LispWorks® for the Windows® Operating System

Editor User Guide

Version 6.0
Contents

1 Introduction 1
   Using the editor within LispWorks 2

2 General Concepts 5
   Window layout 5
   Buffer locations 7
   Modes 8
   Text handling concepts 8
   Executing commands 9
   Basic editing commands 11

3 Command Reference 15
   Aborting commands and processes 16
   Executing commands 17
   Help 18
   Prefix arguments 23
   File handling 24
   Movement 36
   Marks and regions 43
   Locations 46
   Deleting and killing text 47
   Inserting text 52
   Delete Selection 55
   Undoing 55
Contents

Case conversion  56
Transposition  58
Overwriting  59
Indentation  60
Filling  63
Buffers  67
Windows  71
Pages  74
Searching and replacing  76
Comparison  88
Registers  89
Modes  90
Abbreviations  95
Keyboard macros  100
Echo area operations  102
Editor variables  106
Recursive editing  106
Key bindings  107
Running shell commands from the editor  109
Buffers, windows and the mouse  112
Miscellaneous  113

4 Editing Lisp Programs  115

Automatic entry into lisp mode  116
Syntax coloring  116
Functions and definitions  117
Forms  134
Lists  137
Comments  138
Parentheses  141
Documentation  143
Evaluation and compilation  145
Breakpoints  154
Removing definitions  155

5 Emulation  157

Using Microsoft Windows editor emulation  157
Key bindings 158
Replacing the current selection 159
Emulation in Applications 159

6 Advanced Features 161
   Customizing default key bindings 162
   Customizing Lisp indentation 164
   Programming the editor 164
   Editor source code 193

Glossary 195

Index 205
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 Microsoft Windows keys, an alternate keys and behaviour 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.
1 Introduction

- 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 don’t currently have an Editor (check the Windows menu), start one by choosing Tools > Editor or clicking on in the podium toolbar.

To produce a menu bar on each tool in LispWorks for Windows, choose Tools > Preferences... and adjust the Window Options configuration.

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.

1.1.1 About this manual

The 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 Microsoft Windows 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.
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 11). 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 Emacs binding, 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 Introduction
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
2 General Concepts

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 "SJIS" 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.
2.2 Buffer locations

2.2.1 Points

A point is a location in a buffer where editor commands take effect. The current point is generally the location 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 location 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.

- a major mode (such as Fundamental or Lisp). An 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.
2.2.3 Regions

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

2.3 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 in Lisp mode and in Fundamental mode. When a file is loaded into a new buffer, the default mode of that buffer is 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 — Ctrl and Alt

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.

The two modifier keys referred to are the Control (Ctrl) key and the Meta key which is usually Alt.

When using Emacs emulation on a keyboard without an Alt 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 Microsoft Windows editor emulation, the Alt key cannot be used as Meta, and Esc is the cancel gesture, so 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.
2 General Concepts

To continue the search, type Ctrl+M.

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 Alt key to be held down while pressing the character key. Alt+Ctrl+L, used to select the previous buffer displayed, is one such command. If the Esc or Ctrl+M key is being used in place of the Alt 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 Alt+X followed by the full command name (Alt+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

Editor commands can be supplied with an integer argument which sometimes alters the effect of that command. In most cases it means the at the command is repeated that many 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 sometimes have no effect on a command.

### 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 13). 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 16

Ctrl+G

Abort the current command which may either be running or just partially typed in. Use Esc in Microsoft Windows editor emulation.

#### 2.6.2 File handling

See “File handling” on page 24.
2 General Concepts

\texttt{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.

\texttt{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 52 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 \texttt{Return}.

2.6.4 Movement

See “Movement” on page 36.

\texttt{Ctrl+F}  Move the cursor forward one character.
\texttt{Ctrl+B}  Move the cursor backward one character.
\texttt{Ctrl+N}  Move the cursor down one line.
\texttt{Ctrl+P}  Move the cursor up one line.

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

\texttt{Ctrl+A}  Move the cursor to the beginning of the line.
\texttt{Ctrl+E}  Move the cursor to the end of the line.
\texttt{Ctrl+V}  Scroll one screen forward.
\texttt{Alt+V}  Scroll one screen backward.
\texttt{Alt+Shift+<}  Move to the beginning of the buffer.
\texttt{Alt+Shift+>}  Move to the end of the buffer.
2.6 Basic editing commands

2.6.5 Deleting and killing text
See “Deleting and killing text” on page 47.

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 55.

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 Alt+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 43.
2 General Concepts

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 47.

Alt+W    Copy the region between the mark and the current point. See “Deleting and killing text” on page 47.

Ctrl+Y   Insert (yank) a copied region before the current point. See “Inserting text” on page 52.

2.6.8 Help

See “Help” on page 18.

Ctrl+H A string  List all commands whose name contains string.

