, **`Find File`**, **`Save File`**, **`Print File`** and so forth.

Use this form of help when you know what you would like to do, but do not know a specific command to do it.

What Command *Editor Command*

Arguments: *key*
Key sequence: **`Ctrl+H C`** *key*

Displays the command to which *key* is bound. For a more detailed description of *key* use the command **`Describe Key.`**

Use this form of help when you know a default binding but want to know the command name.

Note: this command is also available via the menu command **Help > Editing > Key to Command.**

Describe Command *Editor Command*

Arguments: *command*
Key sequence: **`Ctrl+H D`** *command*

Describes the editor command *command*. Full documentation of that command is printed in a Help window.

Use this form of help when you know a command name and require full details of that command.

Document Command *Editor Command*

Arguments: *command*
Key sequence: **`Ctrl+H Ctrl+D`** *command*

Brings up the on-line version of this manual at the entry for *command.*

The documentation in the on-line manual differs from the editor on-line help (as produced by **`Describe Command`**), but provides similar information. If you are used to the layout and definitions provided in this manual then use this help command instead of **`Ctrl+H D`**.

Generic Describe *Editor Command*

Arguments: *object*
Key sequence: `Ctrl+H G` *object*

Describes *object*, where *object* may take the value *command, key, attribute* or *variable*.

If *object* is *command, key* or *variable* then the command **`Describe Command`**, **`Describe Key`** or **`Describe Editor Variable`** is invoked respectively.

There is no corresponding describe command if the object is *attribute*. Attributes are things such as word delimiters, Lisp syntax and parse field separators. If you are not sure of the attributes documented remember that you can press **`Tab`** to display a completion list.

Describe Key *Editor Command*

Arguments: *key*
Key sequence: `Ctrl+H K` *key*

Describes the command to which *key* is bound. Full documentation of that command is printed in a Help window.

Use this form of help when you know a default binding and require the command name plus full details of that command.

Document Key *Editor Command*

Arguments: *key*
Key sequence: `Ctrl+H Ctrl+K` *key*

Brings up the on-line version of this manual at the entry for *key*.

The documentation in the on-line manual differs slightly from the editor on-line help but usually provides you with the same amount of information. If you are used to the layout and definitions provided in this manual then use this help command instead of **`Describe Key`**.

What Lossage *Editor Command*

Arguments: None
Key sequence: **`Ctrl+H L`**

Displays the last 60 keys typed.

Describe Editor Variable *Editor Command*

Arguments: *variable*
Key sequence: **`Ctrl+H V`** *variable*

Describes *variable* and prints its current value in a Help window.

Use this form of help when you know a variable name and require a description of that variable and/or its current value.

Document Variable *Editor Command*

Arguments: *variable*
Key sequence: **`Ctrl+H Ctrl+V`** *variable*

Brings up the on-line version of this manual at the entry for *variable.*

The documentation in the on-line manual differs slightly from the editor on-line help but usually provides you with the same amount of information. If you are used to the layout and definitions provided in this manual then use this help command instead of **`Describe Editor Variable`**.

Where Is *Editor Command*

Arguments: *command*
Key sequence: **`Ctrl+H W`** *command*

Displays the key sequence to which *command* is bound.

Use this form of help if you know a command name and wish to find the bindings for that command. If no binding exists then a message to this effect is returned.

Note: this command is also available via the menu command **Help > Editing > Command to Key.**

Describe Bindings *Editor Command*

Arguments: None
Key sequence: `Ctrl+H B`

Displays a list of key bindings and associated commands in a Help window. First the minor and major mode bindings for the current buffer are printed, then the global bindings.

3.3.2 Other help commands

Manual Entry *Editor Command*

Arguments: *unix-command*
Key sequence: `l`
Mode: Manual Entry

Displays the UNIX manual page for *unix-command.* The UNIX utility **`man`** is invoked and the manual page is displayed in an Editor window.

The buffer is in Manual Entry mode and you can navigate using keys **`p`**, **`n`**, **`s`** and so on - use **`Describe Bindings`** to see all the Manual Entry mode keys.

With no prefix argument, the same buffer is used each time. With a prefix argument, a new buffer is created for each manual page accessed.

See also: “Major modes” on page 118.

Remote Manual Entry *Editor Command*

Arguments: *machine-name unix-command*
Key sequence: `r`
Mode: Manual Entry

The command **`Remote Manual Entry`** is like **`Manual Entry`**, but runs on another computer using **`rsh`**.

Remove Nroff Backspaces *Editor Command*

Arguments: None
Key sequence: None

The command **`Remove Nroff Backspaces`** removes from the current buffer markers that are used by **`nroff`** to go backspace.

Note: **`Manual Entry`** command removes **`nroff`** backspaces automatically.

3.4 Using prefix arguments

Editor Commands can be supplied with an integer argument which, in many cases, indicates how many times a command is to be executed. This argument is known as a *prefix argument* as it is supplied before the command to which it is to be applied.

A prefix argument applied to some commands has a special meaning. Documentation to this effect is provided with the command definitions where appropriate in this manual. In most other cases the prefix argument repeats the command a certain number of times, or has no effect.

A prefix argument can be supplied to a command by first using the command **`Set Prefix Argument`** (**`Ctrl+U`**) followed by an integer. Negative prefix arguments are allowed. A prefix argument between 0 and 9 can also be supplied using **`Meta+`***digit*.

Set Prefix Argument *Editor Command*

Arguments: *integer*
Key sequence: **`Ctrl+U`** *integer*

Provides a prefix argument which, for many commands, indicates the command is to be invoked *integer* times. The required integer should be input and the command to which it applies invoked without an intervening carriage return.

If no integer is given, the prefix argument defaults to the value of **`prefix-argument-default`**.

If **`Set Prefix Argument`** is invoked more than once before a command, the prefix arguments associated with each invocation are multiplied together and the command to which the prefix arguments are to be applied is repeated this number of times. For example, if you typed in **`Ctrl+U Ctrl+U 2`** before a command, then that command would be repeated 8 times.

prefix-argument-default *Editor Variable*

Default value: 4

The default value for the prefix argument if no integer is provided for **`Set Prefix Argument`**.

None *Key Sequence*

Key sequence: **`Meta+<0-9>`**

Provides a prefix argument in a similar fashion to **`Set Prefix Argument`**, except that only integers from 0 to 9 can be used (unless the key bindings are changed).

Negative Argument *Editor Command*

Arguments: None
Key sequence: None

Negates the current prefix argument. If there is currently no prefix argument then it is set to -1.

There is rarely any need for explicit use of this command. Negative prefix arguments can be entered directly with **`Set Prefix Argument`** by typing a `-` before the integer.

3.5 File handling

This section contains details of commands used for file handling.

The first section provides details on commands used to copy the contents of a file into a buffer for editing, while the second deals with copying the contents of buffers to files.

You may at some point have seen file names either enclosed in `#` characters or followed by a `~` character. These files are created by the editor as backups for the file named. The third section deals with periodic backups (producing file names enclosed in `#`) and the fourth with backups on file saving (producing files followed by `~`).

There are many file handling commands which cannot be pigeon-holed so neatly and these are found in the section "Miscellaneous file operations" on page 39. Commands use to print, insert, delete and rename files are covered here, along with many others.

3.5.1 Finding files

Find File *Editor Command*

Arguments: *pathname*
Key sequence: None

`editor:find-file-command` ***p*** **`&optional`** ***pathname***

Finds a new buffer with the same name as *pathname* (where *pathname* is the name of the file to be found, including its directory relative to the current directory), creating it if necessary, and inserts the contents of the file into the buffer. The contents of the buffer are displayed in an editor pane and may then be edited.

The file is initially read in the external format (encoding) given by the editor variable **`input-format-default`**. If the value of this is **`nil`**, **`cl:open`** chooses the external format to use. The external format is remembered for subsequent reading and writing of the buffer, and its name is displayed in the mode line.

If the file is already being visited a new buffer is not created, but the buffer already containing the contents of that file is displayed instead.

If a file with the specified name does not exist, an empty buffer with that file name is created for editing purposes, but the new file is not created until the appropriate save file command is issued.

If there is no prefix argument, a new Editor window is created for the file. With any prefix argument, the file is shown in the current window.

Another version of this command is **`Wfind File`** which is usually used for finding files.

Wfind File *Editor Command*

Arguments: *pathname*
Key sequence: **`Ctrl+X Ctrl+F`** *pathname*

`editor:wfind-file-command` ***p*** **`&optional`** ***pathname***

Calls **`Find File`** with a prefix argument (that is, the new file is opened in the existing window).

Visit File *Editor Command*

Arguments: *pathname*
Key sequence: None

`editor:visit-file-command` ***p*** **`&optional`** ***pathname buffer***

Does the same as **`Find Alternate File`**, and then sets the buffer to be writable.

Find Alternate File *Editor Command*

Arguments: *pathname*
Key sequence: **`Ctrl+X Ctrl+V`** *pathname*

`editor:find-alternate-file-command` ***p*** **`&optional`** ***pathname buffer***

Does the same as **`Find File`** with a prefix argument, but kills the current buffer and replaces it with the newly created buffer containing the file requested. If the contents of the buffer to be killed have been modified, the user is asked if the changes are to be saved to file.

The argument *buffer* is the buffer in which the contents of the file are to be displayed. *buffer* defaults to the current buffer.

The prefix argument is ignored.

3.5.2 Saving files

Save File *Editor Command*

Arguments: None
Key sequence: `Ctrl+X Ctrl+S`

`editor:save-file-command` ***p*** **`&optional`** ***buffer***

Saves the contents of the current buffer to the associated file. If there is no associated file, one is created with the same name as the buffer, and written in the same encoding as specified by the editor variable **`output-format-default`**, or as defaulted by **`open`** if this is **`nil`**.

The argument *buffer* is the buffer to be saved in its associated file. The default is the current buffer.

Save All Files *Editor Command*

Arguments: None
Key sequence: `Ctrl+X S`

Without a prefix argument, a **Select Buffers To Save:** dialog is displayed asking whether each modified buffer is to be saved. If a buffer has no associated file it is ignored, even if it is modified. The selected buffers are saved.

With a non-nil prefix argument, no such dialog is displayed and all buffers that need saving are saved. You can also prevent the **Select Buffers To Save:** dialog from being displayed by setting the value of the editor variable **`save-all-files-confirm`**.

save-all-files-confirm *Editor Variable*

Default value: **`t`**

When the value is true, **`Save All Files`** prompts for confirmation before writing the modified buffers, when used without a prefix argument.

Write File *Editor Command*

Arguments: *pathname*
Key sequence: **`Ctrl+X Ctrl+W`** *pathname*

`editor:write-file-command` ***p*** **`&optional`** ***pathname buffer***

Writes the contents of the current buffer to the file defined by *pathname*. If the file already exists, it is overwritten. If the file does not exist, it is created. The buffer then becomes associated with the new file.

The argument *buffer* is the name of the buffer whose contents are to be written. The default is the current buffer.

Write Region *Editor Command*

Arguments: *pathname*
Key sequence: None

`editor:write-region-command` ***p*** **`&optional`** ***pathname***

Writes the region between the mark and the current point to the file defined by *pathname*. If the file already exists, it is overwritten. If the file does not exist, it is created.

Append to File *Editor Command*

Arguments: *pathname*
Key sequence: None

Appends the region between the mark and the current point to the file defined by *pathname*. If the file does not exist, it is created.

Backup File *Editor Command*

Arguments: *pathname*
Key sequence: None

Writes the contents of the current buffer to the file defined by *pathname*. If the file already exists, it is overwritten. If it does not exist, it is created.

In contrast with `Write File`, no change is made concerning the file associated with the current buffer as this command is only intended to be used to write the contents of the current buffer to a backup file.

Save All Files and Exit *Editor Command*

Arguments: None
Key sequence: `Ctrl+X Ctrl+C`

A **Select Buffers To Save:** dialog is displayed asking whether each modified buffer is to be saved. If a buffer has no associated file it is ignored, even if it is modified (this operates just like `Save All Files`). When all the required buffers have been saved LispWorks exits, prompting for confirmation first.

add-newline-at-eof-on-writing-file *Editor Variable*

Default value: `:ask-user`

Controls whether the commands `Save File` and `Write File` add a newline at the end of the file if the last line is non-empty.

If the value of this variable is `t` then the commands add a newline and tell the user.

If the value is `nil` the commands never add a newline.

If the value is `:ask-user`, the commands ask whether to add a newline.

3.5.3 Unicode and other file encodings

The editor supports the entire Unicode range, and provided that the system has suitable fonts it should be able to display all the characters correctly. Normally you should not be able to have a character object corresponding to a surrogate code point (these codes are the exclusive range `(#xd800, #xdfff)`). If such an object is inserted, the editor displays its hexadecimal value.

An editor buffer ideally should have an appropriate external format (or encoding) set before you write it to a file. Otherwise an external format specified in the value of the editor variable **`output-format-default`** is used. If the value of **`output-format-default`** is not an external format specifier, then the external format is chosen similarly to the way **`cl:open`** does it. By default this chosen external format will be the Windows code page on Microsoft Windows, and Latin-1 on other platforms.

When using the Editor tool, use **`Set External Format`** to set interactively the external format for the current buffer, or set **Preferences... > Environment > File Encodings > Output** (which in turn sets the editor variable **`output-format-default`**) to provide a global default value. You can also use **`Find File With External Format`** to specify the external format before reading a file.

In situations where you want to open a file in a 16-bit encoding but the file is not actually encoded properly (for example it is actually a binary containing some strings encoded in **`:utf-16`**), use one of the **`:utf-16`** or **`:bmp`** external formats with the parameter **`:use-replacement t`**, for example:

```
(:utf-16 :use-replacement t)
```

These external formats will replace any input that causes errors by the replacement character (code point **`#xfffd`**), and should successfully read correctly encoded **`:utf-16`** strings including supplementary characters.

If you need to edit a file that is not properly encoded, the only external format that can do this is **`:latin-1`**. To insert a multi-byte character, you will have to insert the **`:latin-1`** characters matching the individual bytes in the right order.

See the *LispWorks User Guide and Reference Manual* for a description of external format specifications.

Compatibility Note: In LispWorks 6.1 and earlier versions, **`:unicode`** is the best choice of external format for opening an incorrectly-encoded file. However, in LispWorks 7.0 and later versions **`:unicode`** maps to **`:utf-16`** which is quite likely to give an error trying to read a binary file, unless you supply **`:use-replacement t`** as described above. The error would occur when it sees a 16-bit value which is a surrogate code point.

3.5.3.1 Controlling the external format

Find File With External Format *Editor Command*

Arguments: None
Key sequence: None

The command **`Find File With External Format`** prompts for an external format, and then opens the file as as if by **`Wfind File`**, with the supplied external format.

This external format is also used when subsequently saving the file.

Set External Format *Editor Command*

Arguments: *buffer*
Key sequence: None

Prompts for an external format specification, providing a default which is the buffer's current external format if set, or the value of **`output-format-default`**. Sets the buffer's external format, so that this is used for subsequent file writing and reading.

If a non-nil prefix argument is supplied, the buffer's external format is set to the value of **`output-format-default`** without prompting.

input-format-default *Editor Variable*

Default value: **`nil`**

The default external format used by **`Find File`**, **`Wfind File`** and **`Visit File`** for reading files into buffers.

If the buffer already has an external format (either it has previously been read from a file, or **`Set External Format`** has been used to specify an external format) then **`input-format-default`** is ignored. If the value is **`nil`** and the buffer does not have an external format, **`cl:open`** chooses the external format to use.

The value should be **nil** or an external format specification. See the *Lisp-Works User Guide and Reference Manual* for a description of these and of how **cl:open** chooses an external format.

If you have specified an input encoding via the Editor tool's Preferences dialog, then **input-format-default** is initialized to that value on startup.

output-format-default *Editor Variable*

Default value: **nil**

The default external format used for writing buffers to files.

If the buffer already has an external format (either it has been read from a file, or **Set External Format** has been used to specify an external format) then **output-format-default** is ignored. If the value is **nil** and the buffer does not have an external format, **cl:open** chooses the external format to use.

The value should be **nil** or an external format specification. See the *Lisp-Works User Guide and Reference Manual* for a description of these and of how **cl:open** chooses an external format.

If you have specified an output encoding via the Editor tool's Preferences dialog, then **output-format-default** is initialized to that value on startup.

The default value of **output-format-default** is **nil**.

3.5.3.2 Unwritable characters

If your buffer contains a character *char* which cannot be encoded in the buffer's external format (or the defaulted external format) then attempts to save the buffer will signal an error giving the character name, its offset in the buffer and explaining that *char* is unwritable in the external format.

In particular if your buffer contains a **cl:extended-char** *char* then Latin-1 and other encodings which support only **cl:base-char** are not appropriate.

There are two ways to resolve this:

- Set the external format to one which includes *char,* or

- Delete *char* from the buffer before saving. The commands **`Find Unwritable Character`** and **`List Unwritable Characters`** will help you to identify the character(s) that cannot be written.

You may want a file which is Unicode UTF-16 encoded (external format **`:unicode`**), UTF-8 encoding (**`:utf-8`**) or a language-specific encoding such as **`:shift-jis`** or **`:gbk`**. Or you may want a Latin-1 encoded file, in which case you could supply **`:latin-1-safe`**.

Find Unwritable Character *Editor Command*

Arguments: None
Key sequence: None

Finds the next occurrence of a character in the current buffer that cannot be written using the buffer external format. The prefix argument is ignored.

List Unwritable Characters *Editor Command*

Arguments: None
Key sequence: None

Lists the characters in the current buffer that cannot be written with the buffer external format. The prefix argument is ignored.

Find Non-Base-Char *Editor Command*

Arguments: None
Key sequence: None

The command **`Find Non-Base-Char`** finds the next character in the current buffer that is not a **`cl:base-char`**, starting from the current point.

3.5.4 Auto-saving files

The auto-save feature allows for periodic backups of the file associated with the current buffer. These backups are only made if auto-save is switched on.

This feature is useful if the LispWorks editor is killed in some way (for example, in the case of a system crash or accidental killing of the editor process) before a file is explicitly saved. If automatic backups are being made, the state of a file when it was last auto-saved can subsequently be recovered.

By default, automatic backups are made both after a predefined number of key strokes, and also after a predefined amount of time has elapsed.

By default, auto-saved files are in the same directory as the original file, with the name of the auto-save file (or "checkpoint file") being the name of the original file enclosed within `#` characters.

Toggle Auto Save *Editor Command*

Arguments: None
Key sequence: None

Switches auto-save on if it is currently off, and off if it is currently on.

With a positive prefix argument, auto-save is switched on. With a negative or zero prefix argument, auto-save is switched off. Using prefix arguments with **`Toggle Auto Save`** disregards the current state of auto-save.

`Auto Save Toggle` is a synonym for **`Toggle Auto Save`**.

auto-save is initially on or off in a new buffer according to the value of the editor variable **`default-auto-save-on`**.

default-auto-save-on *Editor Variable*

Default value: **`t`**

The default auto-save state of new buffers.

auto-save-filename-pattern *Editor Variable*

Default value: *"~A#~A#"*

This is a **`format`** control string used to make the filename of the checkpoint file. **`format`** is called with two arguments, the first being the directory namestring and the second being the file namestring of the default buffer pathname.

The default value causes the auto-save file to be created in the same directory as the file for which it is a backup, and with the name surrounded by `#` characters.

auto-save-key-count-threshold *Editor Variable*

Default value: 256

Specifies the number of destructive / modifying keystrokes that automatically trigger an auto-save of a buffer. If the value is `nil`, this feature is turned off.

auto-save-checkpoint-frequency *Editor Variable*

Default value: 300

Specifies the time interval in seconds after which all modified buffers which are in "Save" mode are auto-saved. If the value is `nil`, zero or negative, this feature is turned off.

auto-save-cleanup-checkpoints *Editor Variable*

Default value: `t`.

This variable controls whether an auto-save function will cleanup by deleting the checkpoint file for a buffer after it is saved. If the value is true then this cleanup will occur.

3.5.5 Backing-up files on saving

When a file is explicitly saved in the editor, a backup is automatically made by writing the old contents of the file to a backup before saving the new version of the file. The backup file appears in the same directory as the original file. By default its name is the same as the original file followed by a `~` character.

backups-wanted *Editor Variable*

Default value: `t`

Controls whether to make a backup copy of a file the first time it is modified. If the value is `t`, a backups is automatically made on first saving. If the value is `nil`, no backup is made.

backup-filename-suffix *Editor Variable*

Default value: #\~

This variable contains the character used as a suffix for backup files. By default, this is the tilde (~) character.

backup-filename-pattern *Editor Variable*

Default value: *"~A~A~A"*

This control string is used with the Common Lisp `format` function to create the filename of the backup file. `format` is called with three arguments, the first being the directory name-string and the second being the file name-string of the pathname associated with the buffer. The third is the value of the editor variable *backup-filename-suffix*.

The backup file is created in the same directory as the file for which it is a backup, and it has the same name, followed by the *backup-filename-suffix*.

Note that the backup-suffix can be changed functionally as well as by interactive means. For example, the following code changes the suffix to the `@` character:

```
(setf (editor:variable-value 'editor:backup-filename-suffix
          :current nil) #\@)
```

3.5.6 Miscellaneous file operations

Print File *Editor Command*

Arguments: *file*
Key sequence: None

Prints *file*, using `capi:print-file`. See the *CAPI User Guide and Reference Manual* for details of this function.

Revert Buffer *Editor Command*

Arguments: None
Key sequence: None

If the current buffer is associated with a file, its contents revert to the state when it was last saved. If the buffer is not associated with a file, it is not possible for a previous state to be recovered.

If auto-save is on for the current buffer, the version of the file that is recovered is either that derived by means of an automatic save or by means of an explicit save, whichever is the most recent. If auto-save is off, the buffer reverts to its state when last explicitly saved.

If the buffer has been modified and the value of the variable **`revert-buffer-confirm`** is **`t`** then **`Revert Buffer`** asks for confirmation before reverting to a previous state.

Any prefix argument forces **`Revert Buffer`** to use the last explicitly saved version.

revert-buffer-confirm *Editor Variable*

Default value: **`t`**

When the command **`Revert Buffer`** is invoked, if the value of this variable is **`t`** and the buffer has been modified then confirmation is requested before the revert operation is performed. If its value is **`nil`**, no confirmation is asked for.

Process File Options *Editor Command*

Arguments: None
Key sequence: None

The attribute line at the top of the file is reprocessed, as if the file had just been read from disk. If no major mode is specified in the attribute line, the type of the file is used to determine the major mode. See “Modes” on page 118.

Insert File *Editor Command*

Arguments: *pathname*
Key sequence: `Ctrl+X I` *pathname*

`editor:insert-file-command` ***p*** **`&optional`** ***pathname buffer***

Inserts the file defined by *pathname* into the current buffer at the current point.

The argument *buffer* is the buffer in which the file is to be inserted.

Delete File *Editor Command*

Arguments: *pathname*
Key sequence: None

Deletes the file defined by *pathname*. The user is asked for confirmation before the file is deleted.

Delete File and Kill Buffer *Editor Command*

Arguments: *buffer*
Key sequence: None

`editor:delete-file-and-kill-buffer-command` ***p*** **`&optional`** ***buffer***

After confirmation from the user, this deletes the file associated with *buffer* and then kills the buffer.

Rename File *Editor Command*

Arguments: *file new-file-name*
Key sequence: None

Changes the name of *file* to *new-file-name*.

If you are currently editing the file to be renamed, the buffer remains unaltered, retaining the name associated with the old file even after renaming has taken place. If you then save the current buffer, it is saved to a file with the name of the buffer, that is, to a file with the old name.

Make Directory *Editor Command*

Arguments: None
Key sequence: None

Prompts the user for a directory name and makes it in the filesystem.

The prefix argument is ignored.

List Directory *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X D`**

The command **`List Directory`** prompts for a directory or wild filename and finds or creates a buffer which lists files and allows you to operate on them easily.

See “Directory mode” on page 43 for detailed information about Directory mode.

Save Buffer Pathname *Editor Command*

Arguments: None
Key sequence: None

Pushes the namestring of the pathname of the current buffer onto the kill ring. This namestring can then be inserted elsewhere by commands which access the kill ring, described in “Inserting text” on page 73.

3.6 Filename completion

Expand File Name *Editor Command*

Arguments: None
Key sequence: **`Meta+Tab`**

Key sequence: **`Tab`**
Mode: Shell

The command **`Expand File Name`** expands (completes) the filename at the current point.

The system looks backwards from the current point until it finds a space or other character that is unlikely to be in a filename. The text from this character to the current point is the partial filename to complete.

Invoking **`Expand File Name`** twice in succession offers a list of possible completions.

See also: **`Expand File Name With Space`**

Expand File Name With Space *Editor Command*

Arguments: None
Key sequence: None

The command **`Expand File Name With Space`** is like **`Expand File Name`**, but allows spaces in the filename it tries to complete.

See also: **`Expand File Name`**

3.7 Directory mode

A buffer in Directory mode presents a list of files, and allows you to easily edit any of them, copy or move some of them to another directory, or delete some of them. It also makes it easy to keep a record of which files you already edited.

You open a Directory mode buffer by invoking one of:

- **`Find File`** or **`Wfind File`** with a directory path.
- **`Find File`** or **`Wfind File`** with a wild filename (that is, the name contains the character **`*`**).
- **`List Directory`**.

The editor opens a buffer in Directory mode, listing all the matching files.

Note: If you are opening a directory path (without filename) and there is already a buffer opened with this directory, it finds this buffer, rather than creating another one. You can prevent this by first renaming the existing buffer. Opening a wild path with the **`Find File`** command always creates a new buffer.

A Directory mode buffer can be saved to a file, and because it contains the mode in its attribute line, when you re-open the file it will open in Directory mode. Thus it can be used as a record of what you have done. For example, if you need to visit all the files in some directory and the task will span multiple sessions, you can edit the directory and visit the files from the Directory mode buffer. You periodically save this buffer to a file. Then after quitting your session and restarting you can open the file and have a record of which files you already visited. For this kind of task, Directory mode is probably the simplest method.

The operations that you can do in Directory mode include:

- Editing a file (automatically mark it as edited).
- Marking / unmarking a file.
- Toggle the edited marking.
- Copy or all marked files to another directory.
- Delete all marked files
- Rename the file on the current line
- Make another buffer in Directory mode with some of the files in the current buffer.

3.7.1 Directory mode buffer display

The first 2 lines of a Directory mode buffer are the "header", including the attribute line. The following lines each represent one file. The line starts with spaces for optional marks, followed by the file size in bytes (decimal), followed by the name of the file.

Each of the optional marks in the beginning of a line is either Space for "off", or a specific character for "on" as shown in Table 1

Table 1 Meaning of "on" characters at start of lines in Directory mode

Offset	Character	Meaning
0	+	Edited
1	*	Marked

Table 1 Meaning of "on" characters at start of lines in Directory mode

Offset	Character	Meaning
1	D	Delete

The remainder of this section contains details of the Directory mode commands.

3.7.1 Directory mode commands

In general the buffer in Directory mode is read-only, and can be modified only by the commands below. Commands that do not modify the text can be used as in other buffers. You should not edit the buffer in other ways, because the editor expects a specific structure of the buffer. Commands that just change the contents of the buffer without affecting the file system can be undone as usual. Commands that affect the file system clear the undo information, so it is not possible to undo these.

Directory Mode Next Line *Editor Command*

Arguments: None
Key sequence: **`Space`**, **`N`**, **`Ctrl+N`** or **`Down`**

The command **`Directory Mode Next Line`** moves to the next line in the buffer, with the point on the filename.

Directory Mode Previous Line *Editor Command*

Arguments: None
Key sequence: **`P`**, **`Ctrl+P`** or **`Up`**

The command **`Directory Mode Previous Line`** moves to the previous line in the buffer, with the point on the filename.

Directory Mode Edit File *Editor Command*

Arguments: None
Key sequence: **`Enter`**, **`F`** or **`E`**

The command **`Directory Mode Edit File`** edits the file on the current line, and also automatically marks it as edited. The file is opened in the same window.

Directory Mode Edit File In Other Window *Editor Command*

Arguments: None
Key sequence: **`O`**

The command **`Directory Mode Edit File In Other Window`** edits the file on the current line, and also automatically marks it as edited. The file is opened in another window.

Note: a convenient setup for visiting files is to use **`Split Window Horizontally`** (**`Ctrl+X 5`**) to display the Directory mode buffer, and then editing a file by **`O`** appears in the other editor window inside the same interface.

Directory Mode Mark *Editor Command*

Arguments: None
Key sequence: **`M`**

The command **`Directory Mode Mark`** switches on the mark (the second character) on the current line.

Marks are used by other commands, but do not have any effect otherwise.

After marking the cursor moves to the next line.

With a prefix argument argument it does as many lines as specified by the prefix, while a negative prefix causes lines above the current line to be marked.

Directory Mode Unmark *Editor Command*

Arguments: None
Key sequence: **`U`**

The command **`Directory Mode Unmark`** switches off the mark (the second character) on the current line.

Marks are used by other commands, but do not have any effect otherwise.

After unmarking the cursor moves to the next line.

With a prefix argument argument it does as many lines as specified by the prefix, while a negative prefix causes lines above the current one to be unmarked.

Directory Mode Unmark Backward *Editor Command*

Arguments: None
Key sequence: **`Backspace`**

The command **`Directory Mode Unmark Backward`** moves to the previous line and switches off the mark. This is equivalent to **`Directory Mode Unmark`** with the prefix argument negated (or supplied as -1).

Directory Mode Unflag Edited *Editor Command*

Arguments: None
Key sequence: None

The command **`Directory Mode Unflag Edited`** switches off the edited flag (**`+`** in the first character) on the current line.

Directory Mode Flag Edited *Editor Command*

Arguments: None
Key sequence: None

The command **`Directory Mode Flag Edited`** switches on the edited flag (**`+`** in the first character) on the current line.

Directory Mode Toggle Edited *Editor Command*

Arguments: None
Key sequence: **`-`**

The command `Directory Mode Toggle Edited` changes the state of the edited flag (`+` in the first character) on the current line. The edited flag is merely recorded in the buffer, not stored anywhere else.

Since the flag is switched on automatically when you edit a file from the Directory mode buffer, you normally do not need to change it, but sometimes you may find it useful.

Directory Mode Mark Matches
Directory Mode Unmark Matches
Directory Mode Mark Regexp Matches
Directory Mode Unmark Regexp Matches

Editor Commands

Arguments: None
Key sequence: None

The commands `Directory Mode Mark Matches`, `Directory Mode Unmark Matches`, `Directory Mode Mark Regexp Matches` and `Directory Mode Unmark Regexp Matches` mark or unmark the matching filenames. With a prefix argument, these commands mark the non-matching filenames. These commands first prompt for a string or regexp to match, and then mark or unmark all the matches (non-matches with prefix argument).

See also: `Directory Mode Mark All`

Directory Mode Mark All

Editor Command

Arguments: None
Key sequence: None

The command `Directory Mode Mark All` marks all filenames. With a prefix argument, this command unmarks all filenames.

See also: `Directory Mode Mark Matches`

Directory Mode Mark When Edited Directory Mode Unmark When Edited

Editor Commands

Arguments: None
Key sequence: None

The commands **`Directory Mode Mark When Edited`** and **`Directory Mode Unmark When Edited`** mark and unmark all edited filenames. With a prefix argument, these commands operate on all unedited filenames.

See also: **`Directory Mode Mark All`**

Directory Mode Flag Delete

Editor Command

Arguments: None
Key sequence: **`D`**

The command **`Directory Mode Flag Delete`** switches on the Delete flag (**`D`** in the second character) on the current line.

The Delete flag is used by the command **`Directory Mode Delete`**, otherwise nothing uses it.

After marking the cursor moves to the next line.

With a prefix argument it does as many lines as specified by the prefix. A negative prefix argument causes lines above the current one to be marked for deletion.

Directory Mode Flag Delete When Marked

Editor Command

Arguments: None
Key sequence: None

The command **`Directory Mode Flag Delete When Marked`** flags for deletion all the marked filenames. With a prefix argument, it flags all the unmarked filenames.

3.7.2 Explicit editing of the Directory mode buffer

Directory Mode Kill Line *Editor Command*

Arguments: None
Key sequence: `Ctrl+K`

The command `Directory Mode Kill Line` kills the current line. This is like the ordinary `Kill Line` command, except that it always removes complete lines (rather than from the point), and it gives an editor error if you try to delete part of the header.

Force Undo *Editor Command*

Arguments: None
Key sequence: `Ctrl+_` or `Ctrl+X U`

The command `Force Undo` is the same as `Undo`, but works for a read-only buffer too.