Ctrl+H D command  Describe command, where command is the full command name.

Ctrl+H K key    Describe the command bound to key.
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”
- “Prefix arguments”
- “File handling”
- “Movement”
- “Marks and regions”
- “Deleting and killing text”
- “Inserting text”
- “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 from the editor”
• “Buffers, windows and the mouse”
• “Miscellaneous”

3.1 Aborting commands and processes

Key Sequence

Ctrl+G

Aborts the current command. Ctrl+G (or Esc in Microsoft Windows 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**

**Ctrl+Break**

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

1. If the break gesture is sent to any CAPI interface that is waiting for events, it does "Interface break", as described below.

2. Otherwise 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.

3. Otherwise, if the LispWorks IDE is running, activate or start the Process Browser. Note that the Process Browser tool, documented in the *LispWorks IDE User Guide* can be used to break any other process.

4. Otherwise, if there is a busy process break it.

5. Otherwise, just break the current process.

"Interface break" depends on the interface. For an interface that has another process, notably the Listener with its REPL, it breaks that other process. For most interfaces, in the LispWorks IDE it starts the Process Browser, otherwise just it breaks the interface’s process.

### 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 from the editor” on page 109.
Extended Command

Key sequence: Alt+X

Allows the user to type in a command 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 143).

3.3.1 The help command

Help

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.
3.3 Help

q or n  Quit help.
a string  Display a list of commands whose names contain string.
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.

\texttt{Ctrl+D command}  Bring up the on-line version of this manual for command.

\texttt{g object}  Invoke the appropriate describe object command.

\texttt{k key}  Describe the command to which key is bound.

\texttt{Ctrl+K key}  Bring up the on-line version of this manual for key.

\texttt{l}  describe the last 60 keys typed.

\texttt{v variable}  Describe variable and show its current value.

\texttt{Ctrl+V variable}  Bring up the on-line version of this manual for variable.

\texttt{w command}  Display the key sequence to which command is bound.

\textbf{Apropos Command} \hspace{1cm} Editor Command

Arguments: \textit{string}

Key sequence: \texttt{Ctrl+H A string}

Displays a list of editor commands, variables, and attributes whose names contain \textit{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

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

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

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.
3.3 Help

**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`. 

21
What Lossage  
*Editor Command*

Arguments: None  
Key sequence: \texttt{Ctrl+H L}  
Displays the last 60 keys typed.

Describe Editor Variable  
*Editor Command*

Arguments: \textit{variable}  
Key sequence: \texttt{Ctrl+H V \textit{variable}}  
Describes \textit{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: \textit{variable}  
Key sequence: \texttt{Ctrl+H Ctrl+V \textit{variable}}  
Brings up the on-line version of this manual at the entry for \textit{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: \textit{command}  
Key sequence: \texttt{Ctrl+H W \textit{command}}  
Displays the key sequence to which \textit{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.  
\textbf{Note:} this command is also available via the menu command Help > Editing > Command to Key.
3.4 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** followed by an integer. Negative prefix arguments are allowed. A prefix argument between 0 and 9 can also be supplied using Alt+digit.

---

**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.

---

**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 \texttt{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 \texttt{Set Prefix Argument}.

**None**

*Key Sequence*

Key sequence: \texttt{Alt+<0-9>}

Provides a prefix argument in a similar fashion to \texttt{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 \texttt{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 34. 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 File handling

input-format-default

Editor Variable

Default value: \texttt{nil}

The default external format used by \texttt{Find File}, \texttt{Wfind File} and \texttt{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 \texttt{Set External Format} has been used to specify an external format) then \texttt{input-format-default} is ignored. If the value is \texttt{nil} and the buffer does not have an external format, \texttt{cl:open} chooses the external format to use.

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

If you have specified an input encoding via the Editor tool’s Preferences dialog, then \texttt{input-format-default} is initialized to that value on startup.

3.5.2 Saving files

Save File

Editor Command

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

\texttt{editor:save-file-command \texttt{p}} &optional \texttt{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 \texttt{output-format-default}, or as defaulted by \texttt{open} if this is \texttt{nil}.

The argument \texttt{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 Save Selected Buffers 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 Save Selected Buffers 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

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

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

Arguments: None

Key sequence: Ctrl+X Ctrl+C

A Save Selected Buffers 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

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

**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 *LispWorks 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`.

**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.

3.5 File handling

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.

3.5.3 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
3.5 File handling

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

\textbf{auto-save-cleanup-checkpoints} \quad \textit{Editor Variable}

Default value: \texttt{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.

3.5.4 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 \texttt{-} character.

\textbf{backups-wanted} \quad \textit{Editor Variable}

Default value: \texttt{t}

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

\textbf{backup-filename-suffix} \quad \textit{Editor Variable}

Default value: \texttt{#\~}

This variable contains the character used as a suffix for backup files. By default, this is the tilde (\texttt{-}) character.

\textbf{backup-filename-pattern} \quad \textit{Editor Variable}

Default value: "\texttt{~A~A~A}"

This control string is used with the Common Lisp \texttt{format} function to create the filename of the backup file. \texttt{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:

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

3.5.5 Miscellaneous file operations

Print File

Arguments: file
Key sequence: None

Prints file, using capi:print-file. See the LispWorks CAPI Reference Man-
ual for details of this function.

Revert Buffer

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 recov-
ered 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.
3.5 File handling

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

**revert-buffer-confirm**

*Editor Variable*

Default value: \texttt{t}

When the command Revert Buffer is invoked, if the value of this variable is \texttt{t} and the buffer has been modified then confirmation is requested before the revert operation is performed. If its value is \texttt{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 90.

**Insert File**

*Editor Command*

Arguments: pathname
Key sequence: Ctrl+X I pathname

\texttt{editor:insert-file-command p &optional pathname buffer}

Inserts the file defined by \texttt{pathname} into the current buffer at the current point.

The argument \texttt{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 &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.

### 3.6 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
3.6 Movement

prefix argument repeats these commands a certain number of times in the opposite logical direction. For example, the command \texttt{Ctrl+U 10 Ctrl+B} moves the cursor 10 characters backwards, but the command \texttt{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.

**Forward Character** \textit{(Editor Command)}

- **Arguments:** None
- **Key sequence:** \texttt{Ctrl+F} or Right Arrow on some keyboards
- Moves the current point forward one character.

**Backward Character** \textit{(Editor Command)}

- **Arguments:** None
- **Key sequence:** \texttt{Ctrl+B} or Left Arrow on some keyboards
- Moves the current point backward one character.

**Forward Word** \textit{(Editor Command)}

- **Arguments:** None
- **Key sequence:** \texttt{Alt+F}
- Moves the current point forward one word.

**Backward Word** \textit{(Editor Command)}

- **Arguments:** None
- **Key sequence:** \texttt{Alt+B}
- Moves the current point backward one word.

**Beginning of Line** \textit{(Editor Command)}

- **Arguments:** None
- **Key sequence:** \texttt{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 Arrow on some keyboards

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 Arrow on some keyboards

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*.

**What Line**

*Editor Command*

Arguments: None.

Key sequence: None

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

**Forward Sentence**  
*Editor Command*

Arguments: None  
Key sequence: `Alt+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: `Alt+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: `Alt+]`

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: `Alt+['

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 \ \textit{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 \textit{window} is the name of the window to be scrolled. The default is the current window.

\textbf{Scroll Window Up} \hspace{1cm} \textit{Editor Command}

Arguments: None
Key sequence: Alt+V

editor:scroll-window-up-command \ p \ \&optional \ \textit{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 \textit{window} is the name of the window to be scrolled. The default is the current window.

\textbf{scroll-overlap} \hspace{1cm} \textit{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.
3.6 Movement

**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: `Alt+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: `Alt+Shift+<`

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

Editor Command

Arguments: None
Key sequence: Alt+Shift+>

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

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).

What Cursor Position

Editor Command

Arguments: None
Key sequence: Ctrl+X =

Displays in the echo area the character under the point and the column of the point. Also available via the function:


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.

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.
3.7 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 24 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 112—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.

3.7.1 Marks

Set Mark  

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 makes the 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: Alt+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  

def: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: Alt+@  

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

**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: \texttt{Alt+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: \texttt{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.

3.7.2 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 *LispWorks CAPI Reference Manual* for details of this function.

### 3.8 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.
### 3.9 Deleting and killing text

#### 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.

### 3.9 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 3-55 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 Alt+D deletes 10 words after the current point, but the key sequence Ctrl+U -10 Alt+D deletes 10 words before the current point.

### 3.9.1 Deleting Text

#### Delete Next Character

*Editor Command*

Arguments: None

Key sequence: \texttt{Ctrl+D}

Deletes the character immediately after the current point.

#### Delete Previous Character

*Editor Command*

Arguments: None

Key sequence: Delete

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.
3.9 Deleting and killing text

**Delete Horizontal Space**  
*Editor Command*

Arguments: None  
Key sequence: \texttt{Alt+\backslash}

Deletes all spaces on the line surrounding the current point.

**Just One Space**  
*Editor Command*

Arguments: None  
Key sequence: \texttt{Alt+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: \texttt{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 \texttt{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.

### 3.9.2 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 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: Alt+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.
3.9 Deleting and killing text

**Kill Previous Word**

*Editor Command*

Arguments: None
Key sequence: **Alt+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: **Alt+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: Alt+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: Alt+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.

### 3.10 Inserting text

This section contains details of commands used to insert text from the kill ring—see “Deleting and killing text” on page 47—and various other commands used to insert text and lines into the buffer.