Note: This command can be used in other modes too.

3.7.3 Modifying the file system from the Directory mode buffer

Directory Mode Delete *Editor Command*

Arguments: None
Key sequence: `X`

The command `Directory Mode Delete` deletes the files that are marked for deleting (`D` in second character).

It first confirms that you really want to delete the files, and then deletes them.

It also deletes the corresponding lines and clears the undo information in the Directory mode buffer.

Note: Like anything that deletes files, you need to be careful when using this command.

Note: When deleting many files, it is convenient to first create a buffer with only the marked files using `Directory Mode New Buffer With Flagged Delete`. That makes it easy to see which files you are going to delete.

See also: `Directory Mode Flag Delete`

Directory Mode Copy Marked *Editor Command*

Arguments: None
Key sequence: `C`

The command `Directory Mode Copy Marked` copies the marked files to another directory. First it prompts for a directory, and then copies the marked files to that directory.

This command clears the undo information in the Directory mode buffer.

Note: When copying many files, it is convenient to first create a buffer with only the marked files using `Directory Mode New Buffer With Marked` (keystroke `T`)..That makes it easy to see which files you are going to copy.

Directory Mode Move Marked *Editor Command*

Arguments: None
Key sequence: `Y`

The command `Directory Mode Move Marked` moves the marked files to another directory. First it prompts for a directory, and then moves the marked files to that directory.

This command also removes the corresponding lines and clears the undo information in the Directory mode buffer.

Note: When moving many files, it is convenient to first create a buffer with only the marked files using `Directory Mode New Buffer With Marked` (keystroke `T`). That makes it easy to see which files you are going to move.

Directory Mode Rename *Editor Command*

Arguments: None
Key sequence: **`R`**

The command **`Directory Mode Rename`** renames the file on the current line.

This prompts for a new name for the file, and then renames the file. It then changes the line to contain the new name.

This command clears the undo information in the Directory mode buffer.

3.7.4 Creating new Directory mode buffers

Directory Mode New Buffer With Marked *Editor Command*

Arguments: None
Key sequence: **`T`**

The command **`Directory Mode New Buffer With Marked`** creates a new buffer in Directory mode, containing only the marked lines (that is, those **`with *`**). With a prefix argument, it creates a buffer with only the unmarked lines.

This command does not affect the current buffer.

Note: This is especially useful before doing a batch operation (delete, copy or move) to first check that you are operating on the correct set of files.

Directory Mode New Buffer With Edited *Editor Command*

Arguments: None
Key sequence: **`Ctrl+T`**

The command **`Directory Mode New Buffer With Edited`** creates a new buffer in Directory mode, containing only the edited lines (that is, those with **`+`**).

With a prefix argument, it creates a buffer with only the un-edited lines.

This command does not affect the current buffer.

Directory Mode New Buffer With Flagged Delete *Editor Command*

Arguments: None
Key sequence: `Meta+T`

The command `Directory Mode New Buffer With Flagged Delete` creates a new buffer in Directory mode, containing only the "delete" lines (that is, those with `D`).

With a prefix argument, it creates a buffer with only the lines that are not flagged for deletion.

This command does not affect the current buffer.

Directory Mode New Buffer With Matches *Editor Command*

Arguments: None
Key sequence: `S`

The command `Directory Mode New Buffer With Matches` prompts for a string, and then creates a buffer containing only the lines that match this string. With a prefix argument it creates a buffer with only the non-matching lines.

This command does not affect the current buffer.

Directory Mode New Buffer With Regexp Matches *Editor Command*

Arguments: None
Key sequence: `Meta+S`

The command `Directory Mode New Buffer With Regexp Matches` prompts for a regular expression, and then creates a buffer containing only the lines that match this regular expression. With a prefix argument it creates a buffer with only the non-matching lines.

This command does not affect the current buffer.

3.8 Movement

This section gives details of commands used to move the current point (indicated by the cursor) around the buffer.

The use of prefix arguments with this set of commands can be very useful, as they allow you to get where you want to go faster. In general, using a negative prefix argument repeats these commands a certain number of times in the opposite logical direction. For example, the command **`Ctrl+U 10 Ctrl+B`** moves the cursor 10 characters backwards, but the command **`Ctrl+U -10 Ctrl+B`** moves the cursor 10 characters *forward*.

Some movement commands may behave slightly differently in different modes as delimiter characters may vary.

To help you keep track of places you have visited, commands which are likely move the point some distance record their starting point as a *location*. This location can later be revisited by the commands listed in "Locations" on page 66.

Forward Character *Editor Command*

Arguments: None
Key sequence: **`Ctrl+F`** or **`Right`**

Moves the current point forward one character.

Backward Character *Editor Command*

Arguments: None
Key sequence: **`Ctrl+B`** or **`Left`**

Moves the current point backward one character.

Forward Word *Editor Command*

Arguments: None
Key sequence: **`Meta+F`**

Moves the current point forward one word.

Backward Word *Editor Command*

Arguments: None
Key sequence: **`Meta+B`**

Moves the current point backward one word.

Beginning of Line *Editor Command*

Arguments: None
Key sequence: **`Ctrl+A`**

Moves the current point to the beginning of the current line.

End of Line *Editor Command*

Arguments: None
Key sequence: **`Ctrl+E`**

Moves the current point to the end of the current line.

Next Line *Editor Command*

Arguments: None
Key sequence: **`Ctrl+N`** or **`Down`**

Moves the current point down one line. If that would be after the end of the line, the current point is moved to the end of the line instead.

Previous Line *Editor Command*

Arguments: None
Key sequence: **`Ctrl+P`** or **`Up`**

Moves the current point up one line. If that would be after the end of the line, the current point is moved to the end of the line instead.

Goto Line *Editor Command*

Arguments: *number*
Key sequence: None

Moves to the line numbered *number*.

Records the starting *location* (see “Locations” on page 66).

What Line *Editor Command*

Arguments: None.
Key sequence: None

Prints in the Echo Area the line number of the current point.

Forward Sentence *Editor Command*

Arguments: None
Key sequence: **`Meta+E`**

Moves the current point to the end of the current sentence. If the current point is already at the end of a sentence, it is moved to the end of the next sentence.

Backward Sentence *Editor Command*

Arguments: None
Key sequence: **`Meta+A`**

Moves the current point to the start of the current sentence. If the current point is already at the start of a sentence, it is moved to the beginning of the previous sentence.

Forward Paragraph *Editor Command*

Arguments: None
Key sequence: **`Meta+]`**

Moves the current point to the end of the current paragraph. If the current point is already at the end of a paragraph, then it is moved to the end of the next paragraph.

Backward Paragraph *Editor Command*

Arguments: None
Key sequence: **`Meta+[`**

Moves the current point to the start of the current paragraph. If the current point is already at the start of a paragraph, then it is moved to the beginning of the previous paragraph.

Scroll Window Down *Editor Command*

Arguments: None
Key sequence: **`Ctrl+V`**

`editor:scroll-window-down-command` ***p*** **`&optional`** ***window***

Changes the text that is being displayed to be one screenful forward, minus **`scroll-overlap`**. If the current point is no longer included in the new text, it is moved to the start of the line nearest to the centre of the window.

A prefix argument causes the current screen to be scrolled up the number of lines specified and that number of new lines are shown at the bottom of the window.

The argument *window* is the name of the window to be scrolled. The default is the current window.

Scroll Window Up *Editor Command*

Arguments: None
Key sequence: **`Meta+V`**

`editor:scroll-window-up-command` ***p*** **`&optional`** ***window***

Changes the text that is being displayed to be one screenful back, minus `scroll-overlap`. If the current point is no longer included in the new text, it is moved to the start of the line nearest to the centre of the window.

A prefix argument causes the current screen to be scrolled down the number of lines specified and that number of new lines are shown at the top of the window.

The argument *window* is the name of the window to be scrolled. The default is the current window.

scroll-overlap *Editor Variable*

Default value: 1

Determines the number of lines of overlap when **`Scroll Window Down`** and **`Scroll Window Up`** are used with no prefix argument.

Line to Top of Window *Editor Command*

Arguments: None
Key sequence: None

Moves the current line to the top of the window.

Top of Window *Editor Command*

Arguments: None
Key sequence: None

Moves the current point to the start of the first line currently displayed in the window.

Bottom of Window *Editor Command*

Arguments: None
Key sequence: None

Moves the current point to the start of the last line that is currently displayed in the window.

Move to Window Line *Editor Command*

Arguments: None
Key sequence: **`Meta+Shift+R`**

Without a prefix argument, moves the current point to the start of the center line in the window.

With a positive (negative) integer prefix argument *p*, moves the point to the start of the *p*th line from the top (bottom) of the window.

Beginning of Buffer *Editor Command*

Arguments: None
Key sequence: **`Meta+Shift+<`**

Moves the current point to the beginning of the current buffer.

Records the initial *location* (see “Locations” on page 66).

End of Buffer *Editor Command*

Arguments: None
Key sequence: **`Meta+Shift+>`**

Moves the current point to the end of the current buffer.

Records the initial *location* (see “Locations” on page 66).

Beginning of Buffer Preserving Point *Editor Command*

Arguments: None
Key sequence in Mac OS X editor emulation: **`Home`**

The command **`Beginning of Buffer Preserving Point`** scrolls the current window to the beginning of the buffer, without moving the buffer point.

End of Buffer Preserving Point *Editor Command*

Arguments: None
Key sequence in Mac OS X editor emulation: **`End`**

The command **`End of Buffer Preserving Point`** scrolls the current window to the end of the buffer, without moving the buffer point.

Beginning of Window *Editor Command*

Arguments: None
Key sequence: **`Ctrl+Prior`**

The command **`Beginning of Window`** moves the buffer point to the beginning of the window.

End of Window *Editor Command*

Arguments: None
Key sequence: **`Ctrl+Next`**

The command **`End of Window`** moves the buffer point to the end of the last line that is fully displayed.

Skip Whitespace *Editor Command*

Arguments: None
Key sequence: None

Skips to the next non-whitespace character if the current character is a whitespace character (for example, **`Space`**, **`Tab`** or newline).

Goto Point *Editor Command*

Arguments: *point*
Key sequence: None

Moves the current point to *point*, where *point* is a character position in the current buffer.

Scroll Window Down Preserving Highlight *Editor Command*

Arguments: None
Key sequence: **`Shift+Next`**

The command **`Scroll Window Down Preserving Highlight`** is the same as **`Scroll Window Down`** except that if there is a highlight region it is extended to the new position of the point rather than unhighlighted.

Scroll Window Up Preserving Highlight *Editor Command*

Arguments: None
Key sequence: **`Shift+Prior`**

The command **`Scroll Window Up Preserving Highlight`** is the same as **`Scroll Window Up`** except that if there is a highlight region it is extended to the new position of the point rather than unhighlighted.

Scroll Window Down In Place
Scroll Window Up In Place *Editor Commands*

Arguments: None
Key sequence: None

The commands **`Scroll Window Down In Place`** and **`Scroll Window Up In Place`** scroll the window up or down, keeping the point in the same place on the screen as much as possible.

Without a prefix argument, scrolls one line. With a prefix argument, scrolls that many lines.

Note: These commands differ from other **`Scroll Window...`** commands in that, by default, they scroll one line rather than whole pages. They also retain any highlight.

Scroll Window Up Moving Point *Editor Command*

Arguments: None
Key sequence in Microsoft Windows editor emulation: **`Prior`**
Key sequence in Mac OS X editor emulation: **`Ctrl+Prior`**

The command `Scroll Window Up Moving Point` scrolls the window up. If the current point is not in the newly-displayed text, it is moved appropriately, trying to keep it in the same place on the screen.

Without a prefix argument, it scrolls by the window height less `scroll-overlap`. With a prefix argument *p*, the current window is scrolled *p* lines and *p* new lines are shown at the top.

Scroll Window Down Moving Point *Editor Command*

Arguments: None
Key sequence in Microsoft Windows editor emulation: **`Next`**
Key sequence in Mac OS X editor emulation: **`Ctrl+Next`**

The command `Scroll Window Down Moving Point` scrolls the window down. If the current point is not in the newly-displayed text, it is moved appropriately, trying to keep it in the same place on the screen.

Without a prefix argument, it scrolls by the window height less `scroll-overlap`. With a prefix argument *p*, the current window is scrolled *p* lines and *p* new lines are shown at the bottom.

Scroll Window Up Preserving Point *Editor Command*

Arguments: None
Key sequence in Mac OS X editor emulation: **`Ctrl+Up`** or **`Prior`**

The command `Scroll Window Up Preserving Point` is the same as `Scroll Window Up` except that, when the editor emulation does not force the point to be visible (Microsoft Windows and Mac OS X), it does not move the point when it becomes invisible.

Scroll Window Down Preserving Point *Editor Command*

Arguments: None
Key sequence in Mac OS X editor emulation: **`Ctrl+Down`** or **`Next`**

The command `Scroll Window Down Preserving Point` is the same as `Scroll Window Down` except that, when the emulation does not force the point to be visible (Microsoft Windows and Mac OS X), it does not move the point when it becomes invisible.

3.9 Marks and regions

The first part of this section gives details of commands associated with marking, while the second provides details of a few commands whose area is limited to a region. Other region specific commands are available but are dealt with in more appropriate sections of this manual. For example, `Write Region` is dealt with under the “File handling” on page 27 as it involves writing a region to a file.

Details of marks are kept in a mark ring so that previously defined marks can be accessed. The mark ring works like a stack, in that marks are pushed onto the ring and can only be popped off on a "last in first out" basis. Each buffer has its own mark ring.

Note that marks may also be set by using the mouse—see “Buffers, windows and the mouse” on page 149—but also note that a region must be defined *either* by using the mouse *or* by using editor key sequences, as the region may become unset if a combination of the two is used. For example, using `Ctrl+Space` to set a mark and then using the mouse to go to the start of the required region unsets the mark.

Note: the editor also records *locations* of the current point which can be revisited by the commands listed in “Locations” on page 66. Unlike marks, these locations do not interact with the region.

3.9.1 Marks

Set Mark *Editor Command*

Arguments: None
Key sequence: `Ctrl+Space` or Middle Mouse Button

With no prefix argument, pushes the current point onto the mark ring, effectively setting the mark to the current point, and activates the region.

With a prefix argument equal to the value of the `prefix-argument-default`, `Pop and Goto Mark` is invoked.

With a prefix argument equal to the square of the **`prefix-argument-default`** (achieved by typing **`Ctrl+U Ctrl+U`** before invoking **`Set Mark)`**, **`Pop Mark`** is invoked.

Pop and Goto Mark *Editor Command*

Arguments: None
Key sequence: None

Moves the current point to the mark without saving the current point on the mark ring (in contrast with **`Exchange Point and Mark`**). After the current point has been moved to the mark, the mark ring is rotated. The current region is de-activated.

Pop Mark *Editor Command*

Arguments: None
Key sequence: **`Meta+Ctrl+Space`**

Rotates the mark ring so that the previous mark becomes the current mark. The point is not moved but the current region is de-activated.

Exchange Point and Mark *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X Ctrl+X`**

`editor:exchange-point-and-mark-command` ***p*** **`&optional`** ***buffer***

Sets the mark to the current point and moves the current point to the previous mark. This command can therefore be used to examine the extent of the current region.

The argument *buffer* is the buffer in which to exchange the point and mark. The default value is the current buffer.

Mark Word *Editor Command*

Arguments: *number*
Key sequence: **`Meta+@`**

Marks the word following the current point. A prefix argument, if supplied, specifies the number of words marked.

Mark Sentence *Editor Command*

Arguments: None
Key sequence: None

Puts the mark at the end of the current sentence and the current point at the start of the current sentence. The sentence thereby becomes the current region. If the current point is initially located between two sentences then the mark and current point are placed around the next sentence.

Mark Paragraph *Editor Command*

Arguments: None
Key sequence: **`Meta+H`**

Puts the mark at the end of the current paragraph and the current point at the start of the current paragraph. The paragraph thereby becomes the current region. If the current point is initially located between two paragraphs, then the mark and current point are placed around the next paragraph.

Mark Whole Buffer *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X H`**

Sets the mark at the end of the current buffer and the current point at the beginning of the current buffer. The current region is thereby set as the whole of the buffer.

A non-nil prefix argument causes the mark to be set as the start of the buffer and the current point at the end.

Records the starting *location* (see “Locations” on page 66).

3.9.2 Regions

Count Words Region *Editor Command*

Arguments: None
Key sequence: None

Displays a count of the total number of words in the region between the current point and the mark.

Count Lines Region *Editor Command*

Arguments: None
Key sequence: None

Displays a count of the total number of lines in the region between the current point and the mark.

region-query-size *Editor Variable*

Default value: 60

If the region between the current point and the mark contains more lines than the value of this editor variable, then any destructive operation on the region prompts the user for confirmation before being executed.

Print Region *Editor Command*

Arguments: None
Key sequence: None

Prints the current region, using **`capi:print-text`**. See the *CAPI User Guide and Reference Manual* for details of this function.

3.10 Locations

A *location* is the position of the current point in a buffer at some time in the past. Locations are recorded automatically by the editor for most commands that take you to a different buffer or where you might lose your place within the current buffer (for example **`Beginning of Buffer`**). They are designed to be a more comprehensive form of the mark ring (see **`Pop and Goto Mark`**), but without the interaction with the selected region.

Go Back *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X C`**

Takes you back to the most recently recorded location. If a prefix argument *count* is supplied, it takes you back *count* locations in the location history. If *count* is negative, it takes you forward again *count* locations in the history, provided that no more locations have been recorded since you last went back.

Select Go Back *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X M`**

Takes you back to a previously recorded location, which you select from a list.

Any prefix argument is ignored.

Go Forward *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X P`**

Takes you back to the next location in the ring of recorded locations. If a prefix argument *count* is supplied, it takes you forward *count* locations in the location history. If *count* is negative, it takes you back *count* locations in the history.

3.11 Deleting and killing text

There are two ways of removing text: deletion, after which the deleted text is not recoverable (except with the **`Undo`** command); and killing, which appends the deleted text to the kill ring, so that it may be recovered using the **`Un-Kill`** and **`Rotate Kill Ring`** commands. The first section contains details of commands to delete text, and the second details of commands to kill text.

Note that, if Delete Selection Mode is active, then any currently selected text is deleted when text is entered. See ‘Delete Selection’ on page 3-76 for details.

The use of prefix arguments with this set of commands can be very useful. In general, using a negative prefix argument repeats these commands a certain number of times in the opposite logical direction. For example, the key sequence **`Ctrl+U 10 Meta+D`** deletes 10 words after the current point, but the key sequence **`Ctrl+U -10 Meta+D`** deletes 10 words *before* the current point.

3.11.1 Deleting Text

Delete Next Character *Editor Command*

Arguments: None
Key sequence: **`Ctrl+D`**,
Key sequence: **`Delete`**

Deletes the character immediately after the current point.

Delete Previous Character *Editor Command*

Arguments: None
Key sequence: **`Backspace`**

Deletes the character immediately before the current point.

Delete Previous Character Expanding Tabs *Editor Command*

Arguments: None
Key sequence: None

Deletes the character immediately before the current point, but if the previous character is a **`Tab`**, then this is expanded into the equivalent number of spaces, so that the apparent space is reduced by one.

A prefix argument deletes the required number of characters, but if any of them are tabs, the equivalent spaces are inserted before the deletion continues.

Delete Horizontal Space *Editor Command*

Arguments: None
Key sequence: `Meta+\`

Deletes all spaces on the line surrounding the current point.

Just One Space *Editor Command*

Arguments: None
Key sequence: `Meta+Space`

Deletes all space on the current line surrounding the current point and then inserts a single space. If there was initially no space around the current point, a single space is inserted.

Delete Blank Lines *Editor Command*

Arguments: None
Key sequence: `Ctrl+X Ctrl+O`

If the current point is on a blank line, all surrounding blank lines are deleted, leaving just one. If the current point is on a non-blank line, all following blank lines up to the next non-blank line are deleted.

Delete Region *Editor Command*

Arguments: None
Key sequence: None

Delete the current region. Also available via `editor:delete-region-command`.

Clear Listener *Editor Command*

Arguments: None
Key sequence: None

Deletes the text in a Listener, leaving you with a prompt. Undo information is not retained, although you are warned about this before confirming the command.

This command is useful if the Listener session has grown very large.

Clear Output *Editor Command*

Arguments: None
Key sequence: None

Deletes the text in the **Output** tab of a Listener or Editor tool, or an Output Browser. Undo information is discarded without warning.

This command is useful if the output has grown very large.

3.11.2 Killing text

Most of these commands result in text being pushed onto the kill ring so that it can be recovered. There is only one kill ring for all buffers so that text can be copied from one buffer to another.

Normally each kill command pushes a new block of text onto the kill ring. However, if more than one kill command is issued sequentially, and the text being killed was next to the previously killed text, they form a single entry in the kill ring (exceptions being **`Kill Region`** and **`Save Region`**).

`Append Next Kill` is different in that it affects where a subsequent killed text is stored in the kill ring, but does not itself modify the kill ring.

Kill Next Word *Editor Command*

Arguments: None
Key sequence: **`Meta+D`**

Kills the rest of the word after the current point. If the current point is between two words, then the next word is killed.

Kill Previous Word *Editor Command*

Arguments: None
Key sequence: **`Meta+Backspace`**

Kills the rest of the word before the current point. If the current point is between two words, then the previous word is killed.

Kill Line *Editor Command*

Arguments: None
Key sequence: **`Ctrl+K`**

Kills the characters from the current point up to the end of the current line. If the line is empty then the line is deleted.

Backward Kill Line *Editor Command*

Arguments: None
Key sequence: None

Kills the characters from the current point to the beginning of the line. If the current point is already at the beginning of the line, the current line is joined to the previous line, with any trailing space on the previous line killed.

Forward Kill Sentence *Editor Command*

Arguments: None
Key sequence: **`Meta+K`**

Kills the text starting from the current point up to the end of the sentence. If the current point is between two sentences, then the whole of the next sentence is killed.

Backward Kill Sentence *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X Backspace`**

Kills the text starting from the current point up to the beginning of the sentence. If the current point is between two sentences, then the whole of the previous sentence is killed.

Kill Region *Editor Command*

Arguments: None
Key sequence: **`Ctrl+W`**

Kills the region between the current point and the mark.

Save Region *Editor Command*

Arguments: None
Key sequence: **`Meta+W`**

Pushes the region between the current point and the mark onto the kill ring without deleting it from the buffer. Text saved in this way can therefore be inserted elsewhere without first being killed.

Append Next Kill *Editor Command*

Arguments: None
Key sequence: **`Meta+Ctrl+W`**

If the next command entered kills any text then this text will be appended to the existing kill text instead of being pushed separately onto the kill ring.

Zap To Char *Editor Command*

Arguments: None
Key sequence: **`Meta+Z`**

Prompts for a character and kills text from the current point to the next occurrence of that character in the current buffer. If a prefix argument *p* is used, then it kills to the *p*'th occurrence. If *p* is negative, then it kills backwards.

An editor error is signaled if the character cannot be found in the buffer.

3.12 Inserting text

This section contains details of commands used to insert text from the kill ring—see "Deleting and killing text" on page 67—and various other commands used to insert text and lines into the buffer.

Un-Kill *Editor Command*

Arguments: None
Key sequence: `Ctrl+Y`

Selects (yanks) the top item in the kill ring (which represents the last piece of text that was killed with a kill command or saved with `Save Region`) and inserts it before the current point. The current point is left at the end of the inserted text, and the mark is automatically set to the beginning of the inserted text.

A prefix argument (`Ctrl+U` *number*) causes the item at position *number* in the ring to be inserted. The order of items on the ring remains unaltered.

Un-Kill As String *Editor Command*

Arguments: None
Key sequence: None

Similar to `Un-Kill`, but inserts the text as a Lisp string, surrounded by double-quotes.

Un-Kill As Filename *Editor Command*

Arguments: None
Key sequence: None

Similar to `Un-Kill`, but inserts the text as a filename, converting any back-slash characters to forward slash so that it does not need to be escaped in a Lisp string.

Rotate Kill Ring *Editor Command*

Arguments: None
Key sequence: **`Meta+Y`**

Replaces the text that has just been un-killed with the item that is next on the kill ring. It is therefore possible to recover text other than that which was most recently killed by typing **`Ctrl+Y`** followed by **`Meta+Y`** the required number of times. If **`Un-Kill`** was not the previous command, an error is signaled.

Note that the ring is only *rotated* and no items are actually deleted from the ring using this command.

A prefix argument causes the kill ring to be rotated the appropriate number of times before the top item is selected.

New Line *Editor Command*

Arguments: None
Key sequence: **`Return`**

Opens a new line before the current point. If the current point is at the start of a line, an empty line is inserted above it. If the current point is in the middle of a line, that line is split. The current point always becomes located on the second of the two lines.

A prefix argument causes the appropriate number of lines to be inserted before the current point.

Open Line *Editor Command*

Arguments: None
Key sequence: **`Ctrl+O`**

Opens a new line after the current point. If the current point is at the start of a line, an empty line is inserted above it. If the current point is in the middle of a line, that line is split. The current point always becomes located on the first of the two lines.

A prefix argument causes the appropriate number of lines to be inserted after the current point.

Quoted Insert *Editor Command*

Arguments: *args*
Key sequence: **`Ctrl+Q &rest`** *args*

`Quoted Insert` is a versatile command allowing you to enter characters which are not accessible directly on your keyboard.

A single argument *key* is inserted into the text literally. This can be used to enter control keys (such as **`Ctrl+L`**) into a buffer as a text string. Note that **`Ctrl`** is represented by ^ and **`Meta`** by `^]`.

You may input a character by entering its Octal Unicode code: press **`Return`** to indicate the end of the code. For example enter

```
Ctrl+Q 4 3 Return
```

to input **`#`**.

If you have specified that **`Alt`** acts as your Meta key, you may find that some useful **`Alt`**-modified keys are not available in the usual way from within LispWorks. For example, **`Alt+3`** gets interpreted as **`Meta-3`**. As a way around this problem, **`Quoted Insert`** temporarily suspends LispWorks editor processing of **`Alt`** as Meta. For example, you can input **`#`** on a UK Macintosh keyboard by entering

```
Ctrl+Q Alt+3
```

Self Insert *Editor Command*

Arguments: None
Key sequence: *key*

`editor:self-insert-command` ***p*** **`&optional`** ***char***

This is the basic command used for inserting each character that is typed. The character to be inserted is *char*. There is no need for the user to use this command explicitly.

Dynamic Completion *Editor Command*

Arguments: None
Key sequence: **`Meta+/`**

Tries to complete the current word, by looking backwards for a word that starts with the same characters as have already been typed. Repeated use of this command makes the search skip to successively previous instances of words beginning with these characters. A prefix argument causes the search to progress forwards rather than backwards. If the buffer is in Lisp mode then completion occurs for Lisp symbols as well as words.

3.13 Delete Selection

When in Delete Selection Mode, commands that insert text into the buffer first delete any selected text. Delete Selection Mode is a global editor setting. It is off by default with Emacs keys, and is on by default when using Mac OS X editor emulation.

Delete Selection Mode *Editor Command*

Arguments: None
Key Sequence: None

Toggles Delete Selection Mode, switching it on if it is currently off, and off if it is currently on.

3.14 Undoing

Commands that modify the text in a buffer can be undone, so that the text reverts to its state before the command was invoked, using **`Undo`**. Details of modifying commands are kept in an undo ring so that previous commands can be undone. The undo ring works like a stack, in that commands are pushed onto the ring and can only be popped off on a "last in first out" basis.

`Un-Kill` can also be used to replace text that has inadvertently been deleted.

Undo *Editor Command*

Arguments: None
Key sequence: **`Ctrl+Shift+_`**

Undoes the last command. If invoked repeatedly, the most recent commands in the editing session are successively undone.

See also: **`Clear Undo`**, **`Toggle Global Simple Undo`**

undo-ring-size *Editor Variable*

Default value: 100

The number of items in the undo ring.

3.15 Case conversion

This section provides details of the commands which allow case conversions on both single words and regions of text. The three general types of case conversion are converting words to uppercase, converting words to lowercase and converting the first letter of words to uppercase.

Lowercase Word *Editor Command*

Arguments: None
Key sequence: **`Meta+L`**

Converts the current word to lowercase, starting from the current point. If the current point is between two words, then the next word is converted.

A negative prefix argument converts the appropriate number of words *before* the current point to lowercase, but leaves the current point where it was.

Uppercase Word *Editor Command*

Arguments: None
Key sequence: **`Meta+U`**

Converts the current word to uppercase, starting from the current point. If the current point is between two words, then the next word is converted.

A negative prefix argument converts the appropriate number of words *before* the current point to uppercase, but leaves the current point where it was.

Capitalize Word *Editor Command*

Arguments: None
Key sequence: `Meta+C`

Converts the current word to lowercase, capitalizing the first character. If the current point is inside a word, the character immediately after the current point is capitalized.

A negative prefix argument capitalizes the appropriate number of words *before* the current point, but leaves the point where it was.

Lowercase Region *Editor Command*

Arguments: None
Key sequence: `Ctrl+X Ctrl+L`

Converts all the characters in the region between the current point and the mark to lowercase.

Uppercase Region *Editor Command*

Arguments: None
Key sequence: `Ctrl+X Ctrl+U`

Converts all the characters in the region between the current point and the mark to uppercase.

Capitalize Region *Editor Command*

Arguments: None
Key sequence: None

Converts all the words in the region between the mark and the current point to lowercase, capitalizing the first character of each word.

3.16 Transposition

This section gives details of commands used to transpose characters, words, lines and regions.

Transpose Characters *Editor Command*

Arguments: None
Key sequence: **`Ctrl+T`**

Transposes the current character with the previous character, and then moves the current point forwards one character.

If this command is issued when the current point is at the end of a line, the two characters to the left of the cursor are transposed.

A positive prefix argument causes the character before the current point to be shifted forwards the required number of places. A negative prefix argument has a similar effect but shifts the character backwards. In both cases the current point remains located after the character which has been moved.

Transpose Words *Editor Command*

Arguments: None
Key sequence: **`Meta+T`**

Transposes the current word with the next word, and then moves the current point forward one word. If the current point is initially located between two words, then the previous word is moved over the next word.

A positive prefix argument causes the current or previous word to be shifted forwards the required number of words. A negative prefix argument has a similar effect but shifts the word backwards. In both cases the current point remains located after the word which has been moved.

Transpose Lines *Editor Command*

Arguments: None
Key sequence: `Ctrl+X Ctrl+T`

Transposes the current line with the previous line, and then moves the current point forward one line.

A positive prefix argument causes the previous line to be shifted forwards the required number of lines. A negative prefix argument has a similar effect but shifts the line backwards. In both cases the current point remains located after the line which has been moved.

A prefix argument of zero transposes the current line and the line containing the mark.

Transpose Regions *Editor Command*

Arguments: None
Key sequence: None

Transposes two regions. One region is delineated by the current point and the mark. The other region is delineated by the next two points on the mark ring. To use this command it is necessary to use `Set Mark` at the beginning and end of one region and at the beginning of the other region, and then move the current point to the end of the second region.

3.17 Overwriting

By default each character that you type is inserted into the text, with the existing characters being shifted as appropriate. In Overwrite mode, each character that you type deletes an existing character in the text.

When in Overwrite mode, a character can be inserted without deleting an existing character by preceding it with `Ctrl+Q`.

Overwrite Mode *Editor Command*

Arguments: None
Key sequence: `Insert`

Switches Overwrite mode on if it is currently off, and off if it is currently on.

With a positive prefix argument, Overwrite mode is turned on. With a zero or negative prefix argument it is turned off. Using prefix arguments with `Overwrite Mode` disregards the current state of the mode.

Self Overwrite *Editor Command*

Arguments: None
Key sequence: *key*

If the current point is in the middle of a line, the next character (that is, the character that is highlighted by the cursor) is replaced with the last character typed. If the current point is at the end of a line, the new character is inserted without removing any other character.

A prefix argument causes the new character to overwrite the relevant number of characters.

This is the command that is invoked when each character is typed in overwrite mode. There is no need for users to invoke this command explicitly.

Overwrite Delete Previous Character *Editor Command*

Arguments: None
Key sequence: None

Replaces the previous character with space, except that tabs and newlines are deleted.

3.18 Indentation

This section contains details of commands used to indent text. Indentation is usually achieved by inserting tab or space characters into the text so as to indent that text a predefined number of spaces.

The effect of the editor indentation commands depends on the major mode of the buffer. Where relevant, the command details given below provide information on how they operate in Text mode and Lisp mode. The operation of commands in Fundamental mode is generally the same as that of Text mode.