**Un-Kill**  
*Editor Command*  
Arguments: None  
Key sequence: Ctrl+Y
3.10 Inserting text

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.

Rotate Kill Ring

*Editor Command*

Arguments: None

Key sequence: Alt+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 Alt+Y the required number of times. If Un-Kill was not the previous command, an error is signalled.

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.
3 Command Reference

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 `1.

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 #.

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.
3.11 Delete Selection

**Dynamic Completion**  
*Editor Command*

- **Arguments:** None
- **Key sequence:** `Alt+/

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.

**Expand File Name**  
*Editor Command*

- **Arguments:** None
- **Key sequence:** `Alt+Tab`

Expands the file name at the current point. Issuing this command twice in succession brings up a list of possible completions in a popup window.

### 3.11 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 Microsoft Windows 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.

### 3.12 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 typed repeatedly, the most recent commands in the editing session are successively undone.

**undo-ring-size**

*Editor Variable*

Default value: 100

The number of items in the undo ring.

### 3.13 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: `Alt+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: Alt+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: Alt+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.
3 Command Reference

**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.

**3.14 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: **Alt+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 argu-
3.15 Overwriting

Overwriting 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.

### 3.15 Overwriting

In the default mode of operation, each character that is typed is inserted into the text, with the existing characters being shifted as appropriate. In overwrite mode, each character that is typed 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.

### 3.16 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: Alt+Ctrl+\n
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: `Alt+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.
3.17 Filling

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

**Back to Indentation**

*Editor Command*

Arguments: None  
Key sequence: Alt+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.

3.17 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.

### 3.17.1 Fill commands

**Fill Paragraph**

*Editor Command*

**Arguments:** None

**Key sequence:** `Alt+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:** `Alt+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 \texttt{fill-column} to be set at the required value.

\textbf{fill-prefix} \hfill \textit{Editor Variable}

Default value: \texttt{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 \texttt{nil}, no characters are excluded when text is filled.

If the value 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.

\textbf{Set Fill Prefix} \hfill \textit{Editor Command}

Arguments: None
Key sequence: \texttt{Ctrl+X .}

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

\textbf{Center Line} \hfill \textit{Editor Command}

Arguments: None
Key sequence: None

Centers the current line with reference to the current value of \texttt{fill-column}.

A prefix argument causes the current line to be centered with reference to the required width.
3.17.2 Auto-fill mode

In the default mode of operation, 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.
3.18 Buffers

**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.

### 3.18 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: `Alt+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: `Alt+Ctrl+Shift+L`

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

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 is given on the name of the buffer, its 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: *lw:simple-char*

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

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

Arguments: new-name
Key sequence: None

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

Print Buffer

Arguments: None
Key sequence: None

Prints the current buffer, using capi:print-text. See the LispWorks CAPI Reference Manual for details of this function.

Toggle Buffer Read-Only

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.

Check Buffer Modified

Arguments: None
Key sequence: Ctrl+X Shift+-

Checks whether the current buffer is modified or not.
3.19 Windows

Buffer Not Modified  

Editor Command

Arguments: None

Key sequence: Alt+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.

3.19 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: Ctrl+X 1  
Deletes the next window in the window ring.

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.

**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.
3.20 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 \texttt{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 \texttt{Quoted Insert} (that is, type in \texttt{Ctrl+Q Ctrl+L}).

**Previous Page**

*Editor Command*

Arguments: None  
Key sequence: \texttt{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: \texttt{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.

**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.

### 3.21 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.

#### 3.21.1 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 signalled.

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

<table>
<thead>
<tr>
<th>Key Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Ctrl+S</code></td>
<td>If the search string is empty, repeats the last incremental search, otherwise repeats a forward search for the current search string.</td>
</tr>
</tbody>
</table>


3.21 Searching and replacing

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.

**Alt+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.

**Alt+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.

**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 signalled.

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

\textbf{Ctrl+R} \quad \text{If the search string is empty, repeats the last incremental search, otherwise repeats a backward search for the current search string.}
\text{If the search string cannot be found, starts the search from the end of the buffer (wrap-around search).}

\textbf{Ctrl+S} \quad \text{Changes to a forward search.}

\textbf{Delete} \quad \text{Cancels the last character typed.}

\textbf{Esc} \quad \text{If the search string is empty, invokes a non-incremental search, otherwise exits the search, leaving the current point at the last find.}

\textbf{Ctrl+G} \quad \text{Aborts the search, returning the current point to its original location.}
\text{If the search string cannot be found, cancels the last character typed (equivalent to Delete).}

\textbf{Ctrl+Q} \quad \text{Quotes the next character typed.}

**Forward Search** \quad \textit{Editor Command}

Arguments: \textit{string}
Key sequence: \texttt{Ctrl+S Esc string}

\texttt{editor:forward-search-command p \&optional string the-point}

The default for \textit{the-point} is the current point.
3.21 Searching and replacing

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.

**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.

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 one, 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.

Directory Search

Editor Command

Arguments: \( directory \) \( string \)
Key sequence: None

Searches files in \( directory \) for \( string \). The current working directory is offered as a default for \( directory \).

By default only files with suffix \( .lisp \), \( .lsp \), \( .c \) 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 \( \text{system:*ignorable-file-suffices*} \).

Use the key sequence \( \text{Alt+} \), to find subsequent definitions of the search string.
3.21 Searching and replacing

**Search Files**

*Editor Command*

Arguments: `search-string directory`

Key sequence: `Ctrl+X *`

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 &`

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 `Alt+`, to find subsequent definitions of the search string.
3 Command Reference

**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 programatically using the `search-files` function:

**search-files**

*Function*

Summary

Search all the files in a list for a string.

Package

`editor`

Signature

`search-files &key string files generator => nil`

Arguments

- `string` A string to search for (prompted if not given)
- `files` A list of pathnames of files to search
- `generator` A function to generate the files if none are given

Values

`search-files` returns `nil`. 
3.21 Searching and replacing

Description

**search-files** searches all the files in the list for a given string. If a match is found the file is loaded into a buffer with the cursor on the occurrence. **Alt+-**, makes the search continue until the next occurrence.

Example

```
CL-USER 91 > (editor:search-files
   :files '(".login" ".cshrc")
   :string "alias")
```

3.21.2 Regular expression searching

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, \texttt{ca？r} matches either the string \texttt{cr} or \texttt{car}, and nothing else. An empty regexp followed by \texttt{?} matches an empty part of the input.

^ Matches the next regexp as long as it is at the beginning of a line. For example, ^\texttt{foo} matches the string \texttt{foo} as long as it is at the beginning of a line.

\texttt{$} Matches the previous regexp as long as it is at the end of a line. For example, \texttt{foo\$} matches the string \texttt{foo} as long as it is at the end of a line.

\texttt{[ ]} 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, \texttt{[a-b]} matches either \texttt{a} or \texttt{b} or the empty string. The regexp \texttt{c[a-d]*r} matches strings beginning with \texttt{c} and ending with \texttt{r}, with any number of \texttt{a}'s and \texttt{d}'s in-between.

The characters - and ^ have special meanings inside character sets. - defines a range and ^ defines a complement character set. For example, \texttt{[a-d]} matches any character in the range \texttt{a} to \texttt{d} inclusive. \texttt{[^ab]} matches any character except \texttt{a} or \texttt{b}.

\texttt{\\} Quotes the special characters. For example, \texttt{\*} matches the character \texttt{*} (that is, \texttt{*} has lost its special meaning).

\texttt{\|} Specifies an alternative. For example, \texttt{ab|cd} matches either \texttt{ab} or \texttt{cd}.

\texttt{\(, \)} Provides a grouping construct. For example, \texttt{ab\(cd|ef\)} matches either \texttt{abcd} or \texttt{abef}.
3.21 Searching and replacing

**Regexp Forward Search**

*Editor Command*

Arguments: *string*

Key sequence: `Alt+Ctrl+S string`  

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

Performs a forward 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.

The argument *the-point* specifies the position from which the search is to start. The default is the current point. *limit* specifies a limiting point in the buffer for the search. The default is the end of the buffer.

**Regexp Reverse Search**

*Editor Command*

Arguments: *string*

Key sequence: `Alt+Ctrl+R string`  

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

Performs a 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.

The argument *the-point* specifies the position from which the search is to start. The default is one position before the current point. *limit* specifies a limiting point in the buffer for the search. The default is the current point.

**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*. 
3.21.3 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: `Alt+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.
3.21 Searching and replacing

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 file with the suffix .lisp or .lsp in directory, but only after querying the user. The current working directory is offered as a default for directory. A non-nil prefix argument causes all files to be searched, except for those ending with one of the strings in the list system:*ignore-file-suffixes*. 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.

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

Arguments: target replacement
Key sequence: None

```
(editor:replace-regexp-command p) &optional target replacement
```

Replaces all matches of target regular expression by replacement string, starting from the current point.

See “Regular expression searching” for a description of regular expressions.

Query Replace Regexp

 Arguments: target replacement
 Key sequence: None

```
(editor:query-replace-regexp-command p) &optional target replacement
```

Replaces matches of target regular expression by replacement string, starting from the current point, but only after querying the user. Each time target is matched, an action must be indicated from the keyboard.

See “Regular expression searching” for a description of regular expressions, and Query Replace for the keyboard gestures available.

3.22 Comparison

This section describes commands which compare files and/or buffers against each other.

Diff

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.
### 3.23 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.

#### 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`.

#### 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 \textit{name}.

\textbf{List Registers} \hspace{1cm} \textit{Editor Command}

- Arguments: None
- Key sequence: None
- Lists all existing registers.