Indent *Editor Command*

Arguments: None
Key sequence: **`Tab`**

In Text mode, **`spaces-for-tab #\Space`** characters are inserted. A prefix argument causes this to occur at the start of the appropriate number of lines (starting from the current line).

In Lisp mode, the current line is indented according to the structure of the current Lisp form. A prefix argument *p* causes *p* lines to be indented according to Lisp syntax.

See **`editor:*indent-with-tabs*`** for control over the insertion of **`#\Tab`** characters by this and other indentation commands.

Note: the key sequence **`Tab`** is overridden in Lisp mode to perform **`Indent Selection or Complete Symbol`**.

spaces-for-tab *Editor Variable*

Default value: 8

Determines the width of the whitespace (that is, the number of **`#\Space`** characters) used to display a **`#\Tab`** character.

Indent Region *Editor Command*

Arguments: None
Key sequence: **`Meta+Ctrl+\`**

Indents all the text in the region between the mark and the current point.

In Text mode a block of whitespace, which is **`spaces-for-tab`** wide, is inserted at the start of each line within the region.

In Lisp mode the text is indented according to the syntax of the Lisp form.

In both cases, a prefix argument causes any existing indentation to be deleted and replaced with a block of whitespace of the appropriate width.

Indent Rigidly *Editor Command*

Arguments: None
Key sequence: `Ctrl+X Tab` or `Ctrl+X Ctrl+I`

Indents each line in the region between the current point and the mark by a block of whitespace which is `spaces-for-tab` wide. Any existing whitespace at the beginning of the lines is retained.

A positive prefix argument causes the lines to be indented by the appropriate number of spaces, in addition to their existing space. A negative prefix argument causes the lines to be shifted to the left by the appropriate number of spaces. Where necessary, tabs are converted to spaces.

Indent Selection *Editor Command*

Arguments: None
Key sequence: None

Indents all the text in the selection or the current line if there is no selection. With a prefix argument *p*, any existing indentation is deleted and replaced with a block of space *p* columns wide.

See also **`Indent Selection or Complete Symbol`**.

Delete Indentation *Editor Command*

Arguments: None
Key sequence: `Meta+Shift+^`

Joins the current line with the previous one, deleting all whitespace at the beginning of the current line and at the end of the previous line. The deleted whitespace is normally replaced with a single space. However, if the deleted whitespace is at the beginning of a line, or immediately after a `(`, or immediately before a `)`, then the whitespace is merely deleted without any characters being inserted. If the preceding character is a sentence terminator, then two spaces are left instead of one.

A prefix argument causes the following line to be joined with the current line.

Back to Indentation *Editor Command*

Arguments: None
Key sequence: **`Meta+M`**

Moves the current point to the first character in the current line that is not a whitespace character.

Indent New Line *Editor Command*

Arguments: None
Key sequence: None

Moves everything to the right of the current point to a new line and indents it. Any whitespace before the current point is deleted. If there is a **`fill-prefix`**, this is inserted at the start of the new line instead.

A prefix argument causes the current point to be moved down the appropriate number of lines and indented.

Quote Tab *Editor Command*

Arguments: None
Key sequence: None

Inserts a **`Tab`** character.

A prefix argument causes the appropriate number of tab characters to be inserted.

3.19 Filling

Filling involves re-formatting text so that each line extends as far to the right as possible without any words being broken or any text extending past the **`fill-column`**.

The first section deals with general commands used to fill text, while the second section provides information on Auto-Fill mode and related commands.

3.19.1 Fill commands

Fill Paragraph *Editor Command*

Arguments: None
Key sequence: **`Meta+Q`**

Fills the current paragraph. If the current point is located between two paragraphs, the next paragraph is filled.

A prefix argument causes the current fill operation to use that value, rather than the value of **`fill-column`**.

Fill Region *Editor Command*

Arguments: None
Key sequence: **`Meta+G`**

Fills the region from the current point to the mark.

A prefix argument causes the current fill operation to use that value, rather than the value of **`fill-column`**.

fill-column *Editor Variable*

Default value: 70

Determines the column at which text in the current buffer is forced on to a new line when filling text.

Set Fill Column *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X F`**

Sets the value of **`fill-column`**, for the current buffer, as the column of the current point.

A prefix argument causes **`fill-column`** to be set at the required value.

fill-prefix *Editor Variable*

Default value: **`nil`**

Defines a string which is excluded when each line of the current buffer is re-formatted using the filling commands. For example, if the value is ";;", then these characters at the start of a line are skipped over when the text is re-formatted. This allows you to re-format (fill) Lisp comments. If the value is **`nil`**, no characters are excluded when text is filled.

If the vales is non-nil, any line that does not begin with the value is considered to begin a new paragraph. Therefore, any re-formatting of comments in Lisp code does not intrude outside the commented lines.

Set Fill Prefix *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X .`**

Sets the **`fill-prefix`** of the current buffer to be the text from the beginning of the current line up to the current point. The **`fill-prefix`** may be set to **`nil`** by using this command with the current point at the start of a line.

Center Line *Editor Command*

Arguments: None
Key sequence: None

Centers the current line with reference to the current value of **`fill-column`**.

A prefix argument causes the current line to be centered with reference to the required width.

3.19.2 Auto-Fill mode

By default no filling of text takes place unless specified by using one of the commands described above. A result of this is that the user has to press **`Return`** at the end of each line typed to simulate filling. In Auto-Fill mode lines are broken between words at the right margin automatically as the text is being typed. Each line is broken when a space is inserted, and the text that extends past the right margin is put on the next line. The right hand margin is determined by the editor variable **`fill-column`**.

Auto Fill Mode *Editor Command*

Arguments: None
Key sequence: None

Switches auto-fill mode on if it is currently off, and off if it is currently on.

With a positive prefix argument, auto-fill mode is switched on. With a negative or zero prefix argument, it is switched off. Using prefix arguments with **`Auto Fill Mode`** disregards the current state of the mode.

Auto Fill Space *Editor Command*

Arguments: None
Key sequence: **`Space`**
Mode: Auto-Fill

Inserts a space and breaks the line between two words if the line extends beyond the right margin. A fill prefix is automatically added at the beginning of the new line if the value of **`fill-prefix`** is non-nil.

When **`Space`** is bound to this command in Auto-Fill mode, this key no longer invokes **`Self Insert`**.

A positive prefix argument causes the required number of spaces to be inserted but no line break. A prefix argument of zero causes a line break, if necessary, but no spaces are inserted.

Auto Fill Linefeed *Editor Command*

Arguments: None
Key sequence: `Linefeed`
Mode: Auto-Fill

Inserts a `Linefeed` and a `fill-prefix` (if one exists).

Auto Fill Return *Editor Command*

Arguments: None
Key sequence: `Return`
Mode: Auto-Fill

The current line is broken, between two words if necessary, with no Space being inserted. This is equivalent to `Auto Fill Space` with a zero prefix argument, but followed by a newline.

auto-fill-space-indent *Editor Variable*

Default value: `nil`

When true, Auto-fill commands use `Indent New Comment Line` to break lines instead of `New Line`.

3.20 Buffers

This section contains details of commands used to manipulate buffers.

Select Buffer *Editor Command*

Arguments: *buffer-name*
Key sequence: `Ctrl+X B` *buffer-name*

Displays a buffer called *buffer-name* in the current window. If no buffer name is provided, the last buffer accessed in the current window is displayed. If the buffer that is selected is already being displayed in another window, any modifications to that buffer are shown simultaneously in both windows.

Select Buffer Other Window *Editor Command*

Arguments: *buffer-name*
Key sequence: None

Displays a buffer called *buffer-name* in a new window. If no buffer name is provided, the last buffer displayed in the current window is selected. If the buffer that is selected is already being displayed in another window, any modifications to that buffer are shown simultaneously in both windows.

Select Previous Buffer *Editor Command*

Arguments: None
Key sequence: **`Meta+Ctrl+L`**

Displays the last buffer accessed in a new window. If the buffer that is selected is already being displayed in another window, any modifications to that buffer are shown simultaneously in both windows.

A prefix argument causes the appropriately numbered buffer, from the top of the buffer history, to be selected.

Circulate Buffers *Editor Command*

Arguments: None
Key sequence: **`Meta+Ctrl+Shift+L`**

Move through the buffer history, selecting the successive previous buffers.

Bury Buffer *Editor Command*

Arguments: *buffer*
Key sequence: None

The command **`Bury Buffer`** puts the buffer *buffer*, which defaults to the current buffer, at the end of the buffer list. If the buffer is visible in the current window, it is replaced by the previously selected buffer.

Edit Buffer *Editor Command*

Arguments: *buffer-name*
Key sequence: None

The command **`Edit Buffer`** displays a buffer *buffer-name*, either in the current window if it is suitable, or a suitable window.

Note: windows such as the **Output** tab of the Editor tool are marked internally as not suitable for displaying arbitrary buffers. If **`Edit Buffer`** is invoked when the current window is marked, it finds another window to display the buffer. In contrast, **`Select Buffer`** will signal an editor error in this case.

Kill Buffer *Editor Command*

Arguments: *buffer-name*
Key sequence: **`Ctrl+X K`** *buffer-name*

`editor:kill-buffer-command` ***p*** **`&optional`** ***buffer-name***

Deletes a buffer called *buffer-name*. If no buffer name is provided, the current buffer is deleted. If the buffer that is selected for deletion has been modified then confirmation is asked for before deletion takes place.

List Buffers *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X Ctrl+B`**

Displays a list of all the existing buffers in the Buffers window in the Editor tool. Information shown includes the name of the buffer, its major mode, whether it has been modified or not, the pathname of any file it is associated with, and its size.

A buffer can be selected by clicking the left mouse button on the buffer name. The buttons on the toolbar can then be used to modify the selected buffer.

Create Buffer *Editor Command*

Arguments: *buffer-name*
Key sequence: None

`editor:create-buffer-command` ***p*** **`&optional`** ***buffer-name***

Creates a buffer called *buffer-name*. If no buffer name is provided then the current buffer is selected. If a buffer with the specified name already exists then this becomes the current buffer instead, and no new buffer is created.

New Buffer *Editor Command*

Arguments: None
Key sequence: None

Creates a new unnamed buffer. The buffer is in Lisp mode.

default-buffer-element-type *Editor Variable*

Default value: **`cl:character`**

The character element type used when a new buffer is created, for example by **`New Buffer`**.

Insert Buffer *Editor Command*

Arguments: *buffer-name*
Key sequence: None

Inserts the contents of a buffer called *buffer-name* at the current point. If no buffer name is provided, the contents of the last buffer displayed in the current window are inserted.

Rename Buffer *Editor Command*

Arguments: *new-name*
Key sequence: None

Changes the name of the current buffer to *new-name*.

Toggle Buffer Read-Only *Editor Command*

Arguments: None
Key sequence: `Ctrl+X Ctrl+Q`

Makes the current buffer read only, so that no modification to its contents are allowed. If it is already read only, this restriction is removed.

Set Buffer Transient Edit *Editor Command*

Arguments: None
Key sequence: None

The command `Set Buffer Transient Edit` makes the current buffer writable, and disables auto-saving.

Check Buffer Modified *Editor Command*

Arguments: None
Key sequence: `Ctrl+X Shift+~`

Checks whether the current buffer is modified or not.

Buffer Not Modified *Editor Command*

Arguments: None
Key sequence: `Meta+Shift+~`

`editor:buffer-not-modified-command` ***p*** **`&optional`** ***buffer***

Makes the current buffer not modified.

The argument *buffer* is the name of the buffer to be un-modified. The default is the current buffer.

Print Buffer *Editor Command*

Arguments: None
Key sequence: None

The command **`Print Buffer`** prints the current buffer, by calling **`capi:print-dialog`** to select a printer and then **`capi:print-text`** with the appropriate arguments to print the buffer.

See the *CAPI User Guide and Reference Manual* for details of these functions.

3.21 Windows

This section contains details of commands used to manipulate windows. A window ring is used to hold details of all windows currently open.

New Window *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X 2`**

Creates a new window and makes it the current window. Initially, the new window displays the same buffer as the current one.

Next Window *Editor Command*

Arguments: None
Key sequence: None

Changes the current window to be the next window in the window ring, and the current buffer to be the buffer that is displayed in that window.

Next Ordinary Window *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X O`**

Changes the current window to be the next ordinary editor window, thus avoiding the need to cycle through other window types (for example, Listeners and Debuggers).

Previous Window *Editor Command*

Arguments: None
Key sequence: None

Changes the current window to be the previous window visited, and the current buffer to be the buffer that is displayed in that window.

Delete Window *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X 0`**

Deletes the current window. The previous window becomes the current window.

Delete Next Window *Editor Command*

Arguments: None
Key sequence: None

Deletes the next window in the window ring.

Delete Other Windows *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X 1`**

The command **`Delete Other Windows`** deletes (that is, closes) all other windows inside the same interface. Applicable only inside the LispWorks IDE Editor tool.

See also: **`Delete Next Window`**

Previous Focus Window *Editor Command*

Arguments: None
Key sequence: None

The command **`Previous Focus Window`** switches to the editor pane that previously had the input focus.

Scroll Next Window Down *Editor Command*

Arguments: None
Key sequence: None

The next window in the window ring is scrolled down.

A prefix argument causes the appropriately numbered window, from the top of the window ring, to be scrolled.

Scroll Next Window Up *Editor Command*

Arguments: None
Key sequence: None

The next window in the window ring is scrolled up.

A prefix argument causes the appropriately numbered window, from the top of the window ring, to be scrolled.

Split Window Horizontally *Editor Command*

Arguments: None
Key sequence: `Ctrl+X 5`

Split the current window horizontally, adding a window to the left of the current window or to the right if given a prefix argument. The new window will display the current buffer initially.

Split Window Vertically *Editor Command*

Arguments: None
Key sequence: `Ctrl+X 6`

Split the current window vertically, adding a window above the current window or below if given a prefix argument. The new window will display the current buffer initially.

Unsplit Window *Editor Command*

Arguments: None
Key sequence: `Ctrl+X 7`

Remove another window in the same split column or row. A prefix argument causes all other windows in the same top level windows to be removed. When invoked without a prefix, the next window is removed if there is one, otherwise the previous window is removed.

Toggle Count Newlines *Editor Command*

Arguments: None
Key sequence: None

Controls the size of the scroller in editor-based tools, and how the Editor tool's mode line represents the extent of the displayed part of the buffer.

`Toggle Count Newlines` switches between counting newlines and counting characters in the current buffer. The counting determines what is displayed in the Editor tool's mode line, and how the size of the scroller is computed.

When counting newlines, the mode line shows line numbers and the total number of lines:

***StartLine*-*EndLine* [*TotalLine*]**

When counting characters, the mode line shows percentages based on the characters displayed compared to the total number of characters in the buffer:

***PercentStart*-*PercentEnd*%**

The default behavior is counting newlines, except for very large buffers.

Refresh Screen *Editor Command*

Arguments: None
Key sequence: `Ctrl+L`

Moves the current line to the center of the current window, and then re-displays all the text in all the windows.

A prefix argument of 0 causes the current line to become located at the top of the window. A positive prefix argument causes the current line to become located the appropriate number of lines from the top of the window. A negative prefix argument causes the current line to become located the appropriate number of lines from the bottom of the window.

3.22 Pages

Files are sometimes thought of as being divided into pages. For example, when a file is printed on a printer, it is divided into pages so that each page appears on a fresh piece of paper. The ASCII key sequence `Ctrl+L` constitutes a page delimiter (as it starts a new page on most line printers). A page is the region between two page delimiters. A page delimiter can be inserted into text being edited by using the editor command `Quoted Insert` (that is, type in `Ctrl+Q Ctrl+L`).

Previous Page *Editor Command*

Arguments: None
Key sequence: `Ctrl+X [`

Moves the current point to the start of the current page.

A prefix argument causes the current point to be moved backwards the appropriate number of pages.

Next Page *Editor Command*

Arguments: None
Key sequence: `Ctrl+X ]`

Moves the current point to the start of the next page.

A prefix argument causes the current point to be moved forwards the appropriate number of pages.

Goto Page *Editor Command*

Arguments: None
Key sequence: None

Moves the current point to the start of the next page.

A positive prefix argument causes the current point to be moved to the appropriate page starting from the beginning of the buffer. A negative prefix argument causes the current point to be moved back the appropriate number of pages from the current location. A prefix argument of zero causes the user to be prompted for a string, and the current point is moved to the next page with that string contained in the page title.

Records the starting *location* (see “Locations” on page 66).

Mark Page *Editor Command*

Arguments: None
Key sequence: `Ctrl+X Ctrl+P`

Puts the mark at the end of the current page and the current point at the start of the current page. The page thereby becomes the current region.

A prefix argument marks the page which is the appropriate number of pages on from the current one.

Count Lines Page *Editor Command*

Arguments: None
Key sequence: `Ctrl+X L`

Displays the number of lines in the current page and the location of the current point within the page.

A prefix argument displays the total number of lines in the current buffer and the location of the current point within the buffer (so that page delimiters are ignored).

View Page Directory *Editor Command*

Arguments: None
Key sequence: None

Displays a list of the first non-blank line after each page delimiter.

Insert Page Directory *Editor Command*

Arguments: None
Key sequence: None

Inserts a listing of the first non-blank line after each page delimiter at the start of the buffer, moving the current point to the end of this list. The location of the start of this list is pushed onto the mark ring.

A prefix argument causes the page directory to be inserted at the current point.

3.23 Searching and replacing

This section is divided into three parts. The first two provide details of commands used for searching. These commands are, on the whole, non-modifying and non-destructive, and merely search for strings and patterns. The third part provides details of commands used for replacing a string or pattern.

3.23.1 Searching

Most of the search commands perform straightforward searches, but there are two useful commands (**`Incremental Search`** and **`Reverse Incremental Search`**) which perform incremental searches. This means that the search is started as soon as the first character is typed.

Incremental Search *Editor Command*

Arguments: *string*
Key sequence: **`Ctrl+S`** *string*

Searches forward, starting from the current point, for the search string that is input, beginning the search as soon as each character is typed in. When a match is found for the search string, the current point is moved to the end of the matched string. If the search string is not found between the current point and the end of the buffer, an error is signaled.

The search result is highlighted. You can change the style of the highlighting in the LispWorks IDE by **Preferences... > Environment > Styles > Colors and Attributes > Search Match**.

`Incremental Search` records the starting *location* (see "Locations" on page 66).

With a prefix argument *p* the matches are displayed at a fixed line position, *p* lines below the top of the window. Otherwise, the position of the matched string within the window is influenced by the editor variable `incremental-search-minimum-visible-lines`.

For example, to display successive definitions one line from the top of the text view of the Editor window, enter:

```
Ctrl+U 2 Ctrl+S ( d e f Ctrl+S Ctrl+S
```

All incremental searches can be controlled by entering one of the following key sequences at any time during the search.

`Ctrl+S`	If the search string is empty, repeats the last incremental search, otherwise repeats a forward search for the current search string. If the search string cannot be found, starts the search from the beginning of the buffer (wrap-around search).
`Ctrl+R`	Changes to a backward (reverse) search.
`Delete`	Cancels the last character typed.
`Ctrl+Q`	Quotes the next character typed.
`Ctrl+W`	Adds the next word under the cursor to the search string.
`Meta+Ctrl+Y`	Adds the next form under the cursor to the search string.

`Ctrl+Y`	Adds the remainder of the line under the cursor to the search string.
`Meta+Y`	Adds the current kill string to the search string.
`Ctrl+C`	Add the editor window's selected text to the search string.
`Esc`	If the search string is empty, invokes a non-incremental search, otherwise exits the search, leaving the current point at the last find.
`Ctrl+G`	Aborts the search, returning the current point to its original location. If the search string cannot be found, cancels the last character typed (equivalent to `Delete`).
`Return`	Exits the search, leaving the current point at the last find.

incremental-search-minimum-visible-lines *Editor Variable*

Default value: 3

Determines the minimum of visible lines between an incremental search match and the closest window border (top or bottom). If the point is closer to the border than the value, the point is scrolled to the center of the window.

Lines are counted from the start of the match, and the line where the match starts is included in the count.

Note that this has no effect when doing "fixed position" search (with numeric prefix, for example `Ctrl+U` ***digit*** `Ctrl+S`, or if the window is too short.

Setting the value to 0 makes incremental searching behave as in LispWorks 6.0 and earlier versions, that is the match can be shown on the top or bottom line currently displayed in the window.

Reverse Incremental Search *Editor Command*

Arguments: *string*
Key sequence: `Ctrl+R` *string*

Searches backward, starting from the current point, for the search string that is input, beginning the search as soon as each character is provided. When a match is found for the search string, the current point is moved to the start of the matched string. If the search string is not found between the current point and the beginning of the buffer, an error is signaled.

You can use a fixed line position for the matches and/or modify the style used to display them, as described for **`Incremental Search`**.

With a prefix argument *p* the matches are displayed at a fixed line position, *p* lines below the top of the window. Otherwise, the position of the matched string within the window is influenced by the editor variable **`incremental-search-minimum-visible-lines`**.

The search can be controlled by entering one of the following key sequences at any time during the search.

`Ctrl+R`	If the search string is empty, repeats the last incremental search, otherwise repeats a backward search for the current search string. If the search string cannot be found, starts the search from the end of the buffer (wrap-around search).
`Ctrl+S`	Changes to a forward search.
`Delete`	Cancels the last character typed.
`Esc`	If the search string is empty, invokes a non-incremental search, otherwise exits the search, leaving the current point at the last find.
`Ctrl+G`	Aborts the search, returning the current point to its original location. If the search string cannot be found, cancels the last character typed (equivalent to **`Delete`**).

`Ctrl+Q`	Quotes the next character typed.

Forward Search *Editor Command*

Arguments: *string*
Key sequence: `Ctrl+S Esc` *string*

`editor:forward-search-command` ***p*** `&optional` ***string the-point***

The default for *the-point* is the current point.

Searches forwards from *the-point* for *string*. When a match is found, *the-point* is moved to the end of the matched string. In contrast with `Incremental Search`, the search string must be terminated with a carriage return before any searching is done. If an empty string is provided, the last search is repeated.

Records the starting *location* (see “Locations” on page 66).

Backward Search *Editor Command*

Arguments: *string*
Key sequence: None

`editor:reverse-search-command` ***p*** `&optional` ***string the-point***

The default for *the-point* is the current point.

Searches backwards from *the-point* for *string*. When a match is found, *the-point* is moved to the start of the matched string. In contrast with `Reverse Incremental Search`, the search string must be terminated with a carriage return before any searching is done. If an empty string is provided, the last search is repeated.

Records the starting *location* (see “Locations” on page 66).

`Reverse Search` is a synonym for `Backward Search`.

List Matching Lines *Editor Command*

Arguments: *string*
Key sequence: None

editor:list-matching-lines-command *p* &optional *string*

Lists all lines after the current point that contain *string*, in a Matches window.

A prefix argument causes the appropriate number of lines before and after each matching line to be listed also.

Delete Matching Lines *Editor Command*

Arguments: *string*
Key sequence: None

editor:delete-matching-lines-command *p* &optional *string*

Deletes all lines after the current point that match *string*.

Delete Non-Matching Lines *Editor Command*

Arguments: *string*
Key sequence: None

editor:delete-non-matching-lines-command *p* &optional *string*

Deletes all lines after the current point that do not match *string*.

Search All Buffers *Editor Command*

Arguments: *string*
Key sequence: None

Searches all the buffers for *string*. If only one buffer contains *string*, it becomes the current buffer, with the cursor positioned at the start of the string. If more than one buffer contains the string, a popup window displays a list of those buffers. A buffer may then be selected from this list.

Buffers Search *Editor Command*

Arguments: *search-string*
Key sequence: None

The command **`Buffers Search`** searches all opened buffers for *search-string*, displaying the first match.

Use the key sequence **`Meta+,`** to find subsequent occurrences of *search-string*.

Search Buffers *Editor Command*

Arguments: *search-string*
Key sequence: None

The command **`Search Buffers`** searches all opened buffers using the Search Files tool.

It prompts for a string and then activates the Search Files tool in the **Opened Buffers** mode. See the *LispWorks IDE User Guide* for a description of the Search Files tool.

Directory Search *Editor Command*

Arguments: *directory string*
Key sequence: None

Searches source files in *directory* for *string*. The current working directory is offered as a default for *directory*.

By default only files with suffix **`.lisp`**, **`.lsp`**, **`.c`**, **`.cpp`** or **`.h`** are searched. A non-nil prefix argument causes all files to be searched, except for those ending with one of the strings in the list **`system:*ignorable-file-suffices*`**.

`Directory Search` displays the first match. Use the key sequence **`Meta+,`** to find subsequent occurrences of the search string.

Search Files *Editor Command*

Arguments: *search-string directory*
Key sequence: **`Ctrl+X *`** *search-string directory*

Searches for a string in a directory using a Search Files tool.

The command prompts for *search-string* and *directory* and then raises a Search Files tool. The configuration of the Search Files tool controls which files in the directory are searched. If the search string is not empty, it starts searching automatically, unless a prefix argument is given.

See the *LispWorks IDE User Guide* for a description of the Search Files tool.

Search Files Matching Patterns *Editor Command*

Arguments: *search-string directory patterns*
Key sequence: `Ctrl+X &` *search-string directory patterns*

Searches for a string in files under a directory with names matching given patterns, using a Search Files tool.

The command prompts for *search-string, directory* and *patterns,* and raises a Search Files tool in Roots and Patterns mode. If the search string is not empty, it starts searching automatically, unless a prefix argument is given.

patterns should be a comma-separated set of filename patterns delimited by braces. A pattern where the last component does not contain `*` is assumed to be a directory onto which the Search Files tool adds its own filename pattern. *patterns* defaults to `{*.lisp,*.lsp,*.c,*.h}`.

See the *LispWorks IDE User Guide* for a description of the Search Files tool.

System Search *Editor Command*

Arguments: *system string*
Key sequence: None

Searches the files of *system* for *string*.

Matches are shown in editor buffers consecutively. Use the key sequence `Meta+,` to find subsequent definitions of the search string.

Search System *Editor Command*

Arguments: *search-string system*
Key sequence: None

Prompts for *search-string* and *system* and then raises a Search Files tool in System Search mode, which displays the search results and allows you to visit the files.

See the *LispWorks IDE User Guide* for a description of the Search Files tool.

default-search-kind *Editor Variable*

Default value: **`:string-insensitive`**

Defines the default method of searching. By default, all searching (including regexp searching, and replacing commands) ignores case. If you want searching to be case-sensitive, the value of this variable should be set to **`:string-sensitive`** using **`Set Variable`**.

It is also possible to search a set of files programmatically using the **`search-files`** function:

editor:search-files *Function*

`editor:search-files &key` ***string files generator*** **`=> nil`**

`search-files` searches all the files in a list for a given string.

string is a string to search for (prompted if not given).

files is a list of pathnames of files to search, and *generator* is a function to generate the files if none are supplied.

If a match is found the file is displayed in a buffer with the cursor on the occurrence. **`Meta+-,`** makes the search continue until the next occurrence.

`search-files` returns **`nil`**.

For example:

```
(editor:search-files
 :files '(".login" ".cshrc")
 :string "alias")
```

3.23.2 Regular expression syntax

A regular expression (*regexp*) allows the specification of the search string to include wild characters, repeated characters, ranges of characters, and alternatives. Strings which follow a specific pattern can be located, which makes regular expression searches very powerful.

The regular expression syntax used is similar to that of Emacs. In addition to ordinary characters, a regular expression can contain the following special characters to produce the search pattern:

`.`	Matches any single character except a newline. For example, `c.r` matches any three character string starting with `c` and ending with `r`.
`*`	Matches the previous regexp any number of times (including 0 times). For example, `ca*r` matches strings beginning with `c` and ending with `r`, with any number of `a`'s in-between. An empty regexp followed by `*` matches an empty part of the input. By extension, `^*` will match exactly what `^` matches.
`+`	Matches the previous regexp any number of times, but at least once. For example, `ca+r` matches strings beginning with `c` and ending with `r`, with at least one `a` in-between. An empty regexp followed by `+` matches an empty part of the input.
`?`	Matches the previous regexp either 0 or 1 times. For example, `ca?r` matches either the string `cr` or `car`, and nothing else. An empty regexp followed by `?` matches an empty part of the input.
`^`	Matches the next regexp as long as it is at the beginning of a line. For example, `^foo` matches the string `foo` as long as it is at the beginning of a line.
`$`	Matches the previous regexp as long as it is at the end of a line. For example, `foo$` matches the string `foo` as long as it is at the end of a line.

`[ ]`	Contains a character set to be used for matching, where the other special characters mentioned do not apply. The empty string is automatically part of the character set. For example, `[a.b]` matches either `a` or `.` or `b` or the empty string. The regexp `c[ad]*r` matches strings beginning with `c` and ending with `r`, with any number of `a`'s and `d`'s in-between. The characters `-` and `^` have special meanings inside character sets. `-` defines a range and `^` defines a complement character set. For example, `[a-d]` matches any character in the range `a` to `d` inclusive. `[^ab]` matches any character except `a` or `b`.
`\`	Quotes the special characters. For example, `\*` matches the character `*` (that is, `*` has lost its special meaning).
`|`	Specifies an alternative. For example, `ab|cd` matches either `ab` or `cd`.
`\(, \)`	Provides a grouping construct. For example, `ab\(cd|ef\)` matches either `abcd` or `abef`.

Regexp Forward Search

Regexp Reverse Search

Editor Commands

Arguments: *string*
Key sequence: None

Performs a forward or backward search for *string* using regular expressions. The search pattern must be terminated with a carriage return before any searching is done. If an empty string is provided, the last regexp search is repeated.

See also: `editor:regular-expression-search`

ISearch Forward Regexp *Editor Command*

Arguments: *string*
Key sequence: `Meta+Ctrl+S` *string*

The command `ISearch Forward Regexp` performs incremental search forwards, using regular expression matching.

ISearch Backward Regexp *Editor Command*

Arguments: *string*
Key sequence: `Meta+Ctrl+R` *string*

The command `ISearch Backward Regexp` performs incremental search backwards, using regular expression matching.

Count Occurrences *Editor Command*

Arguments: None
Default binding: None

`editor:count-occurrences-command` ***p*** `&optional` ***regexp***

Counts the number of regular expression matches for the string *regexp* between the current point and the end of the buffer.

`Count Matches` is a synonym for `Count Occurrences`.

3.23.3 Replacement

Replace String *Editor Command*

Arguments: *target replacement*
Key sequence: None

`editor:replace-string-command` ***p*** `&optional` ***target replacement***

Replaces all occurrences of *target* string by *replacement* string, starting from the current point.

Whenever *replacement* is substituted for *target*, case may be preserved, depending on the value of the editor variable `case-replace`.

Query Replace *Editor Command*

Arguments: *target replacement*
Key sequence: **`Meta+Shift+%`** *target replacement*

`editor:query-replace-command` ***p*** **`&optional`** ***target replacement***

Replaces occurrences of *target* string by *replacement* string, starting from the current point, but only after querying the user. Each time *target* is found, an action must be indicated from the keyboard.

Whenever *replacement* is substituted for *target*, case may be preserved, depending on the value of the editor variable **`case-replace`**.

The following key sequences are used to control **`Query Replace`**:

`Space` or **`y`**	Replace *target* by *replacement* and move to the next occurrence of *target*.
`Delete`	Skip *target* without replacing it and move to the next occurrence of *target*.
`.`	Replace *target* by *replacement* and then exit.
`!`	Replace all subsequent occurrences of *target* by *replacement* without prompting.
`Ctrl+R`	Enter recursive edit. This allows the current occurrence of *target* to be edited. When this editing is completed, **`Exit Recursive Edit`** should be invoked. The next instance of *target* is then found.
`Esc`	Quit from **`Query Replace`** with no further replacements.

Directory Query Replace *Editor Command*

Arguments: *directory target replacement*
Key sequence: None

Replaces occurrences of *target* string by *replacement* string for each source file in *directory*, but only after querying the user.

The current working directory is offered as a default for *directory*.

By default only files with suffix `.lisp`, `.lsp`, `.c`, `.cpp` or `.h` are searched. A non-nil prefix argument causes all files to be searched, except for those ending with one of the strings in the list `system:*ignorable-file-suffices*`.

Each time *target* is found, an action must be indicated from the keyboard. For details of possible actions see **`Query Replace`**.

System Query Replace *Editor Command*

Arguments: *system target replacement*
Key sequence: None

Replaces occurrences of *target* string by *replacement* string, for each file in *system*, but only after querying the user. Each time *target* is found, an action must be indicated from the keyboard. For details of possible actions see **`Query Replace`**.

Buffers Query Replace *Editor Command*

Arguments: *target replacement*
Key sequence: None

The command **`Buffers Query Replace`** does a query replace operation on all opened buffers. See **`Query Replace`** for details of the operation.

case-replace *Editor Variable*

Default value: `t`

If the value of this variable is `t`, **`Replace String`** and **`Query Replace`** try to preserve case when doing replacements. If its value is `nil`, the case of the replacement string is as defined by the user.

Replace Regexp

Query Replace Regexp *Editor Commands*

Arguments: *target replacement*
Key sequence: None

`editor:replace-regexp-command` ***p*** `&optional` ***target replacement***

`editor:query-replace-regexp-command` ***p*** `&optional` ***target replacement***

Replaces matches of *target* regular expression by *replacement* string, starting from the current point.

See “Regular expression syntax” for a description of regular expressions.

`Replace Regexp` replaces all matches.

`Query Replace Regexp` asks the user whether to replace each match in turn. Each time *target* is matched, an action must be indicated from the keyboard.

See “Regular expression syntax” for a description of regular expressions, and **`Query Replace`** for the keyboard gestures available.

In *replacement*, the Backslash character `\` has special meaning. It causes the pair of characters of the Backslash and the following character *char* to be replaced for each match before using *replacement*. The character *char* following the Backslash must be one of:

`&`	Use the string that matched the whole pattern.
`#`	Use a string that is the decimal representation of the number of matches that have already been replaced in the current operation (first one will use 0).
`\`	Use the single character string `"\\"`.
A non-zero digit	Use the string that matched the corresponding `\(` and `\)` pair in the pattern, starting from 1. The pairs are counted by the order of appearance of the `\(` in the pattern, so nested pairs have larger numbers than their enclosing pairs.

When *replacement* contains a `\` character, after each match the system replaces all the pairs of `\`*char* by the appropriate string, and uses the result as the replacement string for this match. For example, you can change dates in the form `dd/mm/yyyy` to the form `yyyy-mm-dd` by using

target	`\([0-9][0-9]\)/\([0-9][0-9]\)/\([0-9][0-9][0-9][0-9]\)`
replacement	`\3-\2-\1`

This replaces, for example, `12/03/1979` by `1979-03-12`.

Compatibility note: the special meaning of the Backslash character `\` was introduced in LispWorks 7.0.

3.24 Comparison

This section describes commands which compare files, windows and/or buffers against each other.

Compare Windows *Editor Command*

Arguments: *source1 source2*
Key sequence: None

Compares the text in the current window with the text of another window. The points are left where the text differs.

source1 defaults to the current window. *source2* defaults to the next ordinary window.

Differences in whitespace are ignored by default, according to the value of `compare-ignores-whitespace`.

Compare Buffers *Editor Command*

Arguments: *buffer1 buffer2*
Key sequence: None

Compares the text in the current buffer with that another buffer.

The first argument defaults to the current buffer. The second defaults to the next editor buffer.

Differences in whitespace are ignored by default, according to the value of **`compare-ignores-whitespace`**.

Compare File And Buffer *Editor Command*

Arguments: None
Key sequence: None

The command **`Compare File And Buffer`** compares the text in the buffer with the text in the associated file, which is displayed in another window if the text differs. The points are left where the texts differ.

If the buffer is not associated with a file, **`editor:editor-error`** is called.

compare-ignores-whitespace *Editor Variable*

Initial value: **`t`**

When true, the **`Compare Windows`** and **`Compare Buffers`** commands ignore mismatches due to differences in whitespace.

Diff *Editor Command*

Arguments: *file1 file2*
Key sequence: None

Compares the current buffer with another file.

A prefix argument makes it compare any two files, prompting you for both filenames.

Diff Ignoring Whitespace *Editor Command*

Arguments: *file1 file2*
Key sequence: None

Compares the current buffer with another file, like **`Diff`** but ignoring whitespace.

A prefix argument is interpreted in the same way as by **`Diff`**.

3.25 Registers

Locations and regions can be saved in *registers*. Each register has a name, and reference to a previously saved register is by means of its name. The name of a register, which consists of a single character, is case-insensitive.

Point to Register *Editor Command*

Arguments: *name*
Key sequence: **`Ctrl+X /`** *name*

Saves the location of the current point in a register called *name*, where *name* is a single character.

`Save Position` is a synonym for **`Point to Register`**.

Jump to Register *Editor Command*

Arguments: *name*
Key sequence: **`Ctrl+X J`** *name*

Moves the current point to a location previously saved in the register called *name*.

`Jump to Saved Position` and **`Register to Point`** are both synonyms for **`Jump to Register`**.

Kill Register *Editor Command*

Arguments: *name*
Key sequence: None

Kills the register called *name*.

List Registers *Editor Command*

Arguments: None
Key sequence: None

Lists all existing registers.

Copy to Register *Editor Command*

Arguments: *name*
Key sequence: `Ctrl+X X` *name*

Saves the region between the mark and the current point to the register called *name*. The register is created if it does not exist.

When a prefix argument is supplied, the region is also deleted from the buffer.

`Put Register` is a synonym for **`Copy to Register`**.

Append to Register *Editor Command*

Arguments: *name*
Key sequence: None

Appends the region between the mark and the current point to the value in the register called *name*, which must already exist and contain a region.

When a prefix argument is supplied, the region is also deleted from the buffer.

Prepend to Register *Editor Command*

Arguments: *name*
Key sequence: None

Prepends the region between the mark and the current point to the value in the register called *name*, which must already exist and contain a region.

When a prefix argument is supplied, the region is also deleted from the buffer.

Insert Register *Editor Command*

Arguments: *name*
Key sequence: `Ctrl+X G` *name*

Copies the region from the register called *name* to the current point.

`Get Register` is a synonym for `Insert Register`.

3.26 Modes

A buffer can be in two kinds of mode at once: *major* and *minor*. The following two sections give a description of each, along with details of some commands which alter the modes.

In most cases, the current buffer can be put in a certain mode using the mode name as an Editor Command.

3.26.1 Major modes

The major modes govern how certain commands behave and how text is displayed. Major modes adapt a few editor commands so that their use is more appropriate to the text being edited. Some movement commands are affected by the major mode, as word, sentence, and paragraph delimiters vary with the mode. Indentation commands are very much affected by the major mode (see "Indentation" on page 81).

Major modes available in the LispWorks editor are as follows:

- *Fundamental mode*. Commands behave in their most general manner, default values being used throughout where appropriate.
- *Text mode*. Used for editing straight text and is automatically loaded if the file name ends in `.txt`, `.text` or `.tx`.
- *Lisp mode*. Used for editing Lisp programs and is automatically loaded if the file name ends in `.lisp`, `.lsp`, `.lispworks`, `.slisp`, `.l`, `.mcl` or `.cl`.
- *Directory mode*. Used for listing and operating on files in a directory, after invoking the `List Directory` command.
- *Shell mode*. Used for running interactive shells.
- *Manual Entry mode*. Used for display of Unix manual pages (from `man` command).

The major mode of most buffers may be altered explicitly by using the commands described below.

By default, Lisp mode is the major mode whenever you edit a file with type **`lisp`** (as with several other file types). If you have Lisp source code in files with another file type **`foo`**, put a form like this in your **`.lispworks`** file, adding your file extension to the default set:

```
(editor:define-file-type-hook
  ("lispworks" "lisp" "slisp" "l" "lsp" "mcl" "cl" "foo")
  (buffer type)
  (declare (ignore type))
  (setf (editor:buffer-major-mode buffer) "Lisp"))
```

Fundamental Mode *Editor Command*

Arguments: None
Key sequence: None

Puts the current buffer into Fundamental mode.

Text Mode *Editor Command*

Arguments: None
Key sequence: None

Puts the current buffer into Text mode.

Lisp Mode *Editor Command*

Arguments: None
Key sequence: None

Puts the current buffer into Lisp mode. Notice how syntax coloring is used for Lisp symbols. Also the balanced parentheses delimiting a Lisp form at or immediately preceding the cursor are highlighted, by default in green.

3.26.2 Minor modes

The minor modes determine whether or not certain actions take place. Buffers may be in any number of minor modes. No command details are given here as they are covered in other sections of the manuals.

Minor modes available in the LispWorks editor are as follows:

- *Overwrite mode*. Each character that is typed overwrites an existing character in the text—see “Overwriting” on page 80.
- *Auto Fill mode*. Lines are broken between words at the right hand margin automatically, so there is no need to type **`Return`** at the end of each line—see “Filling” on page 84.
- *Abbrev mode*. Allows abbreviation definitions to be expanded automatically—see “Abbreviations” on page 122.
- *Execute mode*. Used by the Listener and Shell tools to make history commands available (see the *LispWorks IDE User Guide*).

3.26.3 Default modes

default-modes *Editor Variable*

Default value: ("Fundamental")

This editor variable contains the default list of modes for new buffers.

3.26.4 Defining modes

New modes can be defined using the **`defmode`** function.

editor:defmode *Function*

`defmode` ***name*** **`&key`** ***setup-function syntax-table key-bindings no-redefine vars cleanup-function major-p transparent-p precedence*** **`=> nil`**

Defines a new editor mode called *name*.

name is a string containing the name of the mode being defined. *setup-function* is a function which sets up a buffer in this mode. *key-bindings* is a quoted list of key-binding directions. *no-redefine* is a boolean: if true, the mode cannot be re-defined. The default value of *no-redefine* is **`nil`**. *vars* is a quoted list of editor variables and values. *aliases* is a quoted list of synonyms for *name*. *cleanup-function* is a function which is called upon exit from a buffer in this mode. *major-p* is a boolean: if true, the mode is defined as major, otherwise minor. The default value of *major-p* is **`nil`**.

By default, any mode defined is a minor one—specification of major-mode status is made by supplying a true value for *major-p*.

defmode is essentially for the purposes of mode specification—not all of the essential definitions required to establish a new Editor mode are made in a **defmode** call. In the example, below, other required calls are shown.

key-bindings can be defined by supplying a quoted list of bindings, where a binding is a list containing as a first element the (string) name of the Editor command being bound, and as the second, the key binding description (see Chapter 6, "Advanced Features", for example key-bindings).

The state of Editor variables can be changed in the definition of a mode. These are supplied as a quoted list *vars* of dotted pairs, where the first element of the pair is the (symbol) name of the editor variable to be changed, and the second is the new value.

Both *setup-function* and *cleanup-function* are called with the mode and the buffer locked. They can modify the buffer itself, but they must not wait for anything that happens on another process, and they must not modify the mode (for example by setting a variable in the mode), and must not try to update the display.

As an example tet us define a minor mode, **Foo**. **Foo** has a set-up function, called **setup-foo-mode**. All files with suffix **.foo** invoke **Foo**-mode.

Here is the **defmode** form:

```
(editor:defmode "Foo" :setup-function 'setup-foo-mode)
```

The next piece of code makes **.foo** files invoke **Foo**-mode:

```
(editor:define-file-type-hook ("foo") (buffer type)
   (declare (ignore type))
   (setf (editor:buffer-minor-mode buffer "Foo") t))
```

The next form defines the set-up function:

```
(defun setup-foo-mode (buffer)
  (setf (editor:buffer-major-mode buffer) "Lisp")
  (let ((pathname (editor:buffer-pathname buffer)))
    (unless (and pathname
                 (probe-file pathname))
      (editor:insert-string
       (editor:buffer-point buffer)
       #.(format nil ";;; -*- mode :foo -*-~2%(in-package \"CL-
USER\")~2%")))))
```

Now, any files with the suffix `.foo` invoke the **`Foo`** minor mode when loaded into the Editor.

3.27 Abbreviations

Abbreviations (*abbrevs*) can be defined by the user, such that if an abbreviation is typed at the keyboard followed by a word terminating character (such as `Space` or `,`), the expansion is found and used to replace the abbreviation. Typing can thereby be saved for frequently used words or sequences of characters.

There are two kinds of abbreviations: *global abbreviations*, which are expanded in all major modes; and *mode abbreviations*, which are expanded only in defined major modes.

Abbreviations (both global and mode) are only expanded automatically when *Abbrev mode* (a minor mode) is on. The default is for abbrev mode to be off.

All abbreviations that are defined can be saved in a file and reloaded during later editor sessions.

Abbrev Mode *Editor Command*

Arguments: None
Key sequence: None

Switches abbrev mode on if it is currently off, and off if it is currently on. Only when in abbrev mode are abbreviations automatically expanded.

Add Mode Word Abbrev *Editor Command*

Arguments: *abbrev*
Key sequence: `Ctrl+X Ctrl+A` *abbrev*

Defines a mode abbreviation for the word before the current point.

A positive prefix argument defines an abbreviation for the appropriate number of words before the current point. A zero prefix argument defines an abbreviation for all the text in the region between the mark and the current point. A negative prefix argument deletes an abbreviation.

Inverse Add Mode Word Abbrev *Editor Command*

Arguments: *expansion*
Key sequence: `Ctrl+X Ctrl+H` *expansion*

Defines the word before the current point as a mode abbreviation for *expansion*.

Add Global Word Abbrev *Editor Command*

Arguments: *abbrev*
Key sequence: `Ctrl+X +` *abbrev*

Defines a global abbreviation for the word before the current point.

A positive prefix argument defines an abbreviation for the appropriate number of words before the current point. A zero prefix argument defines an abbreviation for all the text in the region between the mark and the current point. A negative prefix argument deletes an abbreviation.

Inverse Add Global Word Abbrev *Editor Command*

Arguments: *expansion*
Key sequence: `Ctrl+X -` *expansion*

Defines the word before the current point as a global abbreviation for *expansion*.

Make Word Abbrev *Editor Command*

Arguments: *abbrev expansion mode*
Key sequence: None

`editor:make-word-abbrev-command` ***p*** `&optional` ***abbrev expansion mode***

Defines an abbreviation for *expansion* without reference to the current point. The default value for *mode* is global.

Abbrev Expand Only *Editor Command*

Arguments: None
Key sequence: None

Expands the word before the current point into its abbreviation definition (if it has one). If the buffer is currently in abbrev mode then this is done automatically on meeting a word defining an abbreviation.

Word Abbrev Prefix Point *Editor Command*

Arguments: None
Key sequence: **`Meta+'`**

Allows the prefix before the current point to be attached to the following abbreviation. For example, if the abbreviation **`valn`** is bound to **`valua-tion`**, typing **`re`** followed by **`Meta+'`**, followed by **`valn`**, results in the expansion **`revaluation`**.

Unexpand Last Word *Editor Command*

Arguments: None
Key sequence: None

Undoes the last abbreviation expansion. If this command is typed twice in succession, the previous abbreviation is restored.

Delete Mode Word Abbrev *Editor Command*

Arguments: *abbrev*
Key sequence: None

`editor:delete-mode-word-abbrev-command` ***p*** `&optional` ***abbrev mode***

Deletes a mode abbreviation for the current mode. A prefix argument causes all abbreviations defined in the current mode to be deleted.

The argument *mode* is the name of the mode for which the deletion is to be applied. The default is the current mode.

Delete Global Word Abbrev *Editor Command*

Arguments: *abbrev*
Key sequence: None

`editor:delete-global-word-abbrev-command` ***p*** **`&optional`** ***abbrev***

Deletes a global abbreviation. A prefix argument causes all global abbreviations currently defined to be deleted.

Delete All Word Abbrevs *Editor Command*

Arguments: None
Key sequence: None

Deletes all currently defined abbreviations, both global and mode.

List Word Abbrevs *Editor Command*

Arguments: None
Key sequence: None

Displays a list of all the currently defined abbreviations in an Abbrev window.

Word Abbrev Apropos *Editor Command*

Arguments: *search-string*
Key sequence: None

`editor:word-abbrev-apropos-command` ***p*** **`&optional`** ***search-string***

Displays a list of all the currently defined abbreviations which contain *search-string* in their abbreviation definition or mode. The list is displayed in an Abbrev window.

Edit Word Abbrevs *Editor Command*

Arguments: None
Key sequence: None

Allows recursive editing of currently defined abbreviations. The abbreviation definitions are displayed in an Edit Word Abbrevs buffer, from where they can be can be added to, modified, or removed. This buffer can then either be saved to an abbreviations file, or **`Define Word Abbrevs`** can be used to define any added or modified abbreviations in the buffer. When editing is complete, **`Exit Recursive Edit`** should be invoked.

Write Word Abbrev File *Editor Command*

Arguments: *filename*
Key sequence: None

`editor:write-word-abbrev-file-command` ***p*** **`&optional`** ***filename***

Saves the currently defined abbreviations to *filename*. If no file name is provided, the default file name defined by the editor variable **`abbrev-pathname-defaults`** is used.

Append to Word Abbrev File *Editor Command*

Arguments: *filename*
Key sequence: None

`editor:append-to-word-abbrev-file-command` ***p*** **`&optional`** ***filename***

Appends all abbreviations that have been defined or redefined since the last save to *filename*. If no file name is provided, the default file name defined by the editor variable **`abbrev-pathname-defaults`** is used.

abbrev-pathname-defaults *Editor Variable*

Default value: **`abbrev.defns`**

Defines the default file name for saving the abbreviations that have been defined in the current buffer.

Read Word Abbrev File *Editor Command*

Arguments: *filename*
Key sequence: None

`editor:read-word-abbrev-file-command` ***p*** **`&optional`** ***filename***

Reads previously defined abbreviations from *filename*. The format of each abbreviation must be that used by **`Write Word Abbrev File`** and **`Insert Word Abbrevs`**.

Insert Word Abbrevs *Editor Command*

Arguments: None
Key sequence: None

Inserts into the current buffer, at the current point, a list of all currently defined abbreviations. This is similar to **`Write Word Abbrev File`**, except that the abbreviations are written into the current buffer rather than a file.

Define Word Abbrevs *Editor Command*

Arguments: None
Key sequence: None

Defines abbreviations from the definition list in the current buffer. The format of each abbreviation must be that used by **`Write Word Abbrev File`** and **`Insert Word Abbrevs`**.

3.28 Keyboard macros

Keyboard macros enable a sequence of commands to be turned into a single operation. For example, if it is found that a particular sequence of commands is to be repeated a large number of times, they can be turned into a keyboard macro, which may then be repeated the required number of times by using Prefix Arguments.

Note that keyboard macros are only available for use during the current editing session.

Define Keyboard Macro *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X Shift+(`**

Begins the definition of a new keyboard macro. All the commands that are subsequently invoked are executed and at the same time combined into the newly defined macro. Any text typed into the buffer is also included in the macro. The definition is ended with **`End Keyboard Macro`**, and the sequence of commands can then be repeated with **`Last Keyboard Macro`**.

End Keyboard Macro *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X Shift+)`**

Ends the definition of a keyboard macro.

Last Keyboard Macro *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X E`**

Executes the last keyboard macro defined. A prefix argument causes the macro to be executed the required number of times.

Name Keyboard Macro *Editor Command*

Arguments: *name*
Key sequence: None

`editor:name-keyboard-macro-command` ***p*** **`&optional`** ***name***

Makes the last defined keyboard macro into a command called *name* that can subsequently be invoked by means of **`Extended Command`**.

Keyboard Macro Query *Editor Command*

Arguments: *action*
Key sequence: **`Ctrl+X Q`** *action*

During the execution of a keyboard macro, this command prompts for an action. It is therefore possible to control the execution of keyboard macros while they are running, to a small extent.

The following actions can be used to control the current macro execution.

`Space`	Continue with this iteration of the keyboard macro and then proceed to the next.
`Delete`	Skip over the remainder of this iteration of the keyboard macro and proceed to the next.
`Escape`	Exit from this keyboard macro immediately.

3.29 Echo area operations

There are a range of editor commands which operate only on the Echo Area (that is, the buffer where the user types in commands).

Although in many cases the key bindings have a similar effect to the bindings used in ordinary buffers, this is just for the convenience of the user. In fact the commands that are invoked are different.

3.29.1 Completing commands

Many of the commands used in the Editor are long, in the knowledge that the user can use completion commands in the Echo Area, and so rarely has to type a whole command name. Details of these completion commands are given below.

Complete Input *Editor Command*

Arguments: None
Key sequence: **`Tab`**
Mode: Echo Area

Completes the text in the Echo Area as far as possible, thereby saving the user from having to type in the whole of a long file name or command. Use **`Tab Tab`** to produce a popup list of all possible completions.

Complete Field *Editor Command*

Arguments: None
Key sequence: `Space`
Mode: Echo Area

Completes the current part of the text in the Echo Area. So, for a command that involves two or more words, if `Complete Field` is used when part of the first word has been entered, an attempt is made to complete just that word.

Confirm Parse *Editor Command*

Arguments: None
Key sequence: `Return`
Mode: Echo Area

Terminates an entry in the Echo Area. The Editor then tries to parse the entry. If `Return` is typed in the Echo Area when nothing is being parsed, or the entry is erroneous, an error is signaled.

Help on Parse *Editor Command*

Arguments: None
Key sequence: `?` or `Help` or `F1`
Mode: Echo Area

Displays a popup list of all possible completions of the text in the echo area.

3.29.2 Repeating echo area commands

The Echo Area commands are recorded in a history ring so that they can be easily repeated. Details of these commands are given below.

Previous Parse *Editor Command*

Arguments: None
Key sequence: **`Meta+P`**
Mode: Echo Area

Moves to the previous command in the Echo Area history ring. If the current input is not empty and the contents are different from what is on the top of the ring, then this input is pushed onto the top of the ring before the new input is inserted.

Next Parse *Editor Command*

Arguments: None
Key sequence: **`Meta+N`**
Mode: Echo Area

Moves to the next most recent command in the Echo Area history ring. If the current input is not empty and the contents are different from what is on the top of the ring, then this input is pushed onto the top of the ring before the new input is inserted.

Find Matching Parse *Editor Command*

Arguments: *match-input-string*
Key sequence: **`Meta+R`**
Mode: Echo Area

The command **`Find Matching Parse`** searches for a previous input containing *match-input-string*, and replaces the current input with it.

3.29.3 Movement in the echo area

Echo Area Backward Character *Editor Command*

Arguments: None
Key sequence: **`Ctrl+B`**
Mode: Echo Area

Moves the cursor back one position (without moving into the prompt).

Echo Area Backward Word *Editor Command*

Arguments: None
Key sequence: **`Meta+B`**
Mode: Echo Area

Moves the cursor back one word (without moving into the prompt).

Beginning Of Parse *Editor Command*

Arguments: None
Key sequence: **`Meta+<`**
Mode: Echo Area

Moves the cursor to the location immediately after the prompt in the Echo Area.

Beginning Of Parse or Line *Editor Command*

Arguments: None
Key sequence: **`Ctrl+A`**
Mode: Echo Area

Moves the cursor to the location at the start of the current line in multi-line input, or to the location immediately after the prompt in the Echo Area.

3.29.4 Deleting and inserting text in the echo area

Echo Area Delete Previous Character *Editor Command*

Arguments: None
Key sequence: **`Backspace`**
Mode: Echo Area

Deletes the previous character entered in the Echo Area.

Echo Area Kill Previous Word *Editor Command*

Arguments: None
Key sequence: **`Meta+Backspace`**
Mode: Echo Area

Kills the previous word entered in the Echo Area.

Kill Parse *Editor Command*

Arguments: None
Key sequence: **`Ctrl+C Ctrl+U`**
Mode: Echo Area

Kills the whole of the input so far entered in the Echo Area.

Insert Parse Default *Editor Command*

Arguments: None
Key sequence: **`Ctrl+C Ctrl+P`**
Mode: Echo Area

Inserts the default value for the parse in progress at the location of the cursor. It is thereby possible to edit the default. Simply typing **`Return`** selects the default without any editing.

Return Default *Editor Command*

Arguments: None
Key sequence: **`Ctrl+C Ctrl+R`**
Mode: Echo Area

Uses the default value for the parse in progress. This is the same as issuing the command **`Insert Parse Default`** and then pressing **`Return`** immediately.

Insert Selected Text *Editor Command*

Arguments: None
Key sequence: `Ctrl+C Ctrl+C`
Mode: Echo Area

Inserts the editor window’s selected text in the echo area.

3.29.5 Display of information in the echo area

What Cursor Position *Editor Command*

Arguments: None
Key sequence: `Ctrl+X =`
Mode: Echo Area

Displays in the echo area the character under the point and the column of the point.

See also: **`Toggle Showing Cursor Info`**

Where Is Point *Editor Command*

Arguments: None
Key sequence: None

Displays in the echo area the position of the current point in terms of characters in the buffer, as a fraction of current point position over total buffer length.

Toggle Showing Cursor Info *Editor Command*

Arguments: None
Key sequence: None

The command **`Toggle Showing Cursor Info`** switches on or off display of cursor info in the echo area.

When display of cursor info is on, the info is updated whenever the cursor moves.

The info contains the character at the cursor position, its Unicode code point, position in the buffer, and column. It is the same information that is given by **`What Cursor Position`**.

3.29.6 Leaving the echo area

Reset Echo Area *Editor Command*

Arguments: None
Key sequence: **`Meta+K`**
Mode: Echo Area

The command **`Reset Echo Area`** resets the echo area, which means aborting any prompting ("recursive edit") and moving the focus to the main editor pane.

3.30 Editor variables

Editor variables are parameters which affect the way that certain commands operate. Descriptions of editor variables are provided alongside the relevant command details in this manual.

Show Variable *Editor Command*

Arguments: *variable*
Key sequence: None

Indicates the value of *variable*.

Set Variable *Editor Command*

Arguments: *variable value*
Key sequence: None

Allows the user to change the value of *variable*.

3.31 Recursive editing

Recursive editing occurs when you are allowed to edit text while an editor command is executing. The mode line of the recursively edited buffer is enclosed in square brackets. For example, when using the command `Query Replace`, the `Ctrl+R` option can be used to edit the current instance of the target string (that is, enter a recursive edit). Details of commands used to exit a recursive edit are given below.

Exit Recursive Edit *Editor Command*

Argument: None
Key sequence: `Meta+Ctrl+Z`

Exits a level of recursive edit, returning to the original command. An error is signaled if not in a recursive edit.

Abort Recursive Edit *Editor Command*

Argument: None
Key sequence: `Ctrl+]`

Aborts a level of recursive edit, quitting the unfinished command immediately. An error is signaled if not in a recursive edit.

3.32 Key bindings

The commands for modifying key bindings that are described below are designed to be invoked explicitly during each session with the Editor. If the user wishes to create key bindings which are set up every session, the function `editor:bind-key` should be used—see "Customizing default key bindings" on page 214.

Bind Key *Editor Command*

Argument: *command key-sequence bind-type*
Key sequence: None

Binds *command* (full command names must be used) to *key-sequence.*

After entering *command*, enter the keys of *key-sequence* and press `Return`.

bind-type can be either buffer, global or mode. If a *bind-type* of buffer or mode is selected, the name of the buffer or mode required must then be entered. When a *bind-type* of buffer is selected, the current buffer is offered as a default. The default value for *bind-type* is "Global".

Unless a bind type of global is selected, the scope of the new key binding is restricted as specified. Generally, most key bindings are global. Note that the Echo Area is defined as a mode, and some commands (especially those involving completion) are restricted to the Echo Area.

Bind String to Key *Editor Command*

Argument: *string key-sequence bind-type*
Key sequence: None

Make *key-sequence* insert *string*.

After entering *string*, enter the keys of *key-sequence* and press `Return`.

bind-type is interpreted as in `Bind Key`.

Delete Key Binding *Editor Command*

Argument: *key-sequence bind-type*
Key sequence: None

Removes a key binding, so that the key sequence no longer invokes any command. The argument *bind-type* can be either buffer, global or mode. If a *bind-type* of buffer or mode is selected, the name of the buffer or mode required must then be entered. The default value for *bind-type* is "Global".

It is necessary to enter the kind of binding, because a single key sequence may sometimes be bound differently in different buffers and modes.

Illegal *Editor Command*

Argument: None
Key sequence: None

Signals an editor error with the message "Illegal command in the current mode" accompanied by a beep. It is sometimes useful to bind key sequences to this command, to ensure the key sequence is not otherwise bound.

Do Nothing *Editor Command*

Argument: None
Key sequence: None

Does nothing. This is therefore similar to **`Illegal`**, except that there is no beep and no error message.

3.33 Execute mode

3.33.1 Listener commands

Use these commands in the Listener tool.

Beginning of Line After Prompt *Editor Command*

Arguments: None
Key sequence: **`Ctrl+A`**
Mode: Execute

The command **`Beginning of Line After Prompt`** moves the current point to the beginning of the current line, unless there is a prompt, in which case the point is moved to the end of the prompt.

With a prefix argument *p*, the point is moved to the beginning of the line *p* lines below the current line.

Insert From Previous Prompt *Editor Command*

Arguments: None
Key sequence: **`Ctrl+J`**
Mode: Execute

The command **`Insert From Previous Prompt`** picks up the form starting from the previous prompt and yanks it to the end of the buffer.

Inspect Star *Editor Command*

Arguments: None
Key sequence: **`Ctrl+C Ctrl+I`**
Mode: Execute

The command **`Inspect Star`** inspects the object that is the value of the symbol **`cl:*`**, which is normally the result of the previous command. Inspecting means activating the Inspector tool with the object.

See the *LispWorks IDE User Guide* for information about the Inspector tool.

Execute or Insert Newline or Yank from Previous Prompt *Editor Command*

Arguments: None
Key sequence: **`Return`**
Mode: Execute

The command **`Execute or Insert Newline or Yank from Previous Prompt`** does one of the actions indicated by its name, depending on the position of the point relative to the prompt.

If the current point is after or in the middle of the last prompt, insert a newline at the end of the buffer, and if there is an acceptable form after the last prompt, execute it.

If the point is before the last prompt, insert the command before the point at the end of the buffer, and move the point to the end of the buffer.

Throw To Top Level *Editor Command*

Arguments: None
Key sequence: **`Meta+K`**
Mode: Execute

The command **`Throw To Top Level`** exits the reading of commands, prints a prompt and starts reading again.

Note: this command is useful after you mistakenly pasted a large amount of text into the listener, and you cannot really see where the prompt is.

3.33.2 History commands

Use these commands in the Listener and Shell tools.

History First *Editor Command*

Arguments: None
Key sequence: **`Ctrl+C <`**
Mode: Execute

The command **`History First`** replaces the current command by the first recorded command in the history of commands in the current page.

Note: the length of the history is limited to 100, so earlier commands are not available.

History Last *Editor Command*

Arguments: None
Key sequence: **`Ctrl+C >`**
Mode: Execute

The command **`History Last`** replaces the current command by the last recorded command in the history of commands in the current page.

History Next *Editor Command*

Arguments: None
Key sequence: **`Meta+N`** or **`Ctrl+C Ctrl+N`**
Mode: Execute

The command **`History Next`** replaces the current command by the next one from the history of commands in the current page.

History Previous *Editor Command*

Arguments: None
Key sequence: `Meta+P` or `Ctrl+C Ctrl+P`
Mode: Execute

The command `History Previous` replaces the current command by the previous one from the history of commands in the current page.

If immediately follows `History Search From Input`, it does the search again.

History Search *Editor Command*

Argument: *search-string*
Key sequence: `Meta+R` or `Ctrl+C Ctrl+R` *search-string*
Mode: Execute

The command `History Search` searches for a previous command containing a supplied string, and replaces the current command with it.

History Kill Current *Editor Command*

Arguments: None
Key sequence: `Ctrl+C Ctrl+K`
Mode: Execute

The command `History Kill Current` deletes the current command, that is the text after the last prompt.

Note: this command is badly named. It has nothing to do with history.

History Search From Input *Editor Command*

Argument: *search-string*
Key sequence: None

The command `History Search From Input` searches for a previous command containing the string entered so far, and replaces the current command with it.

Repeated uses step back to previous matches.

If no string has been entered, the command prompts for a string to match like **`History Search`**.

History Select *Editor Command*

Arguments: None
Key sequence: **`Ctrl+C Ctrl+F`**
Mode: Execute

The command **`History Select`** opens a menu of the previous commands, and replaces the current command with the selection.

History Yank *Editor Command*

Arguments: None
Key sequence: **`Ctrl+C Ctrl+Y`**
Mode: Execute

The command **`History Yank`** inserts the previous command into the current one.

3.33.3 Debugger commands

These commands are applicable only in a **`capi:listener-pane`** (including listener panes in the Debugger and Inspector tools and so on), when in the debugger. Each has a corresponding short debugger command that you can enter at the debugger prompt. These are listed in the description.

The debugger prompt by default looks like this:

```
CL-USER 3 : 1 >
```

The first integer is the number of commands entered in the listener. The second integer is the number of levels deep in the debugger (that is, if it is **`2`** or more, you have entered the debugger recursively).

Debugger Abort *Editor Command*

Arguments: None
Key sequence: **`Meta+A`**
Mode: Execute
Debugger command: **`:a`**

The command **`Debugger Abort`** aborts, meaning invoking the restart that is recognized as the **`cl:abort`** restart.