\textbf{Copy to Register} \hspace{1cm} \textit{Editor Command}

- Arguments: \textit{name}
- Key sequence: \texttt{Ctrl+X X name}
- Saves the region between the mark and the current point to the register called \textit{name}.
  
  \texttt{Put Register} is a synonym for \texttt{Copy to Register}.

\textbf{Insert Register} \hspace{1cm} \textit{Editor Command}

- Arguments: \textit{name}
- Key sequence: \texttt{Ctrl+X G name}
- Copies the region from the register called \textit{name} to the current point.
  
  \texttt{Get Register} is a synonym for \texttt{Insert Register}.

\section*{3.24 Modes}

A buffer can be in two kinds of mode at once: \textit{major} and \textit{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.

\subsection*{3.24.1 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 3-60.

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`.

- **Shell mode.** Used for running interactive shells.

The major mode of most buffers may be altered explicitly by using the com-
mands 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:

```lisp
(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.

### 3.24.2 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 59.
- **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 63.
- **Abbrev mode.** Allows abbreviation definitions to be expanded automatically—see “Abbreviations” on page 95.
- **Execute mode.** Used by the Listener to make history commands available (see the *LispWorks IDE User Guide*).

### 3.24.3 Default modes

**default-modes**

*Editor Variable*

Default value: ("Fundamental")

This editor variable contains the default list of modes for new buffers.
3.24.4 Defining modes

New modes can be defined using the `defmode` macro.

**defmode**

*Function*

**Summary**
Defines new editor modes.

**Package**
editor

**Signature**

```
defmode name &key setup-function syntax-table key-bindings no-redefine vars cleanup-function major-p transparent-p precedence => nil
```

**Arguments**

- `name` A string containing the name of the mode being defined.
- `setup-function` Name of function which sets up a buffer in this mode.
- `key-bindings` A quoted list of key-binding directions.
- `no-redefine` If `t`, the mode cannot be re-defined. The default value is `nil`.
- `vars` A quoted list of Editor variables and values.
- `aliases` A quoted list of synonyms for `name`.
- `cleanup-function` Called upon exit from a buffer in this mode.
- `major-p` If `t`, the mode is defined as major, otherwise minor. The default value is `nil`.

**Values**

`defmode` returns `nil`.

**Description**

This function defines an Editor mode called `name`. By default, any mode defined is a minor one—specification of major-mode status is made by supplying `t` to the `major-p` argument.

`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.

**Example**

Let 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:
3.25 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

An Abbrev command arg

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

Arguments: abbrev

Key sequence: Ctrl+X Ctrl+A abbrev

Now, any files loaded into the Editor with the suffix .foo invoke the Foo minor mode.
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: \texttt{Ctrl}+\texttt{X} \texttt{Ctrl}+\texttt{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: \texttt{Ctrl}+\texttt{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: \texttt{Ctrl}+\texttt{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
3.25 Abbreviations

```
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: `Alt+’`

Allows the prefix before the current point to be attached to the following abbreviation. For example, if the abbreviation `valn` is bound to `valuation`, typing `re` followed by `Alt+’`, 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 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.

3.26 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.
3.26 Keyboard macros

**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.

### 3.27 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.

#### 3.27.1 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`

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.
3.27 Echo area operations

Complete Field Editor Command

Arguments: None
Key sequence: Space
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
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 signalled.

Help on Parse Editor Command

Arguments: None
Key sequences: ?, Help, F1
Displays a popup list of all possible completions of the text in the echo area.

3.27.2 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: Alt+P
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: Alt+N

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.

### 3.27.3 Movement in the echo area

**Echo Area Backward Character**

Editor Command

Arguments: None  
Key sequence: Ctrl+B

Moves the cursor back one position (without moving into the prompt).

**Echo Area Backward Word**

Editor Command

Arguments: None  
Key sequence: Alt+B

Moves the cursor back one word (without moving into the prompt).

**Beginning Of Parse**

Editor Command

Arguments: None  
Key sequence: Alt+<

Moves the cursor to the location immediately after the prompt in the Echo Area.
3.27 Echo area operations

**Beginning Of Parse or Line**  
*Editor Command*

Arguments: None  
Key sequence: `Ctrl+A`

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.

### 3.27.4 Deleting and inserting text in the echo area

**Echo Area Delete Previous Character**  
*Editor Command*

Arguments: None  
Key sequence: `Delete`

Deletes the previous character entered in the Echo Area.

**Echo Area Kill Previous Word**  
*Editor Command*

Arguments: None  
Key sequence: `Alt+Delete`

Kills the previous word entered in the Echo Area.

**Kill Parse**  
*Editor Command*

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

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`

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  
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  
Inserts the editor window’s selected text in the echo area.