Debugger Continue *Editor Command*

Arguments: None
Key sequence: **`Meta+C`**
Mode: Execute
Debugger command: **`:c`**

The command **`Debugger Continue`** continues, meaning invoking the restart that is recognized as the **`cl:continue`** restart.

Debugger Backtrace *Editor Command*

Arguments: None
Key sequence: **`Meta+B`**
Mode: Execute
Debugger command: **`:bq`** or **`:bb`** (approximately)

The command **`Debugger Backtrace`** displays a quick backtrace when in the debugger in a listener window.

A prefix argument makes the backtrace more verbose.

Debugger Edit *Editor Command*

Arguments: None
Key sequence: **`Meta+E`**
Mode: Execute
Debugger command: **`:ed`**

The command **`Debugger Edit`** tries to find the source of the current frame, and if successful displays that source in an Editor tool.

Debugger Next *Editor Command*

Arguments: None
Key sequence: **`Meta+N`**
Mode: Execute
Debugger command: **`:n`**

The command **`Debugger Next`** makes the next frame current.

Enter **`:v`** (**`Debugger Print`**) to see the value in the frame.

Debugger Previous *Editor Command*

Arguments: None
Key sequence: **`Meta+P`**
Mode: Execute
Debugger command: **`:p`**

The command **`Debugger Previous`** makes the previous frame current.

Enter **`:v`** (**`Debugger Print`**) to see the value in the frame.

Debugger Print *Editor Command*

Arguments: None
Key sequence: **`Meta+V`**
Mode: Execute
Debugger command: **`:v`**

The command **`Debugger Print`** displays the current frame.

Debugger Top *Editor Command*

Arguments: None
Key sequence: None
Debugger command: **`:top`**

The command **`Debugger Top`** aborts to the top level.

Throw out of Debugger *Editor Command*

Arguments: None
Key sequence: None

The command **`Throw out of Debugger`** is deprecated, use **`Debugger Top`** and **`Debugger Abort`** instead.

3.34 Running shell commands

The editor allows both single shell commands to be executed and also provides a means of running a shell interactively.

3.34.1 Running shell commands directly from the editor

Shell Command *Editor Command*

Argument: *command*
Key sequence: **`Meta+!`** *command*

The command **`Shell Command`** runs the console (shell) command *command*. The output from the command is displayed in a **Shell Output** buffer.

A prefix argument causes the output from the shell command to be sent to the **`*terminal-io*`** stream rather than the **Shell Output** buffer.

Shell Command On Region *Editor Command*

Arguments: *command*
Key sequence: **`Meta+|`** *command*

The command **`Shell Command On Region`** runs the console (shell) command *command* with the text in the current region as input (by redirection of the standard input), and shows the output.

Without a prefix argument, the output is inserted into the **Shell Output** buffer (which is created if it does not exist). With a prefix argument, the contents of the region are replaced by the output.

Run Command *Editor Command*

Argument: *command*
Key sequence: None

Executes the single shell command *command* in a Shell window. When the command terminates, the subprocess is closed down.

3.34.2 Invoking and using a Shell tool

See also the history commands in "Execute mode" on page 138.

Shell *Editor Command*

Argument: None
Key sequence: None

Opens a Shell window which allows the user to run a shell interactively.

The major mode of the buffer is Shell mode - the variables and key bindings described in this section apply. The minor mode of the buffer is Execute mode so the history key bindings (see "Execute mode" on page 138) can also be used in the Shell window.

Whenever the working directory is changed within the shell, the editor attempts to keep track of these changes and update the default directory of the Shell buffer. When a shell command is issued beginning with a string matching one of the editor variables **`shell-cd-regexp`**, **`shell-pushd-regexp`** or **`shell-popd-regexp`**, the editor recognizes this command as a change directory command and attempt to change the default directory of the Shell buffer accordingly. If you have your own aliases for any of the shell change directory commands, alter the value of the appropriate variable. For example, if the value of **`shell-cd-regexp`** is **`"cd`**" and the shell command **`cd /Applications/LispWorks`** is issued, the next time the editor command **`Wfind File`** is issued, the default directory offered is **`/Applications/LispWorks`**. If you find that the editor has not recognized a change directory command then the editor command **`cd`** may be used to change the default directory of the buffer.

Remote Shell *Editor Command*

Arguments: *machine-name*
Key sequence: None

The command `Remote Shell` prompts for a machine name and then starts a shell which tries to login to that computer using `rsh`.

Note: `Remote Shell` does not work on Microsoft Windows.

CD *Editor Command*

Arguments: *directory*
Key sequence: None
Mode: Shell

Changes the directory associated with the current buffer to *directory*. The current directory is offered as a default.

shell-cd-regexp *Editor Variable*

Default value: `"cd"`
Mode: Shell

A regular expression that matches the shell command to change the current working directory.

shell-pushd-regexp *Editor Variable*

Default value: `"pushd"`
Mode: Shell

A regular expression that matches the shell command to push the current working directory onto the directory stack.

shell-popd-regexp *Editor Variable*

Default value: `"popd"`
Mode: Shell

A regular expression that matches the shell command to pop the current working directory from the directory stack.

prompt-regexp-string *Editor Variable*

Default value: `"^[^#$%>`
`]*[#$%>] *"`
Mode: Shell

The regexp used to find the prompt in a Shell window. This variable is also used in the Listener.

Interrupt Shell Subjob *Editor Command*

Argument: None
Key sequence: `Ctrl+C Ctrl+C`
Mode: Shell

Sends an interrupt signal to the subjob currently being run by the shell. This is equivalent to issuing the shell command `Ctrl+C`.

Note: this command does not work on Microsoft Windows.

Stop Shell Subjob *Editor Command*

Argument: None
Key sequence: `Ctrl+C Ctrl+Z`
Mode: Shell

Sends a stop signal to the subjob currently being run by the shell. This is equivalent to issuing the shell command `Ctrl+Z`.

Note: this command does not work on Microsoft Windows.

Shell Send Eof *Editor Command*

Argument: None
Key sequence: `Ctrl+C Ctrl+D`
Mode: Shell

Sends an end-of-file character (`Ctrl+D`) to the shell, causing either the shell or its current subjob to finish.

Note: this command does not work on Microsoft Windows.

Kill Shell Subjob *Editor Command*

Arguments: None
Key sequence: None

The command `Kill Shell Subjob` tries to kill the subjob in the shell.

At the time of writing, on Solaris it actually sends a `SIGKILL` signal. On other Unix platforms it sends the `VQUIT` characters. On Microsoft Windows it calls `TerminateProcess`.

Terminate Shell Subjob *Editor Command*

Arguments: None
Key sequence: None

The command `Terminate Shell Subjob` tries to kill the subjob in the shell.

At the time of writing, on Solaris it actually sends a `SIGTERM` signal. On other Unix platforms it sends the `VQUIT` characters. On Microsoft Windows it calls `TerminateProcess`.

3.35 Buffers, windows and the mouse

3.35.1 Buffers and windows

You can transfer text between LispWorks Editor buffers and ordinary windows using the commands described below.

Copy to Cut Buffer *Editor Command*

Argument: None
Key sequence: None

Copies the current region to the Cut buffer. The contents of the buffer may then be pasted into a window using the standard method for pasting.

Insert Cut Buffer *Editor Command*

Argument: None
Key sequence: None

Inserts the contents of the Cut buffer at the current point. You can put text from a window into the Cut buffer using the standard method for cutting text (usually by holding the left mouse button while dragging the mouse).

3.35.2 Actions involving the mouse

The functions to which the mouse buttons are bound are not true Editor Commands. As such, the bindings cannot be changed. Details of mouse button actions are given below.

Note that marks may also be set by using editor key sequences—see "Marks and regions" on page 63—but also note that a region must be defined *either* by using the mouse *or* by using editor key sequences, as the region may become unset if a combination of the two is used. For example, using `Ctrl+Space` to set a mark and then using the mouse to go to the start of the required region unsets the mark.

left-button

Moves the current point to the position of the mouse pointer.

shift-left-button

In Emacs emulation, this moves the current point to the location of the mouse pointer and sets the mark to be the end of the new current form or comment line.

control-shift-left-button

Invokes the Editor Command `Save Region`, saving the region between the current point and the mark at the top of the kill ring. If the last command was `control-shift-left-button`, the Editor Command `Kill Region` is invoked instead. This allows one click to save the region, and two clicks to save and kill it.

middle-button

If your mouse has a middle button, it pastes the current selection at the location of the mouse pointer.

right-button

Brings up a context menu, from which a number of useful commands can be invoked. The options include **Cut**, **Copy**, and **Paste**.

shift-right-button

Inserts the form or comment line at the location of the mouse pointer at the current point.

3.36 Interaction with the GUI and the IDE

Activate Interface *Editor Command*

Arguments: *interface-title*
Key sequence: `Ctrl+;` *interface-title*

The command `Activate Interface` prompts for an interface title of an interface in the IDE, and activates it.

Note: this command works only in the LispWorks IDE.

Set Title *Editor Command*

Arguments: *title*
Key sequence: None

The command `Set Title` sets the title of the enclosing interface.

Note: switching buffers in the editor resets the title which will overwrite user changes, but other tool windows in the LispWorks IDE normally do not set their title.

Invoke Tool *Editor Command*

Argument: None
Key sequence: `Ctrl+#`

Invokes a tool in the LispWorks IDE.

Firstly `Invoke Tool` prompts for a character. If you enter a known shortcut character, the corresponding tool is activated. If the character is unknown, it raises the **Tools** menu so you can select from it.

Notes:

1. The shortcut characters can be seen in the **Tools** menu. So if you do not know the shortcut character, you can enter '?' to get the menu, and then note the shortcut character.
2. On Cocoa, in any window in the LispWorks IDE `Command+Ctrl+L` invokes the Listener tool (as an example), which is more convenient than `Invoke Tool`.
3. If the tool does not already exist, one is created if needed.
4. `Invoke Tool` does nothing in a delivered image.

Invoke Menu Item *Editor Command*

Arguments: *menu-item-path*
Key sequence: None

The command `Invoke Menu Item` invokes a menu item, as if the item was activated in any of the usual interactive ways.

The user is asked for a path, which is the title of the menu in the menu bar of the current interface, followed by the title(s) of submenus if any, followed by the item title itself.

The titles must be separated by a `/` (forward slash) and optionally `Space` or `Tab` characters, and other than this they must match (case-insensitive) the string that appears on the screen. For example, to do **File > Open...**, the *menu-item-path* is:

```
file / open...
```

Build Application *Editor Command*

Argument: None
Key sequence: None

The command `Build Application` invokes the Application Builder in the LispWorks IDE and does a build. By default, it uses the current buffer as the build script. If a prefix argument is supplied it prompts for a file to use as the build script.

See also: *LispWorks IDE User Guide,* Application Builder chapter.

Edit Compiler Warnings *Editor Command*

Arguments: None
Key sequence: None

The command `Edit Compiler Warnings` opens and activates the Compilation Conditions Browser, if there is a record of compilation conditions in the session.

Conditions may be generated whenever compiling code in the IDE.

See also: *LispWorks IDE User Guide,* Compilation Conditions Browser chapter.

Inspect Variable *Editor Command*

Arguments: *editor-variable-name*
Key sequence: None

The command `Inspect Variable` activates the Inspector tool with the object that is the value of the supplied editor variable.

List Buffer Definitions *Editor Command*

Arguments: None
Key sequence: None

The command `List Buffer Definitions` switches to the **Buffers** tab in an Editor tool.

Grep *Editor Command*

Arguments: *grep-args*
Key sequence: None

The command **`Grep`** activates the Search Files tool with a grep command.

It prompts for command line arguments, which should comprise the entire command line except for the first word **`grep`**. Then it activates the Search Files tool and invokes the grep command.

If the prefix argument is supplied, it saves all files after prompting and before activating the tool.

Note: the grep command to use is configurable via **`lw:*grep-command*`**. On Unix **`grep`** is available by default. On Microsoft Windows LispWorks uses **`lib/7-1-0-0/etc/grep.exe`** by default.

See also: **`Search Files`**, **`Search Files Matching Patterns`**, **`Search System`**.

Next Search Match *Editor Command*

Arguments: None
Key sequence: **`Ctrl+X ``**

The command **`Next Search Match`** displays the next match from the last search in the Search Files tool.

Next Grep *Editor Command*

Arguments: None
Key sequence: None

The command **`Next Grep`** is deprecated, use **`Next Search Match`** instead.

Show Directory *Editor Command*

Arguments: *path*
Key sequence: None

The command **`Show Directory`** opens the native file browser.

If no prefix argument is supplied and the current buffer is associated with a pathname, the browser is opened with this pathname. Otherwise, it prompts for a *path* to use.

Note: On Windows and Mac OS X, if it is a full filename, the file is selected. On other platforms it only opens the browser with the directory. On GTK+ it tries to use nautilus and if this is not on the path, it fails.

Report Bug *Editor Command*

Argument: None
Key sequence: None

The command **`Report Bug`** opens a window containing the template for reporting bugs in LispWorks. This template can then be filled in and emailed to Lisp Support.

Report Manual Bug *Editor Command*

Argument: None
Key sequence: None

The command **`Report Manual Bug`** opens a window containing the template for reporting bugs in the LispWorks documentation. This template can then be filled in and emailed to Lisp Support.

Bug Report *Editor Command*

Arguments: None
Key sequence: None

The command **`Bug Report`** is an alias for **`Report Bug`**.

Exit Lisp *Editor Command*

Arguments: None
Key sequence: None

The command **`Exit Lisp`** is an alias for **`Save All Files and Exit`**.

3.37 Miscellaneous

break-on-editor-error *Editor Variable*

Default value: `nil`

Specifies whether an `editor:editor-error` generates a Lisp `cerror`, or whether it just displays a message in the Echo Area.

Room *Editor Command*

Argument: None
Key sequence: None

Displays information on the current status of the memory allocation for the host computer.

3.38 Obscure commands

This section documents commands that we believe are unlikely to be useful. If you do find a use for any of these, please tell us at Lisp Support.

Clear Undo *Editor Command*

Arguments: None
Key sequence: None

The command `Clear Undo` clears undo information in the current buffer, after prompting the user for confirmation.

See also: `Undo`

List Faces Display *Editor Command*

Arguments: None
Key sequence: None

The command `List Faces Display` creates an editor buffer and displays in it all known editor faces.

Clear Eval Record *Editor Command*

Arguments: None
Key sequence: None

The command `Clear Eval Record` deletes the record of compilation and evaluation in the current buffer. This record is used by the Stepper to find the source code.

Redo *Editor Command*

Arguments: None
Key sequence: None

The command `Redo` redoes the last undone change. It operates only with simple Undo/Redo selected (see `Toggle Global Simple Undo`).

See also: `Toggle Global Simple Undo`

Toggle Global Simple Undo *Editor Command*

Arguments: None
Key sequence: None

The command `Toggle Global Simple Undo` toggles the type of undo between simple Undo/Redo and the Emacs-style of undo.

With a positive prefix argument simple Undo/Redo is selected, and with a zero or negative prefix argument Emacs-style undo is selected.

Note: the setting is global, that is it affects all editor buffers.

See also: `Undo`

Flush Sections *Editor Command*

Arguments: None
Key sequence: None

The command `Flush Sections` flushes information about the definitions in the current buffer gathered by sectioning, to force the editor to recompute it.

4

Editing Lisp Programs

There are a whole set of editor commands designed to facilitate editing of Lisp programs. These commands are designed to understand the syntax of the Lisp language and therefore allow movement over Lisp constructs, indentation of code, operations on parentheses and definition searching. Lisp code can also be evaluated and compiled directly from the editor.

To use some of these commands the current buffer should be in Lisp mode. For more information about editor modes, see “Modes” on page 118.

Commands are grouped according to functionality as follows:

- “Functions and definitions”
- “Forms”
- “Lists”
- “Comments”
- “Parentheses”
- “Documentation”
- “Evaluation and compilation”
- “Breakpoints”
- “Removing definitions”

4.1 Automatic entry into Lisp mode

Some source files begin with a line of this form

```
;;; -*- Mode: Common-Lisp; Author: m.mouse -*-
```

or this:

```
;; -*- Mode: Lisp; Author: m.mouse -*-
```

A buffer is automatically set to be in Lisp mode when such a file is displayed.

Alternatively, if you have files of Common Lisp code with extension other than **.lisp**, add the following code to your **.lispworks** file, substituting the extensions shown for your own. This ensures that Lisp mode is the major mode whenever a file with one of these extensions is viewed in the editor:

```
(editor:define-file-type-hook
     ("lispworks" "lisp" "slisp" "el" "lsp" "mcl" "cl")
     (buffer type)
     (declare (ignore type))
     (setf (editor:buffer-major-mode buffer) "Lisp"))
```

Another way to make a Lisp mode buffer is the command **New Buffer**, and you can put an existing buffer into Lisp mode via the command **Lisp Mode**.

4.2 Syntax coloring

When in Lisp mode, the LispWorks editor provides automatic Lisp syntax coloring and parenthesis matching to assist the editing of Lisp programs.

You can ensure a buffer is in Lisp mode as described in “Automatic entry into Lisp mode” .

To modify the colors used in Lisp mode syntax coloring, use **Preferences... > Environment > Styles > Colors And Attributes** as described in the *LispWorks IDE User Guide*. Adjust the settings for the styles whose names begin with "Lisp".

Commands controlling syntax coloring have names commencing **Font Lock**, for example **Font Lock Fontify Buffer**.

Font Lock Fontify Block *Editor Command*

Arguments: None
Key sequence: None

The command **`Font Lock Fontify Block`** fontifies some lines the way **`Font Lock Fontify Buffer`** would. The lines could be a Lisp definition, a paragraph, or a specified number of lines.

If a prefix argument is supplied, **`Font Lock Fontify Block`** fontifies that many lines before and after the current point. If no prefix argument is supplied and the editor variable **`font-lock-mark-block-function`** is **`nil`** it fontifies 16 lines before and after. If no prefix argument is supplied and **`font-lock-mark-block-function`** is non-nil, it is used to delimit the region to fontify.

Font Lock Fontify Buffer *Editor Command*

Arguments: None
Key sequence: None

The command **`Font Lock Fontify Buffer`** fontifies the current buffer.

Font Lock Mode *Editor Command*

Arguments: None
Key sequence: None

The command **`Font Lock Mode`** sets Font Lock mode.

Without a prefix argument it switches Font Lock mode on and off. With a prefix argument it sets Font Lock mode on when the argument is positive and off otherwise.

Global Font Lock Mode *Editor Command*

Arguments: *message*
Key sequence: None

The command **`Global Font Lock Mode`** switches Global Font Lock mode on and off.

With a prefix argument it turns Global Font Lock mode on if and only if the argument is positive.

If *message* is non-nil the command displays a message saying whether Font Lock mode is on or off.

It returns the new status of Global Font Lock mode (non-nil means on).

When Global Font Lock mode is enabled, Font Lock mode is automatically turned on for modes that support it, which currently is only Lisp mode.

font-lock-mark-block-function *Editor Variable*

Default value: `lisp-font-lock-mark-block-function`
Mode: Lisp

The editor variable `font-lock-mark-block-function` if non-nil is a function used by `Font Lock Fontify Block` to delimit the region to fontify.

The default value in Lisp mode delimits the current Lisp definition.

See also: `Font Lock Fontify Block`

4.3 Functions and definitions

4.3.1 Movement, marking and specifying indentation

Beginning of Defun *Editor Command*

Argument: None
Key sequence: `Meta+Ctrl+A`

Moves the current point to the beginning of the current top-level form. A positive prefix argument *p* causes the point to be moved to the beginning of the form *p* forms back in the buffer.

End of Defun *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+E`**

Moves the current point to the end of the current top-level form. A positive prefix argument *p* causes the point to be moved to the end of the form *p* forms forward in the buffer.

Mark Defun *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+H`**

Puts the mark at the end of the current top-level form and the current point at the beginning of the form. The definition thereby becomes the current region. If the current point is initially located between two top-level forms, then the mark and current point are placed around the previous top-level form.

Defindent *Editor Command*

Argument: *no-of-args*
Key sequence: None

Defines the number of arguments of the operator to be specially indented if they fall on a new line. The indent is defined for the operator name, for example **`defun`**.

`Defindent` affects the special argument indentation for all forms with that operator which you subsequently indent.

4.3.2 Definition searching

Definition searching involves taking a name (of a macro, variable, editor command, and so on), and finding the actual definition. This is particularly useful in large systems, where code may exist in a large number of source files.

Definitions are found by using information provided either by LispWorks source recording or by a Tags file. If source records or Tags information have not been made available to LispWorks, then the following commands do not work.

To make the information available to LispWorks, set the variable **`dspec:*active-finders*`** appropriately. See the *LispWorks User Guide and Reference Manual* for details.

Source records are created if the variable **`*record-source-files*`** is true when definitions are compiled, evaluated or loaded. See the *LispWorks User Guide and Reference Manual* for details.

Tag information is set up by the editor itself, and can be saved to a file for future use. For each file in a defined system, the tag file contains a relevant file name entry, followed by names and positions of each defining form in that file. Before tag searching can take place, there must exist a buffer containing the required tag information. You can specify a previously saved tag file as the current tag buffer, or you can create a new one using **`Create Tags Buffer`**. GNU Emacs tag files are fully compatible with LispWorks editor tag files.

After a command such as **`Meta+.`** (**`Find Source`**), if there are multiple definitions repeated use of **`Meta+,`** (**`Continue Tags Search`**) finds them in turn. If you then wish to revisit a particular definition, try the commands **`Go Back`** and **`Select Go Back`**.

Find Source *Editor Command*

Argument: *name*
Key sequence: **`Meta+.`** *name*

Tries to find the source code for *name*. The symbol under the current point is offered as a default value for *name.* A prefix argument automatically causes this default value to be used.

If the source code for *name* is found, the file in which it is contained is displayed in a buffer. When there is more than one definition for *name*, **`Find Source`** finds the first definition, and **`Meta+,`** (**`Continue Tags Search`**) finds subsequent definitions.

`Find Source` searches for definitions according to the value of **`dspec:*active-finders*`**. You can control which source record information is searched, and the order in which these are searched, by setting this variable appropriately. See the *LispWorks User Guide and Reference Manual* for details. There is an example setting for this variable in the configuration files supplied.

If `dspec:*active-finders*` contains the value `:tags`, `Find Source` prompts for the name of a tags file, and this is used for the current and subsequent searches.

The found source is displayed according to the value of `editor:*source-found-action*`. This depends on the buffer with the found definition being in Lisp mode. For information on how to ensure this for particular file types, see “Automatic entry into Lisp mode” on page 160.

Find Source for Dspec *Editor Command*

Argument: *dspec*
Key sequence: None

This command is similar to `Find Source`, but takes a definition spec *dspec* instead of a name as its argument.

For example, given a generic function `foo` of one argument, with methods specializing on classes `bar` and `baz`,

```
Find Source for Dspec foo
```

will find each method definition in turn (with the continuation via `Meta+,`) whereas

```
Find Source for Dspec (method foo (bar))
```

finds only the definition of the method on `bar`.

Find Command Definition *Editor Command*

Argument: *command*
Key sequence: None

This command is similar to `Find Source`, but takes the name of an editor command, and tries to find its source code.

Except in the Personal Edition, you can use this command to find the definitions of the predefined editor commands. See the *LispWorks User Guide and Reference Manual* chapter "Customization of LispWorks" for details.

See also: **`Find Key Definition`**

Edit Editor Command *Editor Command*

Argument: *command*
Key sequence: None

This is a synonym for **`Find Command Definition`**.

Find Key Definition *Editor Command*

Arguments: *keys*
Key sequence: **`Ctrl+H Ctrl+S`** *keys*

The command **`Find Key Definition`** prompts for a key sequence *keys*, and finds the source code definition of the editor command (if any) that is bound to it.

See also: **`Find Command Definition`**

Find Source For Current Package *Editor Command*

Argument: None
Key sequence: None

This command is similar to **`Find Source`**, but finds the **`defpackage`** definition for the package at the current point. If a prefix argument is given, it first prompts for a package name.

View Source Search *Editor Command*

Argument: *function*
Key sequence: None

Shows the results of the latest source search (initiated by **`Find Source`** or **`Find Source for Dspec`** or **`Find Command Definition`**) in the **Find Definitions** view of the Editor. See the chapter on the Editor tool in the *LispWorks IDE User Guide* for more information about the **Find Definitions** view.

List Definitions *Editor Command*

Argument: *name*
Key sequence: None

List the definitions for *name*. The symbol under the current point is offered as a default value for *name.* A prefix argument automatically causes this default value to be used.

This command searches for definitions and shows the results in the **Find Definitions** view of the Editor tool instead of finding the first definition. It does not set up the `Meta+,` action.

See the chapter on the Editor tool in the *LispWorks IDE User Guide* for more information about the **Find Definitions** view.

List Definitions For Dspec *Editor Command*

Argument: *dspec*
Key sequence: None

This command is similar to `List Definitions`, but takes a definition spec *dspec* instead of a name as its argument.

This command searches for definitions and shows the results in the **Find Definitions** view of the Editor tool instead of finding the first definition. This command does not set up the `Meta+,` action.

See the chapter on the Editor tool in the *LispWorks IDE User Guide* for more information about the **Find Definitions** view.

Create Tags Buffer *Editor Command*

Argument: None
Key sequence: None

Creates a buffer containing Tag search information, for all the `.lisp` files in the current directory. If you want to use this information at a later date then save this buffer to a file (preferably a file called `TAGS` in the current directory).

The format of the information contained in this buffer is compatible with that of GNU Emacs tags files.

A prefix argument causes the user to be prompted for the name of a file containing a list of files, to be used for constructing the tags table.

Find Tag *Editor Command*

Key sequence: `Meta+?`

Tries to find the source code for a name containing a partial or complete match a supplied string by examining the Tags information indicated by the value of `dspec:*active-finders*`.

The text under the current point is offered as a default value for the string.

If the source code for a match is found, the file in which it is contained is displayed. When there is more than one definition, `Find Tag` finds the first definition, and `Meta+,` (`Continue Tags Search`) finds subsequent definitions.

The found source is displayed according to the value of `editor:*source-found-action*`.

If there is no tags information indicated by the value of `dspec:*active-finders*`, `Find Tag` prompts for the name of a tags file. The default is a file called `TAGS` in the current directory. If there is no such file, you can create one using `Create Tags Buffer`. If you want to search a different directory, specify the name of a tags file in that directory.

See the chapter on the `DSPEC` package in the *LispWorks User Guide and Reference Manual* for information on how to use the `dspec:*active-finders*` variable to control how this command operates. There is an example setting for this variable in the configuration files supplied.

See also `Find Source`, `Find Source for Dspec` and `Create Tags Buffer`.

Tags Search *Editor Command*

Key sequence: None

Exhaustively searches each file mentioned in the Tags files indicated by the value of **`dspec:*active-finders*`** for a supplied string *string*. Note that this does not merely search for definitions, but for any occurrence of the string.

If *string* is found, it is displayed in a buffer containing the relevant file. When there is more than one definition, **`Tags Search`** finds the first definition, and **`Meta+,`** (**`Continue Tags Search`**) finds subsequent definitions.

If there is no Tags file on **`dspec:*active-finders*`**, **`Tags Search`** prompts for the name of a tags file. The default is a file called **`TAGS`** in the current directory. If there is no such file, you can create one using **`Create Tags Buffer`**. If you want to search a different directory, specify the name of a tags file in that directory.

Continue Tags Search *Editor Command*

Argument: None
Key sequence: **`Meta+,`**

Searches for the next match in the current search. This command is only applicable if issued immediately after a **`Find Source`**, **`Find Source for Dspec`**, **`Find Command Definition`**, **`Edit Callers`**, **`Edit Callees`**, **`Find Tag`** or **`Tags Search`** command.

Tags Query Replace *Editor Command*

Key sequence: None

Allows you to replace occurrences of a supplied string *target* by a second supplied string *replacement* in each Tags file indicated by the value of **`dspec:*active-finders*.`**

Each time *target* is found, an action must be specified from the keyboard. For details of the possible actions see **`Query Replace`**.

If there is no Tags file indicated by **`dspec:*active-finders*`**, **`Tags Query Replace`** prompts for the name of a tags file. The default is a file called **TAGS** in the current directory. If there is no such file, you can create one using **Create Tags Buffer**.

Visit Tags File *Editor Command*

Key sequence: None

Prompts for a Tags file *file* and makes the source finding commands use it. This is done by modifying, if necessary, the value of **dspec:*active-finders***.

If *file* is already in **dspec:*active-finders***, this command does nothing.

If there are other Tags files indicated then **Visit Tags File** prompts for whether to add simply add *file* as the last element of **dspec:*active-finders***, or to save the current value of **dspec:*active-finders*** and start a new list of active finders, setting **dspec:*active-finders*** to the new value (**:internal** *file*). In this case, the previous active finders list can be restored by the command **Rotate Active Finders**.

If the value **:tags** appears on the list **dspec:*active-finders*** then *file* replaces this value in the list.

If there is no tags information indicated then **Visit Tags File** simply adds *file* as the last element of **dspec:*active-finders***.

Rotate Active Finders *Editor Command*

Key sequence: **Meta+Ctrl+.**

Rotates the active finders history, activating the least recent one. This modifies the value of **dspec:*active-finders***.

The active finders history can have length greater than 1 if **Visit Tags File** started a new list of active finders, or if a buffer associated with a TAGS file on **dspec:*active-finders*** was killed.

Visit Other Tags File is a synonym for **Rotate Active Finders**.

4.3.3 Tracing functions

The commands described in this section use the Common Lisp `trace` facility. Note that you can switch tracing on and off using `dspec:tracing-enabled-p` - see the *LispWorks User Guide and Reference Manual* for details of this.

Trace Function *Editor Command*

Argument: *function*
Key sequence: None

This command traces *function*. The symbol under the current point is offered as a default value for *function.* A prefix argument automatically causes this default value to be used.

Trace Function Inside Definition *Editor Command*

Argument: *function*
Key sequence: None

This command is like `Trace Function`, except that *function* is only traced within the definition that contains the cursor.

Untrace Function *Editor Command*

Argument: *function*
Key sequence: None

This command untraces *function*. The symbol under the current point is offered as a default value for *function.* A prefix argument automatically causes this default value to be used.

Trace Definition *Editor Command*

Argument: None
Key sequence: None

This command traces the function defined by the current top-level form.

Trace Definition Inside Definition *Editor Command*

Argument: None
Key sequence: None

This command is like **`Trace Definition`**, except that with a non-nil prefix argument, prompts for a symbol to trace. Also, it prompts for a symbol naming a second function, and traces the first only inside this.

Untrace Definition *Editor Command*

Argument: None
Key sequence: None

This command untraces the function defined by the current top-level form.

Untrace All *Editor Command*

Arguments: None
Key sequence: None

The command **`Untrace All`** untraces all traced definitions.

Break Function *Editor Command*

Argument: *function*
Key sequence: None

This command is like **`Trace Function`** but the trace is with **`:break t`** so that when *function* is entered, the debugger is entered.

Break Function on Exit *Editor Command*

Argument: *function*
Key sequence: None

This command is like **`Trace Function`** but the trace is with **`:break-on-exit t`** so that when a called to *function* exits, the debugger is entered.

Break Definition *Editor Command*

Argument: None
Key sequence: None

Like `Trace Definition` but the definition is traced with `:break t`.

Break Definition on Exit *Editor Command*

Argument: None
Key sequence: None

Like `Trace Definition` but the definition is traced with `:break-on-exit t`.

4.3.4 Function callers and callees

The commands described in this section, require that LispWorks is producing cross-referencing information. This information is produced by turning source debugging on while compiling and loading the relevant definitions (see `toggle-source-debugging` in the *LispWorks User Guide and Reference Manual*).

List Callers *Editor Command*

Argument: *dspec*
Key sequence: None

Produces a Function Call Browser window showing those functions that call the definition named by *dspec*. The name of the current top-level definition is offered as a default value for *dspec*. A prefix argument automatically causes this default value to be used.

See "Dspecs: Tools for Handling Definitions" in the *LispWorks User Guide and Reference Manual* for a description of dspecs.

List Callees *Editor Command*

Argument: *dspec*
Key sequence: None

Produces a Function Call Browser window showing those functions that are called by the definition named by *dspec*. The name of the current top-level definition is offered as a default value for *dspec*. A prefix argument automatically causes this default value to be used.

See "Dspecs: Tools for Handling Definitions" in the *LispWorks User Guide and Reference Manual* for a description of dspecs.

Show Paths To *Editor Command*

Argument: *dspec*
Key sequence: None

Produces a Function Call Browser window showing the callers of the definition named by *dspec*. The name of the current top-level definition is offered as a default value for *dspec*. A prefix argument automatically causes this default value to be used.

See "Dspecs: Tools for Handling Definitions" in the *LispWorks User Guide and Reference Manual* for a description of dspecs.

Show Paths From *Editor Command*

Argument: *dspec*
Key sequence: None

Produces a Function Call Browser window showing the function calls from the definition named by *dspec*. The name of the current top-level definition is offered as a default value for *dspec*. A prefix argument automatically causes this default value to be used.

See "Dspecs: Tools for Handling Definitions" in the *LispWorks User Guide and Reference Manual* for a description of dspecs.

Edit Callers *Editor Command*

Argument: *function*
Key sequence: None

Produces an Editor window showing the latest definition found for a function that calls *function*. The name of the current top-level definition is offered as a default value for *function*. A prefix argument automatically causes this default value to be used. The latest definitions of each of the other functions that call *function* are available via the **`Continue Tags Search`** command.

Edit Callees *Editor Command*

Argument: *function*
Key sequence: None

Produces an Editor window showing the latest definition found for a function called by *function*. The name of the current top-level definition is offered as a default value for *function*. A prefix argument automatically causes this default value to be used. The latest definitions of each of the other functions that are called by *function* are available via the **`Continue Tags Search`** command.

4.3.5 Indentation and Completion

Indent Selection or Complete Symbol *Editor Command*

Argument: None
Key sequence: **`Tab`**
Mode: Lisp

Does Lisp indentation if there is a visible region. Otherwise, it attempts to indent the current line. If the current line is already indented correctly then it attempts to complete the symbol before the current point. See **`Complete Symbol`** for more details.

The prefix argument, if supplied, is interpreted as if by **`Indent Selection`** or **`Complete Symbol`**.

Indent or Complete Symbol *Editor Command*

Argument: None
Key sequence: None

Attempts to indent the current line. If the current line is already indented correctly then it attempts to complete the symbol before the current point. See **`Complete Symbol`** for more details.

The prefix argument, if supplied, is interpreted as if by **`Indent`** or **`Complete Symbol`**.

Complete Symbol *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+I`**

Attempts to complete the text before the current point to a symbol. If the string to be completed is not unique, a list of possible completions is displayed.

If the **Use in-place completion** preference is selected then the completions are displayed in a window which allows most keyboard gestures to be processed as ordinary editor input. This allows speedy reduction of the number of possible completions, while you can select the desired completion with **`Return`**, **`Up`** and **`Down`**.

If a prefix argument is supplied then only symbols which are bound or fbound are offered amongst the possible completions.

Abbreviated Complete Symbol *Editor Command*

Argument: None
Key sequence: **`Meta+I`**

Attempts to complete the symbol abbreviation before the current point. If the string to be completed is not unique, a list of possible completions is displayed.

A symbol abbreviation is a sequence of words (sequences of alphanumeric characters) separated by connectors (sequences of non-alphanumeric, non-whitespace characters). Each word (connector) is a prefix of the corresponding word (connector) in the expansions. Thus if you complete the symbol abbreviation `w-o` then `with-open-file` and `with-open-stream` are amongst the completions offered, assuming the COMMON-LISP package is visible.

If the **Use in-place completion** preference is selected then the completions are displayed in a window which allows most keyboard gestures to be processed as ordinary editor input. This allows speedy reduction of the number of possible completions, while you can select the desired completion with `Return`, `Up` and `Down`.

If a prefix argument is supplied then only symbols which are bound or fbound are offered amongst the possible completions.

4.3.6 Miscellaneous

Buffer Changed Definitions *Editor Command*

Argument: None
Key sequence: None

Calculates which definitions have been changed in the current buffer during the current LispWorks session, and displays these in the **Changed Definitions** tab of the Editor tool.

By default the reference point against which changes are calculated is the time when the file was last read into the buffer. A prefix argument equal to the value of the editor variable `prefix-argument-default` means the reference point is the last evaluation. A prefix argument of 1 means the reference point is the time the buffer was last saved to file.

Note: the most convenient way to use this command is via the Editor tool. Switch it to the **Changed Definitions** tab, where you can specify the reference point for calculating the changes.

Function Arglist *Editor Command*

Argument: *function*
Key sequence: `Meta+=` *function*

Prints the arguments expected by *function* in the Echo Area. The symbol under the current point is offered as a default value for *function*. A prefix argument automatically causes this default value to be used.

Example code showing how to use this command to display argument lists automatically is supplied with LispWorks:

```
(example-edit-file "editor/commands/space-show-arglist")
```

Function Argument List *Editor Command*

Argument: *function*
Key sequence: `Ctrl+Shift+A` *function*

The command `Function Argument List` is a more sophisticated version of `Function Arglist` which works on the current form rather than the current symbol.

The symbol at the head of the current form is offered as a default value for *function*, unless that symbol is a member of the list `editor:*find-likely-function-ignores*` in which case the second symbol in the form is offered as the default. A prefix argument automatically causes this default value to be used.

Function Arglist Displayer *Editor Command*

Argument: None
Key sequence: ``Ctrl+` ``

Shows or hides information about the operator in the current form. The command controls display of a special window (displayer) on top of the editor. The displayer shows the operator and its arguments, and tries to highlight the current argument (that is, the argument at the cursor position). If it does not recognize the operator of the current form, it tries the surrounding form, and if that fails it tries a third level of surrounding form.

While the displayer is visible:

`Ctrl++`	Moves the displayer up.
`Ctrl+-`	Moves the displayer down.

You can dismiss the displayer by invoking the command again, or by entering `Ctrl+G`. On Cocoa and Windows it is dismissed automatically when the underlying pane loses the focus.

In the LispWorks IDE you can change the style of the highlighting by **Preferences... > Environment > Styles > Colors and Attributes > Arglist Highlight**.

Additionally, while the displayer is visible:

`Ctrl+/`	Controls whether the documentation string of the operator is also shown.

Lastly, if passed a prefix argument, for example by typing `Ctrl+U Ctrl+`` ` then it displays the operator and its arguments, with highlight, in the Echo Area, rather than a displayer window. This Echo Area display is interface-specific, and implemented only for the Editor and other tools based on the editor.

Describe Class *Editor Command*

Argument: *class*
Key sequence: None

Displays a description of the class named by *class* in a Class Browser tool. The symbol under the current point is offered as a default value for *class.* A prefix argument automatically causes this default value to be used.

Describe Generic Function *Editor Command*

Argument: *function*
Key sequence: None

Displays a description of *function* in a Generic Function Browser tool. The symbol under the current point is offered as a default value for *function.* A prefix argument automatically causes this default value to be used.

Describe Method Call *Editor Command*

Argument: None
Key sequence: None

Displays a Generic Function Browser tool, with a specific method combination shown.

When invoked with a prefix argument *p* while the cursor is in a **`defmethod`** form, it uses the generic function and specializers of the method to choose the method combination.

Otherwise, it prompts for the generic function name and the list of specializers, which can be class names or lists of the form **`(eql`** ***object*****`)`** where *object* is not evaluated.

Describe System *Editor Command*

Argument: *system*
Key sequence: None

Displays a description of the **`defsystem`**-defined system named by *system*. The symbol under the current point is offered as a default value for *system*. A prefix argument automatically causes this default value to be used.

4.4 Forms

4.4.1 Movement, marking and indentation

Forward Form *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+F`**

Moves the current point to the end of the next form. A positive prefix argument causes the point to be moved the required number of forms forwards.

Backward Form *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+B`**

Moves the current point to the beginning of the previous form. A positive prefix argument causes the point to be moved the required number of forms backwards.

Mark Form *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+@`**

Puts the mark at the end of the current form. The current region is that area from the current point to the end of form. A positive prefix argument puts the mark at the end of the relevant form.

Indent Form *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+Q`**

If the current point is located at the beginning of a form, the whole form is indented in a manner that reflects the structure of the form. This command can therefore be used to format a whole definition so that the structure of the definition is apparent.

See **`editor:*indent-with-tabs*`** for control over the insertion of **`#\Tab`** characters by this and other indentation commands.

4.4.2 Killing forms

Forward Kill Form *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+K`**

Kills the text from the current point up to the end of the current form. A positive prefix argument causes the relevant number of forms to be killed forwards. A negative prefix argument causes the relevant number of forms to be killed backwards.

Backward Kill Form *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+Backspace`**

Kills the text from the current point up to the start of the current form. A positive prefix argument causes the relevant number of forms to be killed backwards. A negative prefix argument causes the relevant number of forms to be killed forwards.

Kill Backward Up List *Editor Command*

Argument: None
Key sequence: None

Kills the form surrounding the current form. The cursor must be on the opening bracket of the current form. The entire affected area is pushed onto the kill-ring. A prefix argument causes the relevant number of surrounding lists to be removed.

For example, given the following code, with the cursor on the second open-bracket:

```
(print (do-some-work 1 2 3))
```

`Kill Backward Up List` would kill the outer form leaving this:

```
(do-some-work 1 2 3)
```

Also available through the function **`editor:kill-backward-up-list-command`**.

`Extract List` is a synonym for **`Kill Backward Up List`**.

4.4.3 Macro-expansion of forms

Macroexpand Form *Editor Command*

Argument: None
Key sequence: **`Ctrl+Shift+M`**

Macro-expands the form after the current point. The output is sent to the Output window. A prefix argument causes the output to be displayed in the current buffer.

Walk Form *Editor Command*

Argument: None
Key sequence: **`Meta+Shift+M`**

Produces a macroexpansion of the form after the current point. The output is sent to the Output window. A prefix argument causes the output to be displayed in the current buffer.

Note: **`Walk Form`** does not expand the Common Lisp macros **`cond`**, **`prog`**, **`prog*`** and **`multiple-value-bind`**, though it does expand their subforms.

4.4.4 Miscellaneous

Transpose Forms *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+T`**

Transposes the forms immediately preceding and following the current point. A zero prefix argument causes the forms at the current point and the current mark to be transposed. A positive prefix argument causes the form at or preceding the current point to be transposed with the form the relevant number of forms forward. A negative prefix argument causes the form at or preceding the current point to be transposed with the form the relevant number of forms backward.

Insert Double Quotes For Selection *Editor Command*

Argument: None
Key sequence: **`Meta+"`**

Inserts a pair of double-quotes around the selected text, if any. If there is no selected text and a prefix argument *p* is supplied, insert them around the *p* following (or preceding) forms. Otherwise insert them at the current point. The point is left on the character after the first double-quote.

4.5 Lists

4.5.1 Movement

Forward List *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+N`**

Moves the current point to the end of the current list. A positive prefix argument causes the point to be moved the required number of lists forwards.

Backward List *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+P`**

Moves the current point to the beginning of the current list. A positive prefix argument causes the point to be moved the required number of lists backwards.

Forward Up List *Editor Command*

Argument: None
Key sequence: None

Moves the current point to the end of the current list by finding the first closing parenthesis that is not matched by an opening parenthesis after the current point.

Backward Up List *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+U`**

Moves the current point to the beginning of the current list by finding the first opening parenthesis that is not matched by a closing parenthesis before the current point.

Down List *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+D`**

Moves the current point to a location down one level in the current list structure. A positive prefix argument causes the current point to be moved down the required number of levels.

4.6 Comments

Comment Region *Editor Command*

Arguments: None
Key sequence: None

The command **`Comment Region`** comments a region according to the mode.

This command has an effect only if the **`comment-begin`** variable is set. By default, **`comment-begin`** is set in the Lisp, IDL and C modes.

The commented region is the current region, extended to the beginning of the line where the region starts and the end of the line where it ends.

The prefix argument determines the number of repetitions of the **`comment-begin`** string when the length of **`comment-begin`** is one, as in Lisp mode. When **`comment-begin`** is longer, the prefix argument is ignored. If the prefix argument is **`nil`**, a single character **`comment-begin`** is repeated three times.

Set Comment Column *Editor Command*

Argument: None
Key sequence: **`Ctrl+X ;`**

Sets the comment column to the current column. A positive prefix argument causes the comment column to be set to the value of the prefix argument.

The value is held in the editor variable **`comment-column`**.

Indent for Comment *Editor Command*

Argument: None
Key sequence: **`Meta+;`**

Creates a new comment or moves to the beginning of an existing comment, indenting it appropriately (see **`Set Comment Column`**).

If the current point is in a line already containing a comment, that comment is indented as appropriate, and the current point is moved to the beginning of the comment. An existing double semicolon comment is aligned as for a line of code. An existing triple semicolon comment or a comment starting in column 0, is not moved.

A prefix argument causes comments on the next relevant number of lines to be indented. The current point is moved down the relevant number of lines.

If characters not associated with the comment extend past the comment column, a space is added before starting the comment.

Insert Multi Line Comment For Selection *Editor Command*

Argument: None
Key sequence: **`Meta+#`**

Inserts multi line comment syntax around the selected text, if any. If there is no selected text and a prefix argument *p* is supplied, inserts them around *p* following (or preceding) forms. Otherwise it inserts them at the current point. The point is left on the first character inside the comment.

Up Comment Line *Editor Command*

Argument: None
Key sequence: **`Meta+P`**

Moves to the previous line and then performs an **`Indent for Comment`**.

Down Comment Line *Editor Command*

Argument: None
Key sequence: **`Meta+N`**

Moves to the next line and then performs an **`Indent for Comment`**.

Indent New Comment Line *Editor Command*

Argument: None
Key sequence: **`Meta+J`** or **`Meta+Newline`**

Ends the current comment and starts a new comment on the next line, using the indentation and number of comment start characters from the previous line's comment. If **`Indent New Comment Line`** is performed when the current point is not in a comment line, it simply acts as a **`Return`**.

Kill Comment *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+;`**

Kills the comment on the current line and moves the current point to the next line. If there is no comment on the current line, the point is simply moved onto the next line. A prefix argument causes the comments on the relevant number of lines to be killed and the current point to be moved appropriately.

The comment is identified by matching against the value of `comment-start`.

comment-begin *Editor Variable*

Default value: `";"`
Mode: Lisp

When the value is a string, it is inserted to begin a comment by commands like `Indent for Comment` and `Indent New Comment Line`.

comment-start *Editor Variable*

Default value: `";"`
Mode: Lisp

A string that begins a comment. When the value is a string, it is inserted to start a comment by commands like `Indent New Comment Line`, or used to identify a comment by commands like `Kill Comment`.

comment-column *Editor Variable*

Default value: 0
Mode: Lisp

Column to start comments in. Set by `Set Comment Column`.

comment-end *Editor Variable*

Default value: `nil`
Mode: Lisp

String that ends comments. The value `nil` indicates Newline termination. If the value is a string, it is inserted to end a comment by commands like `Indent New Comment Line`.

4.7 Parentheses

Insert ()

Editor Command

Argument: None
Key sequence: None

Inserts a pair of parentheses, positioning the current point after the opening parenthesis. A prefix argument *p* causes the parentheses to be placed around *p* following (or preceding) forms.

Insert Parentheses For Selection

Editor Command

Argument: None
Key sequence: `Meta+(`

Inserts a pair of parentheses around the selected text, if any. If there is no selected text and a prefix argument *p* is supplied, inserts them around *p* following (or preceding) forms. Otherwise it inserts them at the current point. The point is left on the character after the opening parenthesis.

highlight-matching-parens

Editor Variable

Default value: `t`
Mode: Lisp

When the value is true, matching parentheses are displayed in a different font when the cursor is directly to the right of the corresponding close parenthesis.

Move Over)

Editor Command

Argument: None
Key sequence: `Meta+)`

Inserts a new line after the next closing parenthesis, moving the current point to the new line. Any indentation preceding the closing parenthesis is deleted, and the new line is indented.

Lisp Insert) *Editor Command*

Argument: None
Key sequence: `)`
Mode: Lisp

Inserts a closing parenthesis and highlights the matching opening parenthesis, thereby allowing the user to examine the extent of the parentheses.

Lisp Insert) Indenting Top Level *Editor Command*

Arguments: None
Key sequence: None

The command **`Lisp Insert ) Indenting Top Level`** is the same as **`Lisp Insert )`**, but if it looks like the insertion closes a top level form (when the opening parenthesis is at the beginning of a line) then it also indents the form.

Note: This command is intended as alternative binding to `)` in Lisp mode for users that like this behavior.

Find Unbalanced Parentheses *Editor Command*

Argument: None
Key sequence: None

Moves the point to the end of the last properly matched form, thereby allowing you to easily identify any parentheses in your code which are unbalanced.

`Find Mismatch` is a synonym for **`Find Unbalanced Parentheses`**.

4.8 Documentation

Apropos *Editor Command*

Argument: *string*
Key sequence: **`Ctrl+H A`** *string*

Displays a Symbol Browser tool which lists symbols with symbol names matching *string*. The symbol name at the current point is offered as a default value for *string*.

By default *string* is matched against symbol names as a regular expression. A prefix argument causes a plain substring match to be used instead.

See "Regular expression syntax" on page 107 for a description of regular expression matching. See the *LispWorks IDE User Guide* for a description of the Symbol Browser tool.

Describe Symbol *Editor Command*

Argument: *symbol*
Key sequence: None

Displays a description (that is, value, property list, package, and so on) of *symbol* in a Help window. The symbol under the current point is offered as a default value for *string*. A prefix argument automatically causes this default value to be used.

Function Documentation *Editor Command*

Arguments: None
Key sequence: **`Ctrl+Shift+D`**

`editor:function-documentation-command` ***p***

Prompts for a symbol, which defaults to the symbol at the current point, and displays the HTML documentation for that symbol if it is found in the HTML manuals index pages.

On X11/Motif the prefix argument controls whether a new browser window is created. If the option **Reuse existing browser window** is selected in the browser preferences, then the prefix argument causes the command to create a new browser window. If **Reuse existing browser window** is deselected, then the prefix argument causes the command to reuse an existing browser window.

Show Documentation *Editor Command*

Argument: *name*
Key sequence: `Meta+Ctrl+Shift+A`

Displays a Help window containing any documentation for the Lisp symbol *name* that is present in the Lisp image. This includes function lambda lists, and documentation strings accessible with `cl:documentation`, if any such documentation exists.

Show Documentation for Dspec *Editor Command*

Argument: *dspec*
Key sequence: None

Displays any documentation in the Lisp image for the dspec *dspec*, as described for `Show Documentation`.

dspec is a symbol or list naming a definition, as described in the *LispWorks User Guide and Reference Manual*.

4.9 Evaluation and compilation

The commands described below allow the user to evaluate (interpret) or compile Lisp code that exists as text in a buffer. In some cases, the code may be used to modify the performance of the Editor itself.

4.9.1 General Commands

current-package *Editor Variable*

Default value: `nil`

If non-nil, defines the value of the current package.

Set Buffer Package *Editor Command*

Argument: *package*
Key sequence: None

Set the package to be used by Lisp evaluation and compilation while in this buffer. Not to be used in the Listener, which uses the value of **`*package*`** instead.

Set Buffer Output *Editor Command*

Argument: *stream*
Key sequence: None

Sets the output stream that evaluation results in the current buffer are sent to.

4.9.2 Evaluation commands

Evaluate Defun *Editor Command*

Argument: None
Key sequence: **`Meta+Ctrl+X`**

Evaluates the current top-level form. If the current point is between two forms, the previous form is evaluated.

If the form is a **`defvar`** form, then the command may first make the variable unbound, according to the value of **`evaluate-defvar-action`**, and hence assign the new value. This is useful because **`cl:defvar`** does not reassign the value of a bound variable but when editing a program it is likely that you do want the new value.

evaluate-defvar-action *Editor Variable*

Default value: **`:reevaluate-and-warn`**

This affects the behavior of **`Evaluate Defun`** and **`Compile Defun`** when they are invoked on a **`defvar`** form. The allowed values are:

`:evaluate-and-warn`

Do not make the variable unbound before evaluating the form, and warn that it was not redefined.

`:evaluate` Do not make the variable unbound before evaluating the form, but do not warn that it was not redefined.

`:reevaluate-and-warn`

Make the variable unbound before evaluating the form, and warn that it was therefore redefined.

`:reevaluate`

Make the variable unbound before evaluating the form, but do not warn that it was therefore redefined.

Reevaluate Defvar *Editor Command*

Argument: None
Key sequence: None

Evaluates the current top-level form if it is a **`defvar.`** If the current point is between two forms, the previous form is evaluated. The form is treated as if the variable is not bound.

`Re-evaluate Defvar` is a synonym for **`Reevaluate Defvar`**.

Evaluate Expression *Editor Command*

Argument: *expression*
Key sequence: **`Esc Esc`** *expression*
Key sequence: **`Meta+Esc`** *expression*

Evaluates *expression*. The expression to be evaluated is typed into the Echo Area and the result of the evaluation is displayed there also.

Evaluate Last Form *Editor Command*

Argument: None
Key sequence: **`Ctrl+X Ctrl+E`**

Evaluates the Lisp form preceding the current point.

Without a prefix argument, prints the result in the Echo Area. With a non-nil prefix argument, inserts the result into the current buffer.

Evaluate Region *Editor Command*

Argument: None
Key sequence: `Ctrl+Shift+E`

Evaluates the Lisp forms in the region between the current point and the mark.

Evaluate Buffer *Editor Command*

Argument: None
Key sequence: None

Evaluates the Lisp forms in the current buffer.

Load File *Editor Command*

Argument: *file*
Key sequence: None

Loads *file* into the current eval server, so that all Lisp forms in the file are evaluated.

Toggle Error Catch *Editor Command*

Argument: None
Key sequence: None

Toggles error catching for expressions evaluated in the editor. By default, if there is an error in an expression evaluated in the editor, a Notifier window is opened which provides the user with a number of options, including debug, re-evaluation and aborting of the editor command. However, this behavior can be changed by using `Toggle Error Catch`, so that in the event of an error, the error message is printed in the Echo Area, and the user is given no restart or debug options.

Evaluate Buffer Changed Definitions *Editor Command*

Argument: None
Key sequence: None

Evaluates definitions that have been changed in the current buffer during the current LispWorks session (use **`Buffer Changed Definitions`** on page 177 to see which definitions have changed). A prefix argument equal to the value of **`prefix-argument-default`** causes evaluation of definitions changed since last evaluated. A prefix argument of 1 causes evaluation of definitions changed since last saved.

Evaluate Changed Definitions *Editor Command*

Argument: None
Key sequence: None

Evaluates definitions in all Lisp buffers that have been changed during the current LispWorks session. The effect of prefixes is the same as for **`Evaluate Buffer Changed Definitions`**.

Evaluate System Changed Definitions *Editor Command*

Argument: *system*
Key sequence: None

Evaluates definitions that have been changed in *system* during the current LispWorks session.

4.9.3 Evaluation in Listener commands

Evaluate Defun In Listener *Editor Command*

Argument: *editp*
Key sequence: None

This command works rather like **`Evaluate Defun`** in that it evaluates the current top-level form and handles **`defvar`** forms usefully. However, instead of doing the evaluation in the Editor window, it copies the form into a Listener window as if you had entered it there.

Normally the evaluation is done immediately, but if a prefix argument is given, the text is inserted into the Listener for you to edit before pressing **`Return`** to evaluate it.

An **`in-package`** form is inserted before the form when necessary, so this will change the current package in the Listener.

Evaluate Last Form In Listener *Editor Command*

Argument: *editp*
Key sequence: None

This command works rather like **`Evaluate Last Form`** in that it evaluates the Lisp form preceding the current point. However, instead of doing the evaluation in the Editor window, it copies the form into a Listener window as if you had entered it there.

Normally the evaluation is done immediately, but if a prefix argument is given, the text is inserted into the Listener for you to edit before pressing **`Return`** to evaluate it.

An **`in-package`** form is inserted before the form when necessary, so this will change the current package in the Listener.

Evaluate Region In Listener *Editor Command*

Argument: *editp*
Key sequence: None

This command works rather like **`Evaluate Region`** in that it evaluates the Lisp forms in the current region. However, instead of doing the evaluation in the Editor window, it copies the forms into a Listener window as if you had entered them there.

Normally the evaluation is done immediately, but if a prefix argument is given, the forms are inserted into the Listener for you to edit before pressing **`Return`** to evaluate them.

An **`in-package`** form is inserted before the forms when necessary, so this will change the current package in the Listener.

4.9.4 Compilation commands

Compile Defun *Editor Command*

Argument: None
Key sequence: **`Ctrl+Shift+C`**

Compiles the current top-level form. If the current point is between two forms, the previous form is evaluated.

If the form is a **`defvar`** form, then the command may first make the variable unbound, according to the value of **`evaluate-defvar-action`**, and hence assign the new value.This is useful because **`cl:defvar`** does not reassign the value of a bound variable but when editing a program it is likely that you do want the new value.

Compile Region *Editor Command*

Argument: None
Key sequence: **`Ctrl+Shift+R`**

Compiles the Lisp forms in the region between the current point and the mark.

Compile File *Editor Command*

Argument: *file*
Key sequence: None

Compiles *file* unconditionally, with **`cl:compile-file`**.

No checking is done on write dates for the source and binary files, to see if the file needs to be compiled. Also, no checking is done to see if there is a buffer for the file that should first be saved.

Compile Buffer *Editor Command*

Argument: None
Key sequence: **`Ctrl+Shift+B`**

Reads, compiles and then executes in turn each of the Lisp forms in the current buffer.

Compile Buffer File *Editor Command*

Argument: None
Key sequence: None

Compiles the source file in the current buffer as if by **`Compile File`**, but checks the buffer and file first.

If the buffer is modified it is saved (updating the source file) before compilation, although if **`compile-buffer-file-confirm`** is true the command prompts for confirmation before saving and compiling.

If its associated binary (fasl) file is older than the source file or does not exist or the prefix argument is supplied then the file is compiled, although if **`compile-buffer-file-confirm`** is **`t`** the command prompts for confirmation before compiling.

If the binary file is up to date, command prompts for confirmation before compiling, although this prompt can be avoided by supplying the prefix argument.

Compile and Load Buffer File *Editor Command*

Arguments: None
Key sequence: None

The command **`Compile and Load Buffer File`** compiles the source file in the current buffer just like **`Compile Buffer File`**, with the same checks.

It then loads the compiled file. In the case that the binary file is up to date and the user declines to compile, the command first prompts for confirmation before loading the existing binary file.

Compile and Load File *Editor Command*

Arguments: *filename*
Key sequence: None

The command **`Compile and Load File`** prompts for a filename, and then compiles and loads that file.

compile-buffer-file-confirm *Editor Variable*

Default value: **`t`**

Determines whether **`Compile Buffer File`** should prompt for a compilation to proceed. If the value is true, the user is always prompted for confirmation.

Compile Buffer Changed Definitions *Editor Command*

Argument: None
Key sequence: None

Compiles definitions that have been changed in the current buffer during the current LispWorks session (use **`Buffer Changed Definitions`** to see which definitions have changed). A prefix argument equal to the value of **`prefix-argument-default`** causes compilation of definitions changed since last compiled. A prefix argument of 1 causes compilation of definitions changed since last saved.

Compile Changed Definitions *Editor Command*

Argument: None
Key sequence: None

Compiles definitions in all Lisp buffers that have been changed during the current LispWorks session. The effect of prefixes is the same as for **`Compile Buffer Changed Definitions`**.

Compile System *Editor Command*

Argument: *system*
Key sequence: None

Compiles all files in the system *system*.

If ASDF is loaded and the LispWorks tools are configured to use it, then this command works with ASDF systems as well as those defined by **`lispworks:defsystem`**.

Compile System Changed Definitions *Editor Command*

Argument: *system*
Key sequence: None

Compiles definitions that have been changed in *system* during the current LispWorks session.

Disassemble Definition *Editor Command*

Argument: *definition*
Key sequence: None

Outputs assembly code for *definition* to the Output window, compiling it first if necessary. The name of the current top-level definition is offered as a default value for *definition*.

Edit Recognized Source *Editor Command*

Argument: None
Key sequence: **`Ctrl+X ,`**

Edit the source of the next compiler message, warning or error. It should be used while viewing the Output window. Without a prefix argument, it searches forwards in the Output window until it finds text which it recognizes as a compiler message, warning or error, and then shows the source code associated with that message. With a prefix argument, it searches backwards.

4.10 Code Coverage

These commands allow you to visualize code coverage data by coloring the source code in a LispWorks editor.

4.10.1 Coloring code coverage

By default, these commands call **`hcl:editor-color-code-coverage`** with *for-editing* **`t`**. This means that they find the existing buffer for the file if there is one (always true for **`Code Coverage Current Buffer`**), and do not modify the text at all. When used with a prefix argument, these commands pass *for-editing* **`nil`**, which causes creation of a special buffer without a pathname and different name, and then coloring contains counters.

Code Coverage Current Buffer *Editor Command*

Argument: None
Key sequence: None

Colors the code in the current buffer with code coverage data.

The file named by the buffer pathname of the current buffer needs to have code coverage data in the default code coverage data. This may be set by **`hcl:code-coverage-set-editor-default-data`** or the commands **`Code Coverage Set Default Data`** and **`Code Coverage Load Default Data`**).

If a prefix argument is supplied, then a buffer without a pathname is created with a different name from the source file, which prevents accidental overwriting of the source file.

The actual coloring is done by calling **`hcl:editor-color-code-coverage`**, see the *LispWorks User Guide and Reference Manual* for details.

See also: **`Code Coverage File`**.

Code Coverage File *Editor Command*

Argument: None
Key sequence: None

Prompts for a file, opens and colors it with code coverage data in the same way as **`Code Coverage Current Buffer`**.

See also: **`Code Coverage Current Buffer`**

4.10.2 Setting the default code coverage data

Code Coverage Load Default Data *Editor Command*

Argument: None
Key sequence: None

Sets the default code coverage data that the editor uses to color.

The command prompts for a filename, and passes the result to **`hcl:code-coverage-set-editor-default-data`**.

See also: **`Code Coverage Current Buffer`**

Code Coverage Set Default Data *Editor Command*

Argument: None
Key sequence: None

Sets the default code coverage data that the editor uses to color.

The command prompts for a string, reads and evaluates it, and then passes the result to **`hcl:code-coverage-set-editor-default-data`**.

See also: **`Code Coverage Current Buffer`**

4.11 Breakpoints

These commands operate on breakpoints, which are points in code where execution stops and the LispWorks IDE invokes the Stepper tool.

See "Breakpoints" in the *LispWorks IDE User Guide* for more information about breakpoints and the Stepper tool.

4.11.1 Setting and removing breakpoints

Toggle Breakpoint *Editor Command*

Argument: None
Key sequence: None

If there is no breakpoint at the current point, sets a breakpoint there if possible. If there is a breakpoint at the current point, removes it.

4.11.2 Moving between breakpoints

Next Breakpoint *Editor Command*

Argument: None
Key sequence: None

Moves the point to the next breakpoint in the current buffer. If given a numeric prefix argument *p*, it skips *p*-1 breakpoints.

Previous Breakpoint *Editor Command*

Argument: None
Key sequence: None

Moves the point to the previous breakpoint in the current buffer. If given a numeric prefix argument *p*, it skips *p*-1 breakpoints.

4.12 Stepper commands

Stepper Breakpoint
Stepper Continue
Stepper Macroexpand
Stepper Next
Stepper Restart
Stepper Show Current Source
Stepper Step
Stepper Step Through Call
Stepper Step To Call
Stepper Step To Cursor
Stepper Step To End
Stepper Step To Value
Stepper Undo Macroexpand *Editor Commands*

Arguments: None
Key sequence: None

These commands run the corresponding Stepper command in the current Stepper tool.

See "Stepper controls" in the *LispWorks IDE User Guide* for more information about these commands and the Stepper tool.

4.13 Removing definitions

These commands allow the user to remove definitions from the running Lisp image. It uses Common Lisp functionality such as **`fmakunbound`**, **`makunbound`** and **`remove-method`** to undefine Lisp functions, variables, methods and so on.

Note: This does not mean deleting the source code.

4.13.1 Undefining one definition

Undefine *Editor Command*

Argument: None
Key sequence: None

Without a prefix argument, this undefines the current top level definition. That is, the defining form around or preceding the current point.

With a non-nil prefix argument, this does not undefine the definition but instead inserts into the buffer a Lisp form which, if evaluated, would undefine the definition.

Undefine Command *Editor Command*

Argument: None
Key sequence: None

Prompts for the name of an Editor command, and undefines that command.

4.13.2 Removing multiple definitions

Undefine Buffer *Editor Command*

Argument: None
Key sequence: None

Undefines all the definitions in the current buffer.

Undefine Region *Editor Command*

Argument: None
Key sequence: None

Undefines the definitions in the current region.