### 3.28 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.

### 3.29 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: Alt+Ctrl+Z  
Exits a level of recursive edit, returning to the original command. An error 
is signalled 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 immedi-
ately. An error is signalled if not in a recursive edit.

### 3.30 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 162.

**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.
3.31 Running shell commands from the editor

**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.

**3.31 Running shell commands from the editor**

The editor allows both single MS DOS commands to be executed and also provides a means of running a shell interactively.

**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.

**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, and the minor mode is Execute mode so the history key bindings available in the Listener 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 recognises 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 C:\temp` is issued, the next time the editor command `wfind File` is issued, the default directory offered is `C:\temp`. If you find that the editor has not recognised a change directory command then the editor command `cd` may be used to change the default directory of the buffer.

**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-poptd-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.
3.31 Running shell commands from the editor

**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.
3.32 Buffers, windows and the mouse

3.32.1 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).

3.32.2 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 43—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.
3.33 Miscellaneous

**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.

3.33 Miscellaneous

**Report Bug**

*Editor Command*

Argument: None
Key sequence: None

Opens a window containing the template for reporting bugs in LispWorks. This template can then be filled in and emailed to Lisp Support.

**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.

**Report Manual Bug**  
*Editor Command*

Argument: None  
Key sequence: None

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.

**Room**  
*Editor Command*

Argument: None  
Key sequence: None

Displays information on the current status of the memory allocation for the host computer.

**Build Application**  
*Editor Command*

Argument: None  
Key sequence: None

Invokes the Application Builder in the LispWorks IDE and does a build. By default, it uses the current buffer as the build script. If given a prefix argument it prompts for a file to use as the build script.
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 90.

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:

```lisp
(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".
4.3 Functions and definitions

4.3.1 Movement, marking and specifying indentation

**Beginning of Defun**  
*Editor Command*

- **Argument:** None
- **Key sequence:** Alt+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:** Alt+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:** Alt+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 \texttt{defun}.

\texttt{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 for a function (or a macro, variable, editor command, and so on), and finding the actual definition of that function. This is particularly useful in large systems, where code may exist in a large number of source files.

Function 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 \texttt{dspec:*active-finders*} appropriately. See the \textit{LispWorks User Guide and Reference Manual} for details.

Source records are created if the variable \texttt{*record-source-files*} is true when definitions are compiled, evaluated or loaded. See the \textit{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 \texttt{Create Tags Buffer}. GNU Emacs tag files are fully compatible with LispWorks editor tag files.

\textbf{Find Source} \begin{flushright} \textit{Editor Command} \end{flushright} 

\begin{itemize}
  \item \textbf{Argument:} name
  \item \textbf{Key sequence:} Alt+. \texttt{name}
\end{itemize}
4.3 Functions and definitions

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 Alt+, (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 116.

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 Alt+), 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.

**Edit Editor Command**

*Editor Command*

Argument: `command`
Key sequence: None

This is a synonym for **Find Command Definition**.

**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.
4.3 Functions and definitions

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 \texttt{Alt+}, action.

See the chapter on the Editor tool in the \textit{LispWorks IDE User Guide} for more information about the Find Definitions view.

**List Definitions For Dspec**

\textit{Editor Command}

Argument: \textit{dspec}  
Key sequence: None

This command is similar to \texttt{List Definitions}, but takes a definition spec \textit{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 \texttt{Alt+}, action.

See the chapter on the Editor tool in the \textit{LispWorks IDE User Guide} for more information about the Find Definitions view.

**Create Tags Buffer**

\textit{Editor Command}

Argument: None  
Key sequence: None

Creates a buffer containing Tag search information, for all the .\texttt{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 \texttt{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**

\textit{Editor Command}

Key sequence: \texttt{Alt+?}
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 `Alt+`, (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 `Alt+`, (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: `Alt+`,

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: Alt+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.
4.3 Functions and definitions

**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.
Break Function  
Editor Command

Argument: function  
Key sequence: None

This command is like Trace Function but the trace is with \texttt{: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 \texttt{: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 \texttt{:break t}.

Break Definition on Exit  
Editor Command

Argument: None  
Key sequence: None

Like Trace Definition but the definition is traced with \texttt{: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 \texttt{toggle-source-debugging} in the LispWorks User Guide and Reference Manual).
4.3 Functions and definitions

**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.


**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.


**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.

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.


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 Functions and definitions

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: predicate
Key sequence: Alt+Ctrl+I

Attempts to complete the symbol before the current point. 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 predicate is non-nil then only symbols which are bound or fbound are offered amongst the possible completions.

**Abbreviated Complete Symbol**  
**Editor Command**

Argument: predicate  
Key sequence: Alt+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 predicate is non-nil 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 that 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 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: `Alt+= 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.

Note: example code showing how to use this command to display argument lists automatically is supplied with LispWorks, in the file `examples/editor/commands/space-show-arglist.lisp`

**Function Arglist Display**

*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.
Additionally, while the displayer is visible:

- **Ctrl+/** Controls whether the documentation string of the operator is also shown.
- **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.

### Function Argument List

**Editor Command**

- **Argument:** `function`
- **Key sequence:** **Ctrl+Shift+A function**

This command is similar to **Function Arglist**, except that 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-functionignores*` 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.

### 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.
4.3 Functions and definitions

**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: `Alt+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: `Alt+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: `Alt+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: `Alt+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:** `Alt+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:** `Alt+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: Alt+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: Alt+Ctrl+T
4.5 Lists

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: \texttt{Alt+"}

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: \texttt{Alt+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: \texttt{Alt+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: Alt+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: Alt+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

**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: Alt+;  

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: Alt+#  

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: \texttt{Alt+P}

Moves to the previous line and then performs an \texttt{Indent for Comment}.

Down Comment Line

*Editor Command*

Argument: None
Key sequence: \texttt{Alt+N}

Moves to the next line and then performs an \texttt{Indent for Comment}.

Indent New Comment Line

*Editor Command*

Argument: None
Key sequence: \texttt{Alt+J} or \texttt{Alt+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 \texttt{Indent New Comment Line} is performed when the current point is not in a comment line, it simply acts as a \texttt{Return}.

Kill Comment

*Editor Command*

Argument: None
Key sequence: \texttt{Alt+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 \texttt{comment-start}. 

---

4 Editing Lisp Programs
4.7 Parentheses

**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:** \( \text{Alt}+\)`

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:** \( \text{Alt}+\)`

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.

**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:** None

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.


**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
4 Editing Lisp Programs

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`