4.14 Remote debugging

Connect Remote Debugging *Editor Command*

Arguments: *host port*
Key sequence: None

Connects to a remote client for remote debugging. Without a prefix argument, also immediately open a Listener.

Reconnect Remote Listener *Editor Command*

Arguments: None
Key sequence: None

Reconnects a Remote Listener to a remote client. It can be used only in a Remote Listener after the client side has disconnected, which may be either because the read-eval-print loop on the client side exited, or the connection was closed (which may also be because the client crashed). The command tries to reconnect the Listener to the same client, which can work if the connection is still open, if there is another connection to the same client, or if the client is listening for connections.

Remote Evaluate Buffer *Editor Command*

Arguments: None
Key sequence: None

Evaluates, in the remote client, the Lisp forms in the current buffer.

Remote Evaluate Region *Editor Command*

Arguments: None
Key sequence: None

Evaluates, in the remote client, the Lisp forms in the current region.

Remote Evaluate Defun *Editor Command*

Arguments: None
Key sequence: None

Evaluates, in the remote client, the current top level form.

Remote Evaluate Last Form *Editor Command*

Arguments: None
Key sequence: None

Evaluates, in the remote client, the Lisp form preceding the current point.

Remote Evaluate Region In Listener *Editor Command*

Arguments: None
Key sequence: None

Evaluates, in a Remote Listener, the Lisp forms in the current region.

Remote Evaluate Defun In Listener *Editor Command*

Arguments: None
Key sequence: None

Evaluates, in a Remote Listener, the current top level form.

Remote Evaluate Last Form In Listener *Editor Command*

Arguments: None
Key sequence: None

Evaluates, in a Remote Listener, the Lisp form preceding the current point.

Set Default Remote Debugging Connection *Editor Command*

Arguments: None
Key sequence: None

Sets the default remote debugging connection.

5 Emulation

By default the LispWorks Editor emulates GNU Emacs. This is often unusable for programmers familiar only with Mac OS X keys and behavior: for instance, a selection is not deleted on input, and most of the commonly used keys differ.

The LispWorks editor can be switched to emulate the Mac OS X model instead of Emacs.

When using Mac OS X editor emulation the main differences are:

- An alternate set of key bindings for the commonly-used commands.
- The abort gesture for the current editor command is **`Esc`**, not **`Ctrl+G`**.
- Inserted text replaces any currently selected text.
- The cursor is a vertical bar rather than a block.

5.1 Using Mac OS X editor emulation

To switch Mac OS X editor emulation on, use **Preferences... > Environment > Emulation**. See the section "Configuring the editor emulation" in the *LispWorks IDE User Guide* for details.

5.2 Key bindings

The key bindings for Mac OS X editor emulation are supplied in the LispWorks library file `config/mac-key-binds.lisp`. This file is loaded the first time that you use Mac OS X editor emulation, or on startup if your preference is stored.

5.2.1 Finding the keys

There are several ways to find the key for a given command, and the command on a given key:

- The files `mac-key-binds.lisp` and `selection-key-binds.lisp` show the default state, just like `key-binds.lisp` shows the Emacs bindings.
- The Editor command `Describe Bindings` shows all the current key bindings, including those specific to the buffer, the major mode and any minor modes that are in effect.
- The Editor command `Describe Key` reports the command on a given key.
- The Editor command `Where Is` reports the key for a given command.
- Use the `Help > Editing` menu.

5.2.2 Modifying the Key Bindings

As in Emacs emulation, the key sequences to which individual commands are bound can be changed, and key bindings can be set up for commands which are not, by default, bound to any key sequences.

Interactive means of modifying key bindings are described in “Key bindings” on page 136. Key bindings can also be defined programmatically via `editor:bind-key` forms similar to those in `mac-key-binds.lisp`.

However, note that you must use `editor:set-interrupt-keys` if you wish to alter the abort gesture.

5.2.3 Accessing Emacs keys

When Mac OS X editor emulation is on, most Emacs keys are still available since keystrokes like `Ctrl+X` and `Ctrl+S` do not clash with standard Mac OS X bindings. For example, to invoke the command `WFind File`, simply enter:

```
Ctrl+X Ctrl+F
```

If you have chosen not to have an Emacs Meta key (see “Using Mac OS X editor emulation”) you can use **`Ctrl+M`** instead. For example, to run the command **`Skip Whitespace`**, enter:

```
Ctrl+M X Skip Whitespace
```

5.2.4 The Alt modifier and editor bindings

In Microsoft Windows emulation on Microsoft Windows, keystrokes with the **`Alt`** modifier key are used by the system to activate the menu bar. Therefore these keystrokes, for example **`Alt+A`** and **`Alt+Ctrl+A`** are not available to the editor.

Windows accelerators always take precedence over editor key bindings, so in Emacs emulation the **`Alt`** modifier key only acts as Meta though keystrokes with **`Alt`** if there is no accelerator which matches.

On Cocoa, the preference for the Meta key affects the operation of menu accelerators (shortcuts). If **`Command`** is used as Meta, then it will not be available for use as an accelerator.

5.3 Replacing the current selection

When using Mac OS X editor emulation, Delete Selection Mode is active so that selected text is deleted when you type or paste text. Also, **`Delete`** deletes the current selection.

Note: Delete Selection Mode can also be used independently of Mac OS X editor emulation. See “Delete Selection” on page 76 for details.

5.4 Emulation in Applications

If you include the LispWorks editor (via **`capi:editor-pane`** or its subclasses) in an application, then by default your interfaces will use Microsoft Windows emulation on Windows, Mac OS X editor emulation on Cocoa, and Emacs emulation on Linux and other Unix-like systems.

To override this behavior in your interface classes, define a method on **`capi:interface-keys-style`**. See the *CAPI User Guide and Reference Manual* for details.

To override this behavior in your delivered application, use the delivery keyword **`:editor-style`**. See the *LispWorks Delivery User Guide* for details.

6

Advanced Features

The editor can be customized, both interactively and programmatically, to suit the users requirements.

The chapter “Command Reference” provides details of commands used to customize the editor for the duration of an editing session (see “Keyboard macros” on page 127, “Key bindings” on page 136, “Editor variables” on page 135). This chapter contains information on customizing the editor on a permanent basis.

There are a number of ways in which the editor may be customized:

- The key sequences to which individual commands are bound can be changed, and key bindings can be set up for commands which are not, by default, bound to any key sequences—see “Customizing default key bindings” on page 214.
- The indentation used for Lisp forms can be modified to suit the preferences of the user—see “Customizing Lisp indentation” on page 216.
- Additional editor commands can be created by combining existing commands and providing specified arguments for them—see “Programming the editor” on page 216.

Note that the default configuration files mentioned in this chapter were used when LispWorks was released. They are not read in when the system is run, so any modification to them will have no effect. If the user wishes to modify the behavior of LispWorks in any of these areas, the modifying code should be included in the **.lispworks** file, or an image containing the modifications should be saved.

6.1 Customizing default key bindings

The key sequences to which individual commands are bound can be changed, and key bindings can be set up for commands which are not, by default, bound to any key sequences. Interactive means of modifying key bindings are described in “Key bindings” on page 136.

This section describes the editor function **bind-key**, which is used to establish bindings programmatically. If you want to alter your personal key bindings, put the modifying code in your **.lispworks** file.

The default Emacs key bindings can be found in the file **config/key-binds.lisp** in the LispWorks library directory. See “Key bindings” for details of the key binds files used in other editor emulations.

editor:bind-key *Function*

editor:bind-key *name key* &optional *kind where*

Binds the command *name* to the key sequence or combination *key*.

kind can take the value **:global**, **:mode**, or **:buffer**.

The default for *kind* is **:global**. which makes the binding apply in all buffers and all modes, unless overridden by a mode-specific or buffer-specific binding.

If *where* is not supplied, the binding is for the current emulation. Otherwise *where* should be either **:emacs** or **:mac**, meaning that the binding is for Emacs emulation or Mac OS editor emulation respectively.

Note: before the editor starts, the current emulation is **:emacs**. Therefore **bind-key** forms which do not specify *where* and which are evaluated before the editor starts (for example, in your initialization file) will apply to Emacs emulation only. Thus for example

```
(bind-key "Command" "Control-Right")
```

when evaluated in your initialization file will establish an Emacs emulation binding. The same form when evaluated after editor startup will establish a binding in the current emulation: Emacs or Mac OS editor emulation.

It is best to specify the intended emulation:

```
(editor:bind-key "Command" "Control-Right" :global :mac)
```

If *kind* is **`:buffer`** the binding applies only to a buffer which should be specified by the value of *where*.

If *kind* is **`:mode`** the binding applies only to a mode which should be specified by *where*.

If this function is called interactively via the command **`Bind Key`**, you will be prompted as necessary for the kind of binding, the buffer or the mode. The binding is for the current emulation. **`Tab`** completion may be used at any stage.

The following examples, which are used to implement some existing key bindings, illustrate how key sequences can be specified using **`bind-key`**.

```
(editor:bind-key "Forward Character" "Control-f")
(editor:bind-key "Forward Word" "Meta-f")
(editor:bind-key "Save File" #("Control-x" "Control-s"))
(editor:bind-key "ISearch Forward Regexp" "Meta-Control-s")
(editor:bind-key "Complete Field" #\space :mode "Echo Area")
(editor:bind-key "Backward Character" "left")
(editor:bind-key "Forward Word" #("control-right"))
```

editor:bind-string-to-key *Function*

`editor:bind-string-to-key` ***string key*** **`&optional`** ***kind where***

Binds the text string *string* to the keyboard shortcut *key* without the need to create a command explicitly. Using *key* inserts *string* in the current buffer. The *kind* and *where* arguments are as for **`editor:bind-key`**.

editor:set-interrupt-keys *Function*

`editor:set-interrupt-keys` ***keys*** **`&optional`** ***input-style***

The key that aborts the current editor command is handled specially by the editor. If you wish to change the default (from `Ctrl+G` for Emacs) then you must use this function rather than **`editor:bind-key`**. See the file **`config/mac-key-binds.lisp`** for an example.

6.2 Customizing Lisp indentation

The indentation used for Lisp forms can be modified to suit the preferences of the user.

The default indentations can be found in the file **`config/indents.lisp`** in the LispWorks library directory. If you want to alter your personal Lisp indentation, put the modifying code in your **`.lispworks`** file.

editor:setup-indent *Function*

`editor:setup-indent` ***form-name no-of-args*** **`&optional`** ***standard special***

Modifies the indentation, in Lisp Mode, for the text following an instance of *form-name*. The arguments *no-of-args*, *standard* and *special* should all be integers. The first *no-of-args* forms following the *form-name* become indented *special* spaces if they are on a new line. All remaining forms within the scope of the *form-name* become indented *standard* spaces.

For example, the default indentation for **`if`** in Lisp code is established by:

```
(editor:setup-indent "if" 2 2 4)
```

This determines that the first 2 forms after the **`if`** (that is, the **`test`** and the **`then`** clauses) get indented 4 spaces relative to the **`if`**, and any further forms (here, just an **`else`** clause) are indented by 2 spaces.

6.3 Programming the editor

The editor functions described in this section can be combined and provided with arguments to create new commands.

Existing editor commands can also be used in the creation of new commands. Every editor command documented in this manual is named by a string *command* which can be used to invoke the command interactively, but there is also associated with this a standard Lisp function (the "command function") named by a symbol exported from the editor package. You can use this symbol to call the command programmatically. For example, the editor command **`Forward Character`** is referred to by **`editor:forward-character-command`**.

The first argument of any command function is the prefix argument *p*, and this must therefore be included in any programmatic call, even if the prefix argument is ignored. Some commands have additional optional arguments. For example to insert 42 #\! characters, you would call

```
(editor:self-insert-command 42 #\!)
```

Details of these optional arguments are provided in the command descriptions throughout this manual.

See **`editor:defcommand`** for the details of how to create new commands.

Note: code which modifies the contents of a **`capi:editor-pane`** (for example a displayed editor buffer) must be run only in the interface process of that pane.

The following sections describe editor functions that are not interactive editor commands.

6.3.1 Calling editor functions

All editor commands and some other editor functions expect to be called within a dynamic context that includes settings for the current buffer and current window. This happens automatically when using the editor interactively.

You can set up the context in a CAPI application by using the function **`capi:call-editor`** (see the *CAPI User Guide and Reference Manual*).

You can also use the following function to call editor commands and functions.

editor:process-character *Function*

`editor:process-character` ***char window***

Processes *char* in a dynamic context where the current window is *window* and the current buffer is the buffer currently displayed in *window*.

The *char* can be one of the following:

- A string, naming an editor command to invoke.
- A list of the form `(`***function*** `.` ***args***`)`, which causes *function* to be called with *args*. The items in *args* are not evaluated.
- A function or symbol, which is called with `nil` as its argument (like a command function would be if there is no prefix argument).
- A character or `system:gesture-spec` object, which is treated as if it has been typed on the keyboard.

There is no return value. The processing may happen in another thread, so may not have competed before this function returns.

6.3.2 Defining commands

editor:defcommand *Macro*

`defcommand` ***name lambda-list command-doc function-doc*** `&body` ***forms*** `=>` ***command-function***

Defines a new editor command. *name* is a usually string naming the new editor command which can invoked in the editor via **`Extended Command`**, and *command-function* is a symbol naming the new command function which can be called programmatically. The *command-function* symbol is interned in the current package.

lambda-list is the lambda list of the new command, which must have at least one argument which is usually denoted *p*, the prefix argument.

command-doc and *function-doc* should be strings giving detailed and brief descriptions of the new command respectively.

forms is the Lisp code for the command.

The name of the command must be a string, while the name of the associated command function must be a symbol. There are two ways in which *name* can be supplied. Most simply, *name* is given as a string, and the string is taken to be the name of the editor command. The symbol naming the command function is computed from that string: spaces are replaced with hyphens and alphabetic characters are uppercased, but otherwise the symbol name contains the same characters as the string with `-COMMAND` appended.

If a specific function name, different to the one `defcommand` derives itself, is required, then this can be supplied explicitly. To do this, *name* should be a list: its first element is the string used as the name of the command, while its second and last element is the symbol used to name the Lisp command function.

For example the following code defines an editor command, `Move Five`, which moves the cursor forward in an editor buffer by five characters.

```
(editor:defcommand "Move Five" (p)
   "Moves the current point forward five characters.
    Any prefix argument is ignored."
   "Moves five characters forward."
   (editor:forward-character-command 5))
=>
MOVE-FIVE-COMMAND
```

The prefix argument *p* is not used, and is there simply because the *lambda-list* must have at least one element.

Use `Meta+X Move Five` to invoke the command.

As another example this command changes all the text in a writable buffer to be uppercase:

```
(editor:defcommand "Uppercase Buffer" (p)
     "Uppercase the buffer contents" ""
  (declare (ignore p))
  (let* ((buffer (editor:current-buffer))
         (point (editor:buffer-point buffer))
         (start (editor:buffers-start buffer))
         (end (editor:buffers-end buffer)))
    (editor:set-current-mark start)
    (editor:move-point point end)
    (editor:uppercase-region-command nil)))
```

Having defined your new command, you can invoke it immediately by `Meta+X Uppercase Buffer`.

You could also call it programmatically:

```
(uppercase-buffer-command nil)
```

If you anticipate frequent interactive use of **`Uppercase Buffer`** you will want to bind it to a key. You can do this interactively for the current session using **`Bind Key`**. Also you can put something like this in your initialization file to establish the key binding for each new session:

```
(editor:bind-key "Uppercase Buffer" #("Control-x" "Meta-u"))
```

Then, entering **`Ctrl+X Meta+U`** will invoke the command.

Define Command Synonym *Editor Command*

Arguments: *new-name*, *command-name*
Key sequence: None

The command **`Define Command Synonym`** prompts for a string and an existing command name, and makes the string be a synonym for the existing command name.

6.3.3 Buffers

Each buffer that you manipulate interactively using editor commands is an object of type **`editor:buffer`** that can be used directly when programming the editor. Buffers contain an arbitrary number of **`editor:point`** objects, which are used when examining or modifying the text in a buffer (see “Points” on page 227).

6.3.3.1 Buffer locking

Each buffer contains a lock that is used to prevent more than one thread from modifying the text, text properties or points within the buffer simultaneously. All of the exported editor functions (**`editor:insert-string`**, **`editor:move-point`** etc) claim this lock implicitly and are therefore atomic with respect to other such functions.

In situations where you want to make several changes as one atomic operation, use one of the macros **`editor:with-buffer-locked`** or **`editor:with-point-locked`** to lock the buffer for the duration of the operation. For example, if you want to delete the next character and replace it by a space:

```
(editor:with-buffer-locked ((editor:current-buffer))
  (editor:delete-next-character-command nil)
  (editor:insert-character (editor:current-point)
                           #\Space))
```

In addition, you sometimes want to examine the text in a buffer without changing it, but ensure that no other thread can modify it in the meantime. This can be achieved by locking the buffer using **`editor:with-buffer-locked`** or **`editor:with-point-locked`** and passing the *for-modification* argument as **`nil`**. For example, if you are computing the beginning and end of some portion of the text in a buffer and then performing some operation on that text, you may want to lock the buffer to ensure that no other threads can modify the text while your are processing it.

editor:with-buffer-locked *Macro*

`editor:with-buffer-locked` (***buffer*** **`&key`** ***for-modification check-file-modification block-interrupts***) **`&body`** ***body*** **`=>`** ***values***

Evaluates *body* while holding the lock in *buffer.* At most one thread can lock a buffer at a time and the macro waits until it can claim the lock.

If *for-modification* is non-nil (the default), the contents of *buffer* can be modified by *body*. If *for-modification* is **`nil`**, the contents of *buffer* cannot be modified until *body* returns and trying to do so from within *body* will signal an error. If the buffer is read-only and *for-modification* is non-nil, then an **`editor:editor-error`** is signaled. The status of the lock can be changed to *for-modification* (see **`editor:change-buffer-lock-for-modification`**). If the buffer is read-only, an **`editor:editor-error`** occurs if *for-modification* is **`t`**.

The macro **`editor:with-buffer-locked`** can be used recursively, but if the outermost use passed **`nil`** as the value of *for-modification*, then inner uses cannot pass non-nil as the value of *for-modification*, unless **`editor:change-buffer-lock-for-modification`** is used to change the lock status.

If *check-file-modification* is non-nil (the default) and the buffer is associated with a file and has not already been modified, then the modification time of the file is compared to the time that the file was last read. If the file is newer than the buffer, then the user is asked if they want to re-read the file into the buffer, and if they do then the file is re-read and the operations aborts. Otherwise, there is no check for the file being newer than the buffer.

If *block-interrupts* is non-nil, the body is evaluated with interrupts blocked. This is useful if the buffer may be modified by an interrupt function, or some interrupt function may end up waiting for another thread that may wait for the buffer lock, which would cause a deadlock. The default is not to block interrupts.

Note that using a non-nil value for *block-interrupts* is not the same as using the **`without-interrupts`** or **`without-preemption`** macros. It just stops the current thread from calling interrupt functions, so other threads might run while the body is being evaluated.

The *values* returned are those of *body*.

editor:with-point-locked *Macro*

`editor:with-point-locked` `(`***point*** `&key` ***for-modification check-file-modification block-interrupts errorp***`)` `&body` ***body*** `=>` ***values***

Evaluates *body* while holding the lock in the buffer that is associated with *point*. In addition, the macro checks that *point* is valid and this check is atomic with respect to calls to the function **`editor:delete-point`**. The values of *for-modification*, *check-file-modification* and *block-interrupts* have the same meanings as for **`editor:with-buffer-locked`**.

The value of *errorp* determines the behavior when *point* is not valid. If *errorp* is non-nil, an error is signaled, otherwise **`nil`** is returned without evaluating *body*. The point may be invalid because it does not reference any buffer (that is, it has been deleted), or because its buffer was changed by another thread while the current thread was attempting to lock the buffer.

The *values* returned are those of *body*, or **`nil`** when *errorp* is **`nil`** and *point* is not valid.

editor:change-buffer-lock-for-modification *Function*

`editor:change-buffer-lock-for-modification` ***buffer*** **`&key`** ***check-file-modification force-modification*** **`=>`** ***result***

Changes the status of the lock in the buffer *buffer* to allow modification of the text. *buffer* must already be locked for non-modification by the current thread (that is, it must be dynamically within a **`editor:with-buffer-locked`** or **`editor:with-point-locked`** form with *for-modification* **`nil`**).

buffer	An editor buffer.
check-file-modification	A boolean.
force-modification	A boolean.
result	**`:buffer-not-locked`**, **`:buffer-out-of-date`** or **`:buffer-not-writable`**.

If *check-file-modification* is non-nil, the same test as described for **`editor:with-buffer-locked`** is performed, and if the file has been modified then **`:buffer-out-of-date`** is returned without changing anything (it does not prompt the user to re-read the file).

The default value of *check-file-modification* is **`t`**.

force-modification controls what happens if the buffer is read-only. If *force-modification* is **`nil`**, the function returns **`:buffer-not-writable`** and does nothing. If it is non-nil, the status is changed. The buffer remains read-only.

result is **`nil`** if the status of the locking was changed to *for-modification*, or if the status of the buffer lock was already *for-modification*. Otherwise, *result* is a keyword indicating why the status could not be changed. When *result* is non-nil, the status of the locking remains unchanged.

The returned value can be be one of:

`:buffer-not-locked`

The buffer is not locked by the current thread.

`:buffer-not-writable`

The buffer is not writable, and *force-modification* is `nil`.

`:buffer-out-of-date`

The file that is associated with the buffer was modified after it was read into the editor, the buffer is not modified, and *check-file-modification* is non-nil.

6.3.3.2 Buffer operations

editor:*buffer-list* *Variable*

Contains a list of all the buffers in the editor.

editor:current-buffer *Function*

`editor:current-buffer`

Returns the current buffer.

editor:buffer-name *Function*

`editor:buffer-name` ***buffer***

Returns the name of *buffer*.

editor:window-buffer *Function*

`editor:window-buffer` ***window***

Returns the buffer currently associated with *window*.

editor:buffers-start *Function*

`editor:buffers-start` ***buffer***

Returns the starting point of *buffer*.

editor:buffers-end *Function*

`editor:buffers-end` ***buffer***

Returns the end point of *buffer*.

editor:buffer-point *Function*

`editor:buffer-point` ***buffer***

Returns the current point in *buffer*.

editor:use-buffer *Macro*

`editor:use-buffer` ***buffer*** `&body` ***forms***

Makes *buffer* the current buffer during the evaluation of *forms*.

editor:buffer-from-name *Function*

`editor:buffer-from-name` ***name***

Returns the buffer called *name* (which should be a string). If there is no buffer with that name, `nil` is returned.

editor:make-buffer *Function*

`make-buffer` ***name*** `&key` ***modes contents temporary base-name name-pattern***

Creates or returns an existing buffer.

name should be a string or `nil`.

modes should be a list of strings naming modes. The first mode must be a major mode, and the rest minor modes. The default value of *modes* is the value of `default-modes`.

base-name should be a string or `nil`. If *name* and *temporary* are both `nil` then *base-name* must be a string.

contents should be a string, `nil` or `t` (default value `nil`).

temporary is a boolean (default value `nil`).

name-pattern should be a string (default value `"~a<~a>"`).

When *name* is non-nil, it is the name of the buffer. If there is already a buffer with this name which is not temporary and the *temporary* argument is `nil`, `make-buffer` returns that buffer. Before doing so, it sets its contents to *contents* unless *contents* is `t`. When *contents* is `nil`, the buffer is made empty.

If *name* is `nil` or *temporary* is non-nil or a buffer with the name cannot be found, then a new buffer is made and returned. The buffer's contents is set to *contents* if *contents* is a string, and otherwise the buffer is made empty. The name of the buffer is set to *name* if *name* is non-nil.

If *temporary* is `nil`, the buffer is added to the internal tables of the editor. If *name* is non-nil, it is used. Otherwise `make-buffer` tries to use *base-name*. If there is already a buffer with this name, it constructs another name by

```
(format nil name-pattern base-name n)
```

with different integers *n* until it constructs an unused name, which it uses as the buffer's name.

If *temporary* is non-nil, the buffer is not added to the internal tables. It is also marked as temporary, which mainly means that it does not have auto-save and backup files, and avoids calling general hooks when it is modified.

Notes:

Using `:temporary t` gives you a buffer that is 'yours', that is the editor does not do anything with it except in response to explicit calls from your code. Except when actually editing files, this is the most useful way of using buffers in most cases.

`capi:editor-pane` with the `:buffer :temp` initarg uses

```
(make-buffer ... :temporary t)
```

editor:goto-buffer *Function*

`editor:goto-buffer` ***buffer in-same-window***

Makes *buffer* the current buffer. If *buffer* is currently being shown in a window then the cursor is moved there. If *buffer* is not currently in a window and *in-same-window* is non-nil then it is shown in the current window, otherwise a new window is created for it.

editor:clear-undo *Function*

`editor:clear-undo` ***buffer***

Clears any undo information in the buffer *buffer*.

6.3.4 Points

Locations within a buffer are recorded as **`editor:point`** objects. Each point remembers a character position within the buffer and all of the editor functions that manipulate the text of a buffer locate the text using one or more point objects (sometimes the current point).

A point's *kind* controls what happens to the point when text in the buffer is inserted or deleted.

`:temporary` points are for cases where you need read-only access to the buffer. They are like GNU Emacs "points". They have a lower overhead than the other kinds of point and do not need to be explicitly deleted, but do not use them in cases where you make a point, insert or delete text and then use the point again, since they do not move when the text is changed. Also, do not use them in cases where more than one thread can modify their buffer without locking the buffer first (see “Buffer locking” on page 220)

`:before-insert` and **`:after-insert`** points are for cases where you need to make a point, insert or delete text and still use the point afterwards. They are like GNU Emacs "markers". The difference between these two kinds is what happens when text is inserted. For a point at position *n* from the start of the buffer, inserting *len* characters will leave the point at either position *n* or *n+len* according to the following table.

Table 6.1 Editor point positions after text insertion

kind	Insert at < *n*	Insert at = *n*	Insert at > *n*
`:before-insert`	*n+len*	*n*	*n*
`:after-insert`	*n+len*	*n+len*	*n*

When text is deleted, **`:before-insert`** and **`:after-insert`** points are treated the same: points <= the start of the deletion remain unchanged, points >= the end of the deletion are moved with the text and points within the deleted region are automatically deleted and cannot be used again.

All points with kind other than `:temporary` are stored within the data structures of the editor buffer so they can be updated when the text changes. A point can be removed from the buffer by `editor:delete-point`, and point objects are also destroyed if their buffer is killed.

editor:point-kind *Function*

`editor:point-kind` ***point***

Returns the kind of the point, which is `:temporary`, `:before-insert` or `:after-insert`.

editor:current-point *Function*

`editor:current-point`

Returns the current point. See also `editor:buffer-point`.

editor:current-mark *Function*

`editor:current-mark &optional` ***pop-p no-error-p***

Returns the current mark. If *pop-p* is `t`, the mark ring is rotated so that the previous mark becomes the current mark. If no mark is set and *no-error-p* is `t`, `nil` is returned; otherwise an error is signaled. The default for both of these optional arguments is `nil`.

editor:set-current-mark *Function*

`editor:set-current-mark` ***point***

Sets the current mark to be *point*.

editor:point< *Function*

`editor:point<` ***point1 point2***

Returns non-nil if *point1* is before *point2* in the buffer.

editor:point<= *Function*

`editor:point<=` ***point1 point2***

Returns non-nil if *point1* is before or at the same offset as *point2* in the buffer.

editor:point> *Function*

`editor:point>` ***point1 point2***

Returns non-nil if *point1* is after *point2* in the buffer.

editor:point>= *Function*

`editor:point>=` ***point1 point2***

Returns non-nil if *point1* is after or at the same offset as *point2* in the buffer.

editor:copy-point *Function*

`editor:copy-point` ***point*** **`&optional`** ***kind new-point***

Makes and returns a copy of *point*. The argument *kind* can take the value **`:before`**, **`:after`**, or **`:temporary`**. If *new-point* is supplied, the copied point is bound to that as well as being returned.

editor:delete-point *Function*

`editor:delete-point` ***point***

Deletes the point *point*.

This should be done to any non-temporary point which is no longer needed.

editor:move-point *Function*

`editor:move-point` ***point new-position***

Moves *point* to *new-position*, which should itself be a point.

editor:start-line-p *Function*

`editor:start-line-p` ***point***

Returns **`t`** if *point* is immediately before the first character in a line, and **`nil`** otherwise.

editor:end-line-p *Function*

`editor:end-line-p` ***point***

Returns **`t`** if *point* is immediately after the last character in a line, and **`nil`** otherwise.

editor:same-line-p *Function*

`editor:same-line-p` ***point1 point2***

Returns **`t`** if *point1* and *point2* are on the same line, and **`nil`** otherwise.

editor:save-excursion *Macro*

`editor:save-excursion &rest` ***body***

Saves the location of the point and the mark and restores them after completion of *body*. This restoration is accomplished even when there is an abnormal exit from *body*.

editor:with-point *Macro*

`editor:with-point` ***point-bindings*** **`&rest`** ***body***

point-bindings is a list of bindings, each of the form **`(`*****var point*** **`[`*****kind*****`])`**. Each variable *var* is bound to a new point which is a copy of the point *point* though possibly with a different kind, if *kind* is supplied. If *kind* is not supplied, then the new point has *kind* **`:temporary`**.

The forms of *body* are evaluated within the scope of the point bindings, and then the points in each variable *var* are deleted, as if by **`editor:delete-point`**. Each point *var* is deleted even if there was an error when evaluating *body*.

The main reason for using **`with-point`** to create non-temporary points is to allow *body* to modify the buffer while keeping these points up to date for later use within *body*.

6.3.5 Regular expression searching

editor:regular-expression-search

`regular-expression-search` ***point pattern*** **`&key`** ***forwardp prompt limit to-end brackets-limits*** **`=>`** ***match-len*****`,`** ***brackets-limits-vector***

Search for *pattern* starting from *point*.

point must be an **`editor:point`** object or **`nil`**, meaning the result of calling **`editor:current-point`**.

pattern can be a string, a "precompiled" pattern (result of **`lw:precompile-regexp`**), or **`nil`**.

forwardp is a boolean (default value **`t`**) specifying the direction to search.

prompt is a string used to prompt for a pattern when *pattern* is **`nil`**.

limit should be **`nil`** or an **`editor:point`** specifying a limit for the search.

to-end is a boolean (default value **`t`**), specifying whether to move the point to the end of the match when searching forward.

brackets-limits is a boolean specifying whether **`regular-expression-search`** should return a vector of brackets-limits.

`regular-expression-search` performs a search starting from *point* for the *pattern*, in the direction specified by *forwardp*, up to to *limit* if specified, or the buffer's end (when *forwardp* is non-nil) or the buffer's start (when *forwardp* is **`nil`**). If it succeeds, it then moves the point, either to the end of that match when both *forwardp* and *to-end* are non-nil (the default), or to the beginning of the match.

When *pattern* is non-nil it must be either a string or a precompiled pattern created with **`lw:precompile-regexp`**. If *pattern* is a string, **`regular-expression-search`** "precompiles" it before searching, so using a precompiled pattern is more efficient when using the same pattern repeatedly.

If *pattern* is `nil`, `regular-expression-search` first prompts for a pattern in the echo area, using the *prompt*. If *pattern* is non-nil, *prompt* is ignored.

Return values: If `regular-expression-search` is successful, it returns the length of the string that it matched, and if *brackets-limits* is non-nil, a second value which is a vector of the limits of the matches of each `\(` and `\)` pair in the pattern. The meaning of the vector is described in the manual entry for `lw:find-regexp-in-string` in the *LispWorks User Guide and Reference Manual*.

Compatibility note: `regular-expression-search` was exported but not documented in LispWorks 6.1 and earlier versions. *brackets-limits* was introduced in LispWorks 7.0.

See also:
`lw:find-regexp-in-string` , `lw:regexp-find-symbols` and `lw:pre-compile-regexp` in the *LispWorks User Guide and Reference Manual*
"Regular expression syntax" on page 107

6.3.6 The echo area

editor:message *Function*

`editor:message` ***string*** **`&rest`** ***args***

A message is printed in the Echo Area. The argument *string* must be a string, which may contain formatting characters to be interpreted by `format`. The argument *args* consists of arguments to be printed within the string.

editor:clear-echo-area *Function*

`editor:clear-echo-area` **`&optional`** ***string force***

Clears the Echo Area. The argument *string* is then printed in the Echo Area. If *force* is non-nil, the Echo Area is cleared immediately, with no delay. Otherwise, there may be a delay for the user to read any existing message.

6.3.7 Editor errors

Many editor commands and functions signal an error on failure (using **editor:editor-error** as described below). This causes the current operation to be aborted.

In many cases, the user will not want the operation to abort completely if one of the editor commands it uses is not successful. For example, the operation may involve a search, but some aspects of the operation should continue even if the search is not successful. To achieve this, the user can catch the **editor:editor-error** using a macro such as **handler-case**.

For example, one part of an application might involve moving forward 5 forms. If the current point cannot be moved forward five forms, generally the Editor would signal an error. However, this error can be caught. The following trivial example shows how a new message could be printed in this situation, replacing the system message.

```
(editor:defcommand "Five Forms" (p)
   "Tries to move the current point forward five forms,
    printing out an appropriate message on failure."
   "Tries to move the current point forward five forms."
   (handler-case
      (editor:forward-form-command 5)
      (editor:editor-error (condition)
         (editor:message "could not move forward five"))))
```

editor:editor-error *Function*

editor:editor-error ***string*** **&rest** ***args***

By default this prints a message in the Echo Area, sounds a beep, and exits to the top level of LispWorks, aborting the current operation. The argument *string* must be a string, which is interpreted as a control string by **format**. As with **editor:message**, *args* can consist of arguments to be processed within the control string.

The behavior is affected by **break-on-editor-error**.

6.3.8 Files

editor:find-file-buffer *Function*

`editor:find-file-buffer` ***pathname*** `&optional` ***check-function***

Returns a buffer associated with the file *pathname*, reading the file into a new buffer if necessary. The second value returned is `T` if a new buffer is created, and `nil` otherwise. If the file already exists in a buffer, its consistency is first checked by means of *check-function*. If no value is supplied for *check-function*, `editor:check-disk-version-consistent` is used.

editor:fast-save-all-buffers *Function*

`editor:fast-save-all-buffers &optional` ***ask***

Saves all modified buffers which are associated with a file. If *ask* is non-nil then confirmation is asked for before saving each buffer. If *ask* is not set, all buffers are saved without further prompting.

Unlike the editor command `Save All Files` this function can be run without any window interaction. It is thus suitable for use in code which does not intend to allow the user to leave any buffers unsaved, and from the console if it is necessary to save buffers without re-entering the full window system.

editor:check-disk-version-consistent *Function*

`editor:check-disk-version-consistent` ***pathname buffer***

Checks that the date of the file *pathname* is not more recent than the last time *buffer* was saved. If *pathname* is more recent, the user is prompted on how to proceed. Returns `t` if there is no need to read the file from disk and `nil` if it should be read from disk.

editor:buffer-pathname *Function*

`editor:buffer-pathname` ***buffer***

Returns the pathname of the file associated with *buffer*. If no file is associated with *buffer*, `nil` is returned.

6.3.8.1 File encodings in the editor

In an application which writes editor buffers to file, you can do this to set the external format of a given buffer:

```
 (setf (editor:buffer-external-format buffer) ef-spec)
```

You can also set a global default external format for editor buffers:

```
(setf (editor:variable-value 'editor:output-format-default
                             :global)
      ef-spec)
```

Then *ef-spec* will be used when a buffer itself does not have an external format.

See “Unicode and other file encodings” on page 32 for a full description of the editor’s file encodings interface.

6.3.9 Inserting text

editor:insert-string *Function*

`editor:insert-string` ***point string*** **`&optional`** ***start end***

Inserts *string* at *point* in the current buffer. The arguments *start* and *end* specify the indices within *string* of the substring to be inserted. The default values for *start* and *end* are 0 and `(length` *string*`)` respectively.

editor:kill-ring-string *Function*

`editor:kill-ring-string &optional` ***index***

Returns either the topmost string on the kill ring, or the string at *index* places below the top when *index* is supplied.

The editor kill ring stores the strings copied by the editor, in order to allow using them later.

editor:points-to-string *Function*

`editor:points-to-string` ***start end***

Returns the string between the points *start* and *end*.

6.3.10 Indentation

editor:*indent-with-tabs* *Variable*

Controls whether indentation commands such as **`Indent`** and **`Indent Form`** insert whitespace using **`#\Space`** or **`#\Tab`** characters when changing the indentation of a line.

The initial value is **`nil`**, meaning that only the **`#\Space`** character is inserted.

A true value for **`editor:*indent-with-tabs*`** causes the indentation commands to insert **`#\Tab`** characters according to the value of **`spaces-for-tab`** and then pad with **`#\Space`** characters as needed.

6.3.11 Lisp

editor:*find-likely-function-ignores* *Variable*

Contains a list of symbols likely to be found at the beginning of a form (such as **`apply`**, **`funcall`**, **`defun`**, **`defmethod`**, **`defgeneric`**).

editor:*source-found-action* *Variable*

This variable determines how definitions found by the commands **`Find Source`**, **`Find Source for Dspec`** and **`Find Tag`** are shown. The value should be a list of length 2.

The first element controls the positioning of the definition: when **`t`**, show it at the top of the editor window; when a non-negative fixnum, position it that many lines from the top; and when **`nil`**, position it at the center of the window.

The second element can be `:highlight`, meaning highlight the definition, or `nil`, meaning do not highlight it.

The initial value of `*source-found-action*` is `(nil :highlight)`.

6.3.12 Movement

editor:line-end *Function*

`editor:line-end` ***point***

Moves *point* to be located immediately before the next newline character, or the end of the buffer if there are no following newline characters.

editor:line-start *Function*

`editor:line-start` ***point***

Moves *point* to be located immediately after the previous newline character, or the start of the buffer if there are no previous newline characters.

editor:character-offset *Function*

`editor:character-offset` ***point n***

Moves *point* forward *n* characters. If *n* is negative, *point* moves back *n* characters.

editor:word-offset *Function*

`editor:word-offset` ***point n***

Moves *point* forward *n* words. If *n* is negative, *point* moves back *n* words.

editor:line-offset *Function*

`editor:line-offset` ***point n*** `&optional` ***to-offset***

Moves *point n* lines forward, to a location *to-offset* characters into the line. If *n* is negative, *point* moves back *n* lines. If *to-offset* is `nil` (its default value), an attempt is made to retain the current offset. An error is signaled if there are not *n* further lines in the buffer.

editor:form-offset *Function*

`editor:form-offset` ***point n*** `&optional` ***form depth***

Moves *point* forward *n* Lisp forms. If *n* is negative, point moves back *n* forms. If *form* is `t` (its default value) then atoms are counted as forms, otherwise they are ignored. Before point is moved forward *n* forms, it first jumps out *depth* levels. The default value for *depth* is 0.

6.3.13 Prompting the user

The following functions can be used to prompt for some kind of input, which is generally typed into the Echo Area.

The following keyword arguments are common to a number of prompting functions.

`:must-exist`	Specifies whether the value that is input by the user must be an existing value or not. If `:must-exist` is non-nil, the user is prompted again if a non-existent value is input.
`:default`	Defines the default value that is selected if an empty string is input.
`:default-string`	Specifies the string that may be edited by the user (with `Insert Parse Default`).
`:prompt`	Defines the prompt that is written in the Echo Area. Most prompting functions have a default prompt that is used if no value is supplied for `:prompt`.
`:help`	Provides a help message that is printed if the user types "`?`".

editor:prompt-for-file *Function*

`editor:prompt-for-file &key` ***direction must-exist create-directories default default-string prompt help***

Prompts for a file name, and returns a pathname.

`:direction` You can specify *direction* `:input` (when expecting to read the file) or *direction* `:output` (when expecting to write the file). This controls the default value of *must-exist*, which is false for *direction* `:output` and true otherwise.

`:create-directories`
If *create-directories* is true, then the user is prompted to create any missing directories in the path she enters. The default is false for *direction* `:output` and true otherwise.

See above for an explanation of the other arguments.

editor:prompt-for-buffer *Function*

`editor:prompt-for-buffer &key` ***prompt must-exist default default-string help***

Prompts for a buffer name, and returns the buffer. See above for an explanation of the keywords.

The default value of *must-exist* is `t`. If *must-exist* is `nil` and the buffer does not exist, it is created.

editor:prompt-for-integer *Function*

`editor:prompt-for-integer &key` ***prompt must-exist default help***

Prompts for an integer. See above for an explanation of the keywords.

editor:prompt-for-string *Function*

`editor:prompt-for-string &key` ***prompt default default-string clear-echo-area help***

Prompts for a string. No checking is done on the input. The keyword *clear-echo-area* controls whether or not the echo area is cleared (that is, whether the text being replaced is visible or not). The default for this keyword is `t`. See above for an explanation of the remaining keywords.

editor:prompt-for-variable *Function*

`editor:prompt-for-variable &key` ***must-exist prompt default default-string help***

Prompts for an editor variable. See above for an explanation of the keywords. The default value of *must-exist* is `t`.

6.3.14 In-place completion

editor:complete-in-place *Function*

`editor:complete-in-place` ***complete-func*** **`&key`** ***extract-func skip-func insert-func***

Performs a non-focus completion at the editor current point.

complete-func should be a function designator with signature:

`complete-func` ***string*** **`&optional`** ***user-arg*** **`=>`** ***result***

string should be a string to complete. *user-arg* is the second return value of *extract-func*, if this is not `nil`. *result* should be a list of items to be displayed in the list panel of the non-focus window.

extract-func must be a function designator with signature

`extract-func` ***point*** **`=>`** ***string*,** ***user-arg***

point should be a `Point` object

extract-func needs to move *point* to the beginning of the text that will be replaced if the user confirms. It should return two values: *string* is the string to complete, and *user-arg* can be any Lisp object. *string* is passed to the function *complete-func*, and if *user-arg* is non-nil it is also passed.

The default value of *extract-func* is a function which searches backwards until it finds a non-alphanumeric character, or the beginning of the buffer. It then moves its *point* argument forward to the next character. The function returns its first value *string* the string between this and the original location of the point, and it returns **`nil`** as the second value *user-arg*.

skip-func, if supplied, must be a function designator with signature

`skip-func` ***point***

point should be a **`Point`** object

point will be used as the end of the region to replace by the completion. At the call to *skip-func*, the point is located at the same place as the point that was passed to *extract-func* (after it moved). *skip-func* needs to move *point* forward to the end of the text that should be replaced when the user wants to do the completion. If *skip-func* is not supplied, the end point is set to the current point.

insert-func, if supplied, must be a function designator with signature

`insert-func` ***result string user-arg*** **`=>`** ***string-to-use***

result is the item selected by the user, *string* is the original string that was returned by *extract-func*, and *user-arg* is the second value returned by *extract-func* (regardless of whether this value is **`nil`**). It must return a string, *string-to-use*, which is inserted as the the completion.

If *insert-func* is not supplied, the completion item is inserted. If it is not a string it is first converted by **`prin1-to-string`**.

When **`editor:complete-in-place`** is called, it makes a copy of the current point and passes it to *extract-func*. It then copies this point and positions it either using *skip-func* or the current point. These two points define the text to be replaced. **`editor:complete-in-place`** then calls *complete-func*, and use the result to raise a non-focus window next to the current point. The interaction of this window is as described in *CAPI User Guide and Reference Manual*.

Note: **`editor:complete-with-non-focus`** is a deprecated synonym for **`editor:complete-in-place`**.

6.3.15 Variables

editor:define-editor-variable *Function*

`editor:define-editor-variable` ***name value*** **`&optional`** ***documentation***

Defines an editor variable.

name	Symbol naming the variable.
value	The value to assign to the variable.
mode	A string naming a mode.
documentation	A documentation string.

The macro **`editor:define-editor-variable`** defines a global editor variable. There is only one global value, so repeated uses of **`editor:define-editor-variable`** overwrite each other.

`editor:define-editor-variable` gives a readable value of defining a variable, and is recognized by the LispWorks source code location system. However variables can also be defined dynamically by calling **`(setf editor:variable-value)`**. Variable values may be accessed by **`editor:variable-value`**.

A variable has only one string of documentation associated with it. **`editor:variable-value`** overwrites the existing documentation string, if there is any. You can see the documentation by the command **`Describe Editor Variable`**. It can can be accessed programmatically by **`editor:editor-variable-documentation`**.

Note: for backwards compatibility *name* can also be a string, which is converted to a symbol by uppercasing, replacing **`#\Space`** by **`#\-`**, and interning in the EDITOR package. This may lead to clashes and so you should use a symbol for *name*, not a string.

editor:define-editor-mode-variable *Function*

`editor:define-editor-mode-variable` ***name mode value*** **`&optional`** ***documentation***

Defines an editor variable in the specified mode.

mode	A string naming a mode.
name, value	As for **`editor:define-editor-variable`**.
documentation	As for **`editor:define-editor-variable`**, except that **`editor:define-editor-mode-variable`** installs the documentation only if the editor variable does not already have any documentation.

`editor:define-editor-mode-variable` defines a variable in the specified mode. There is one value per variable per mode.

`editor:define-editor-mode-variable` gives a readable value of defining a variable in a mode, and is recognized by the LispWorks source code location system. However mode variables can also be defined dynamically by calling **`(setf editor:variable-value)`**. Mode variable values may be accessed by **`editor:variable-value`**.

editor:editor-variable-documentation *Function*

`editor:editor-variable-documentation` ***editor-variable-name***

editor-variable-name

A symbol naming an editor variable.

Returns the documentation associated with the editor variable, if any.

Note: For backwards compatibility a string *editor-variable-name* is also accepted, as described for **`editor:define-editor-variable`**.

editor:variable-value *Accessor*

`editor:variable-value` ***name*** **`&optional`** ***kind where***

The reader returns the value of the editor variable *name*, where *name* is a symbol. An error is signaled if the variable is undefined. The argument *kind* can take the value **`:current`**, **`:buffer`**, **`:global`** or **`:mode`**. The default value of *kind* is **`:current`**.

When *kind* is **`:current`** the argument *where* should be **`nil`** (the default, meaning the current buffer) or an editor buffer object or the name of a buffer. The variable value for the specified buffer is returned or (if there is no current buffer) then the global variable value is returned.

kind can also be **`:buffer`**, and then *where* should be an editor buffer object.

For example, the code given below will, by default, return the value **`:ask-user`**.

```
(editor:variable-value
  'editor:add-newline-at-eof-on-writing-file)
```

The value of variables may also be altered using the setter of this function. For example, the code given below will allow buffers to be saved to file without any prompt for a missing newline.

```
(setf
   (editor:variable-value
      'editor:add-newline-at-eof-on-writing-file)
    nil)
```

editor:variable-value-if-bound *Function*

`editor:variable-value-if-bound` ***name*** **`&optional`** ***kind where***

Returns the value of the variable *name*. If the variable is not bound, **`nil`** is returned. The arguments are as for **`editor:variable-value`**.

editor:buffer-value *Function*

`editor:buffer-value` ***buffer name*** **`&optional`** ***errorp***

Accesses the value of the editor variable *name* in the buffer specified by *buffer.*

name should be a symbol and *buffer* should be a point object or a buffer object.

If the editor variable is undefined and *errorp* is true, an error is signaled. If the variable is undefined and *errorp* is false, **`nil`** is returned. The default value of *errorp* is **`nil`**.

6.3.16 Windows

editor:current-window *Function*

`editor:current-window`

Returns the current window.

editor:redisplay *Function*

`editor:redisplay`

Redisplays any window that appears to need it. In general, the contents of a window may not be redisplayed until there is an event to provoke it.

Note: **`editor:redisplay`** will update a modified editor buffer only when that buffer is the **`editor:current-buffer`**. Take care to call **`editor:redisplay`** in an appropriate context.

editor:window-text-pane *Function*

`editor:window-text-pane` ***window***

Returns the **`capi:editor-pane`** associated with an editor window.

6.3.17 Faces

editor:face *System Class*

An instance of the system class **`editor:face`** describes the "face" of some text. It specifies the colors of the text and background, the font, and whether the text is bold, italic or underlined.

A **`editor:face`** is created by calling **`editor:make-face`**. It is used by various interface functions, for example **`hcl:code-coverage-set-editor-colors`** and **`hcl:write-string-with-properties`**. Note that in general you can use a face name, that is associated with a **`editor:face`** by **`editor:make-face`**, instead of the actual **`editor:face`** object.

editor:make-face *Function*

`editor:make-face` ***name*** `&key` ***if-exists foreground background font bold-p italic-p underline-p inverse-p documentation*** `=>` ***face***

name	A symbol.
if-exists	`nil`, `:overwrite` or `:error`.
foreground, background	CAPI colors or `nil`.
font	A `graphics-ports:font` object or `nil`.
bold-p, italic-p, underline-p, inverse-p	Booleans.
documentation	A string or `nil`.
face	A `editor:face` object.

The function `editor:make-face` returns a `editor:face` , either new or existing, and may associate it with *name*. `editor:face` objects are used by some interface function such as `hcl:code-coverage-set-editor-colors` and `hcl:write-string-with-properties`.

If *name* is non-nil, `editor:make-face` first checks if a `editor:face` with this name already exists. If it exists, then *if-exists* controls what happens:

`nil`	Return the existing `editor:face` object as it is (the default).
`:overwrite`	Reset the existing `editor:face` to default values and set its slots using the supplied keywords. The existing face is returned. This also causes Editor windows to update, and where this face is used the display will change accordingly.
`:error`	Signal an error.

If there is no existing `editor:face`, either because *name* is `nil` or because it has not been made yet, `editor:make-face` creates a new `editor:face` from the supplied keywords. If *name* is non-nil, the `editor:face` is associated with *name*, so future calls to **`editor:make-face`** with the same *name* will find it and *name* can be used in interface functions.

None of the keywords is required, and they all default to **`nil`**. For *foreground*, *background* and *font*, **`nil`** means use the default value, that is the color or font that the text would have drawn if the *face* was not applied.

foreground and *background* specify the colors to use. When they are non-nil, they must be a CAPI color. See the chapter “The Color System" in the *CAPI User Guide and Reference Manual* for description of colors.

font specifies the font to use. It must be a **`graphics-ports:font`** object, typically the result of **`graphics-ports:find-best-font`**. See "Portable font descriptions" in the "Drawing - Graphics Ports" chapter in the *CAPI User Guide and Reference Manual* for details. Note that the editor does not work properly with fonts of different height.

bold-p, *italic-p* and *underline-p* specify whether the text should be bold, italic or underlined respectively.

inverse-p specifies that the foreground and background colors are swapped, which causes the text to be drawn in the current background color using the current foreground color as the background. The effective background color is either the *background* argument if it is non-nil, or the default otherwise, and the same for the effective foreground color.

documentation is stored in the **`editor:face`**, and can be retrieved by calling **`cl:documentation`** with **`editor:face`** as the *doc-type* argument. **`cl:documentation`** can be called either with a **`editor:face`** object or with *name*.

6.3.18 Examples

6.3.18.1 Example 1

The following simple example creates a new editor command called **`Current Line`**.

```
(editor:defcommand "Current Line" (p)
    "Computes the line number of the current point and
     prints it in the Echo Area"
    "Prints the line number of the current point"
    (let* ((cpoint (editor:current-point))
           (svpoint (editor:copy-point cpoint))
           (count 0))
          (editor:beginning-of-buffer-command nil)
          (loop
             (if (editor:point> cpoint svpoint)
                 (return))
             (unless (editor:next-line-command nil)
                 (return))
             (incf count))
          (editor:move-point cpoint svpoint)
          (editor:message "Current Line Number: ~S " count)))
```

6.3.18.2 Example 2

This example creates a new editor command called **`Goto Line`** which moves the current point to the specified line number.

```
(editor:defcommand "Goto Line" (p)
   "Moves the current point to a specified line number.
    The number can either be supplied via the prefix
    argument, or, if this is nil, it is prompted for."
   "Moves the current point to a specified line number."
   (let ((line-number
          (or p (editor:prompt-for-integer
                 :prompt "Line number: "
                 :help "Type in the number of the line to
                  go to"))))
        (editor:beginning-of-buffer-command nil)
        (editor:next-line-command line-number)))
```

6.3.18.3 Example 3

The following example illustrates how text might be copied between buffers. First, *string* is set to all the text in **`from-buf`**. This text is then copied to the end of **`to-buf`**.

```
(defun copy-string (from-buf to-buf)
   (let ((string (editor:points-to-string
           (editor:buffers-start from-buf)
           (editor:buffers-end from-buf))))
        (editor:insert-string (editor:buffers-end to-buf) string)))
```

To test this example, two buffers named `t1` and `t2` should be created. Then, to copy all the text from `t1` to the end of `t2`:

```
(copy-string (editor:buffer-from-name "t1")
             (editor:buffer-from-name "t2"))
```

6.4 Editor source code

The section does not apply to LispWorks Personal Edition.

LispWorks comes with source code for the editor, which you can refer to when adding editor extensions.

6.4.1 Contents

The directory `lib/7-1-0-0/src/editor/` contains most of the source files of the LispWorks editor. Some low-level source code is not distributed.

6.4.2 Source location

To enable location of editor definitions by `Find Source` and related commands, configure LispWorks as described under "Finding source code" in the *LispWorks User Guide and Reference Manual*.

6.4.3 Guidelines for use of the editor source code

Some care is needed when working with the supplied editor source code, to ensure that you do not compromise the IDE or introduce a dependency on a particular release of LispWorks.

In particular please note:

- The editor source code may not match the compiled code in the Lisp-Works image exactly, for example if editor patches have been loaded.
- Modifications to the EDITOR package definition are not allowed.

- Redefining existing definitions is not recommended. It is better to define a new command to do what you want. If you find a bug or have a useful extension to an existing definition then please let us know.
- Do not rely on the expansion of exported macros.
- If you use any internal (that is, not exported) EDITOR symbols, please tell us, so we can consider how to support your requirements. In addition, some internal macros have been removed from the LispWorks image and these should not be used.

7

Self-contained examples

This chapter enumerates the set of examples in the LispWorks library relevant to the content of this manual. Each example file contains complete, self-contained code and detailed comments, which include one or more entry points near the start of the file which you can run to start the program.

To run the example code:

1. Open the file in the Editor tool in the LispWorks IDE. Evaluating the call to **`example-edit-file`** shown below will achieve this.
2. Compile the example code, by **`Ctrl+Shift+B`**.
3. Run the example command, by **`Meta+X`** *command-name* or by the keystroke defined in an **`editor:bind-key`** form.
4. Read the comment at the top of the file, which may contain further instructions on how to interact with the example.

7.1 Example commands

```
(example-edit-file "editor/commands/spell-word")

(example-edit-file "editor/commands/space-show-arglist")

(example-edit-file "editor/commands/delete-deletes-selection")
```

```
(example-edit-file "editor/commands/split-line")
```

```
(example-edit-file "editor/commands/insert-date")
```

7.2 Syntax coloring example

These three files illustrate a way to implement Common Lisp syntax coloring in the editor.

```
(example-edit-file "editor/syntax-coloring/defsys")
```

```
(example-edit-file "editor/syntax-coloring/syntax-coloring")
```

```
(example-edit-file "editor/syntax-coloring/pkg")
```

Note: the editor now has built-in syntax coloring for Lisp mode buffers. If you run the example code above, it will override the built-in syntax coloring.

Glossary

Abbrev

An abbrev (abbreviation) is a user defined text string which, when typed into a buffer, may be expanded into another string using Abbrev Mode. Typing can therefore be saved by defining short strings to be expanded into frequently used longer words or phrases.

Abbrevs should not be confused with the abbreviated symbol completion implemented by the command **`Abbreviated Complete Symbol`**.

Abbrev Mode

Abbrev mode is a minor mode which allows abbrevs to be automatically expanded when typed into a buffer.

Attribute Line

A first line in a source file of the form

```
;; -*- Mode: Lisp; Package: CL-USER; -*-
```

is the attribute line. Its keys and values are processed by editor commands such as **`Process File Options`**.

Auto-Fill Mode

Auto-fill mode is a minor mode which allows lines to be broken between words at the right margin automatically as the text is being typed. This means that `Return` does not have to be pressed at the end of each line to simulate filling.

Auto-Saving

Auto-saving is the automatic, periodic backing-up of the file associated with the current buffer.

Backup

When a file is explicitly saved in the editor, a backup is automatically made by writing the old contents of the file to a backup before saving the new version of the file. The name of the backup file is that of the original file followed by a ~ character.

Binding

A binding is made up of one or more *key sequences*. A command may have a default binding associated with it, which executes that command. Bindings provide a quick and easy way to execute commands.

Buffer

A buffer is a temporary storage area used by the editor to hold the contents of a file while the process of editing is taking place.

Case Conversion

Case conversion means changing the case of text from lower to upper case and vice versa.

Completion

Completion is the process of expanding a partial or abbreviated name into the full name. Completion can used for expanding symbols, editor command names, filenames and editor buffer names.

Control Key

The Control key (`Ctrl`) is used as part of many key sequences. `Ctrl` must be held down while pressing the required character key.

Ctrl Key

See *Control Key*.

Current

The adjective *current* is often used to describe a point, buffer, mark, paragraph, and similar regions of text, as being the text area or item on which relevant commands have an effect. For example, the *current buffer* is the buffer on which most editor commands operate.

Cursor

The cursor is the rectangle (in Emacs emulation) or vertical bar (in other emulations) seen in a buffer which indicates the position of the current point within that buffer.

Customization

Customization means making changes to the way the editor works. The editor can be customized both in the short and long term to suit the users requirements. Short term customization involves altering the way the editor works for the duration of an editing session by using standard editor commands, while long term customization involves programming the editor.

Default

A default is the value given to an argument if none is specified by the user.

Deleting

Deleting means removing text from the buffer without saving it. The alternative is *killing*.

Echo Area

The Echo Area is a buffer used to display and input editor information. Commands are typed into this buffer and editor produced messages are displayed here.

Emulation

The LispWorks Editor can behave like GNU Emacs, or like a typical editor on the Mac OS X platform. Keys, cursors, behavior with selected text and other functionality differs. We use the term Mac OS X editor emulation to denote this alternate behavior.

Escape Key

The Escape key (`Esc`) has its own functionality but is mostly used in Emacs emulation in place of the `Meta` key when no such key exists on a keyboard. `Esc` must be typed *before* pressing the required character key.

Extended Command

Most editor commands can be invoked explicitly by using their full command names, preceded by the `Meta+X` key sequence. A command issued in such a way is known as an extended command.

Fill Prefix

The fill prefix is a string which is ignored when filling takes place. For example, if the fill prefix is `;;`, then these characters at the start of a line are skipped over when the text is re-formatted.

Filling

Filling involves re-formatting text so that each line extends as far to the right as possible without any words being broken or any text extending past a predefined right-hand column.

Global Abbrev

A global abbrev is an abbrev which can be expanded in all major modes.

History Ring

The history ring records Echo Area commands so that they can easily be repeated.

Incremental Search

An incremental search is a search which is started as soon as the first character of the search string is typed.

Indentation

Indentation is the blank space at the beginning of a line. Lisp, like many other programming languages, has conventions for the indentation of code to make it more readable. The editor is designed to facilitate such indentation.

Insertion

Insertion is the process of inputting text into a buffer.

Keyboard Macro

A keyboard macro allows a sequence of editor commands to be turned into a single operation. Keyboard macros are only available for the duration of an editing session.

Key Sequence

A key sequence is a sequence of characters used to issue, or partly issue, an editor command. A single key sequence usually involves holding down one of two specially defined modifier keys (that is **`Ctrl`** and **`Meta`**), while at the same time pressing another key.

Killing

Killing means removing text from a buffer and saving it in the kill ring, so that the text may be recovered at a later date. The alternative is *deleting*.

Kill Ring

The kill ring stores text which has been killed, so that it may be recovered at a later date. Text can be re-inserted into a buffer by *yanking*. There is only one kill ring for all buffers so that text can be copied from one buffer to another.

Location

A location is the position of a point which is saved automatically such that you can revisit it by commands such as `Go Back`.

Major Mode

Major modes govern how certain commands behave. They adapt a few editor commands so that their use is more appropriate to the text being edited. For example, the concept of indentation is radically different in Lisp mode and Fundamental mode. Each buffer is associated with one major mode.

Mark

A mark stores the position of a point in a buffer which is associated with the current region and may be used for reference at a later date. More than one mark may be associated with a single buffer and saved in a mark ring.

Mark Ring

The mark ring stores details of marks, so that previously defined marks can be accessed. The mark ring works like a stack, in that marks are pushed onto the ring and can only be popped off on a "last in first out" basis. Each buffer has its own mark ring.

Meta Key

On most PC keyboards this key is synonymous with the `Alt` key. However, there are many different types of keyboard, and the `Meta` key may not be marked with "Alt" or "Meta". It may be marked with a special character, such as a diamond, or it may be one of the function keys — try `F11`.

In Emacs emulation, `Meta` must be held down while pressing the required character key. As some keyboards do not have a `Meta` key, the *Escape* (`Esc`) key can be used in place of `Meta`.

On Cocoa, you can configure "Meta" by choosing **Preferences... > Environment > Emulation**.

Minor Mode

The minor modes determine whether or not certain actions take place. For example, when Abbrev mode is on, abbrevs are automatically expanded when typed into a buffer. Buffers may possess any number of minor modes.

Mode

Each buffer has one or more modes associated with it: a major mode and zero or more minor modes. Major modes govern how certain commands behave, while minor modes determine whether or not certain actions take place.

Mode Abbrev

A mode abbrev is an abbrev which is expanded only in predefined major modes.

Mode Line

At the bottom of each buffer is a mode line that provides information concerning that buffer. The information displayed includes name of the buffer, major mode, minor mode and whether the buffer has been modified or not.

Newline

Newline is a whitespace character which terminates a line of text.

Overwrite Mode

Overwrite mode is a minor mode which causes each character typed to replace an existing character in the text.

Page

A page is the region of text between two page delimiters. The ASCII key sequence `Ctrl+L` constitutes a page delimiter (as it starts a new page on most line printers).

Pane

A pane is the largest portion of an editor window, used to display the contents of a buffer.

Paragraph

A paragraph is defined as the text within two paragraph delimiters. A blank line constitutes a paragraph delimiter. The following characters at the beginning of a line are also paragraph delimiters: **`Space Tab @ - ' )`**

Prefix Argument

A prefix argument is an argument supplied to a command which sometimes alters the effect of that command, but in most cases indicates how many times that command is to be executed. This argument is known as a *prefix* argument as it is supplied before the command to which it is to be applied. Prefix arguments sometimes have no effect on a command.

Point

A point is a position in a buffer where editor commands take effect. The *current* point is generally between the character indicated by the cursor and the previous character (that is, it actually lies *between* two characters). Many types of commands (moving, inserting, deleting) operate with respect to the current point, and indeed move that point.

Recursive Editing

Recursive editing occurs when you are allowed to edit text while an editor command is executing.

Region

A region is the area of text between the mark and the current point. Many editor commands affect only a specified region.

Register

Registers are named slots in which locations and regions can be saved for later use.

Regular Expression Searching

A regular expression (regexp) allows the specification of a search string to include wild characters, repeated characters, ranges of characters, and alternatives. Strings which follow a specific pattern can be located, which makes regular expression searches very powerful.

Replacing

Replacing means substituting one string for another.

Saving

Saving means copying the contents of a buffer to a file.

Scrolling

Scrolling means slightly shifting the text displayed in a pane either upwards or downwards, so that a different portion of the buffer is displayed.

Searching

Searching means moving the current point to the next occurrence of a specified string.

Sentence

A sentence begins wherever a paragraph or previous sentence ends. The end of a sentence is defined as consisting of a sentence terminating character followed by two spaces or a newline. The following characters are sentence terminating characters: `. ? !`

Tag File

A tag file is one which contains information on the location of Lisp function definitions in one or more files. For each file in a defined system, the tag file contains a relevant file name entry, followed by names and positions of each defining form in that file. This information is produced by the editor and is required for some definition searches.

Transposition

Transposition involves taking two units of text and swapping them round so that each occupies the other's former position.

Undoing

Commands that modify text in a buffer can be undone, so that the text reverts to its state before the command was invoked.

Undo Ring

An undo ring is used to hold details of modifying commands so that they can be undone at a later date. The undo ring works like a stack, in that commands are pushed onto the ring and can only be popped off on a "last in first out" basis.

Variable (Editor)

Editor variables are parameters which affect the way that certain commands operate.

Whitespace

Whitespace is any consecutive run of the whitespace characters `Space`, `Tab` or Newline.

Window

A window is an object used by the window manager to display data. When the editor is called up, an editor window is created and displayed.

Window Ring

A window ring is used to hold details of all windows currently open.

Word

A word is a continuous string of alphanumeric characters (that is, the letters A–Z and numbers 0–9). In most modes, any character which is not alphanumeric is treated as a word delimiter.

Yanking

Yanking means inserting a previously killed item of text from the kill ring at a required location. This is often known as *pasting*.

Index