```lisp
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: `Alt+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

4.9 Evaluation and compilation

The commands described below allow the user to evaluate (interpret) or com-
pile 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

Default value: nil

If non-nil, defines the value of the current package.

Set Buffer Package

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

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: Alt+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, whilst defvar does not reassign the value of a bound variable, 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.
4.9 Evaluation and compilation

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: Alt+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.
4 Editing Lisp Programs

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 130 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.
4.9 Evaluation and compilation

**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, whilst \texttt{defvar} does not reassign the value of a bound variable, when editing a program it is likely that you do want the new value.

**Compile Region**  
*Editor Command*

- Argument: None
- Key sequence: \texttt{Ctrl+Shift+R}
- Compiles the Lisp forms in the region between the current point and the mark.

**Compile File**  
*Editor Command*

- Argument: \texttt{file}
- Key sequence: None
- Compiles \texttt{file} unconditionally, with \texttt{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: \texttt{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 \texttt{Compile File}, but checks the file first. If its associated binary (fasl) file is older than the source file or does not exist then the file is compiled. If the binary file is up to date, the user is asked whether the file should be compiled anyway.
When `compile-buffer-file-confirm` is true, the user is always asked for confirmation, even when the date of the source file is later than that of the binary file.

A prefix argument causes the file to be compiled without checking the date or existence of the binary 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`. 
4.9 Evaluation and compilation

**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 Breakpoints

These commands operate on breakpoints, which are points in code where execution stops and the LispWorks IDE invokes the Stepper tool.


4.10.1 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.

4.10.2 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.
4.11 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.

4.11.1 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.

4.11.2 Removing multiple definitions

**Undefine Buffer**

*Editor Command*

Argument: None
Key sequence: None

Undefines all the definitions in the current buffer.
Undefine Region

Argument: None
Key sequence: None

Undefines the definitions in the current region.
By default the LispWorks Editor emulates GNU Emacs. This is often unusable for programmers familiar only with Microsoft Windows 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 Microsoft Windows model instead of Emacs.

When using Microsoft Windows emulation the main differences are:

- An alternate set of key bindings for the commonly-used commands.
- The Alt key activates the menu bar, and does not act as Meta when used as the only modifier key.
- 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 Microsoft Windows editor emulation

To switch Microsoft Windows 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 Microsoft Windows emulation are supplied in the Lisp-Works library file config\msw-key-binds.lisp. This file is loaded the first time that you use Microsoft Windows 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 msw-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 107. Key bindings can also be defined programmatically via editor:bind-key forms similar to those in msw-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 Microsoft Windows emulation is on, Emacs keys are still available via the prefix Ctrl+E. For example, to invoke the command WFind File, enter:
5.3 Replacing the current selection

Ctrl+E Ctrl+X Ctrl+F

Note that you will not have Alt behaving as the Meta key. However 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 Microsoft Windows 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 Microsoft Windows emulation. See “Delete Selection” on page 55 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 editor emulation on Cocoa, and Emacs emulation on Unix and Linux.

To override this behavior in your interface classes, define a method on capi:interface-keys-style. See the LispWorks CAPI Reference Manual for details.
To override this behavior in your delivered application, use the delivery keyword \texttt{:editor-style}. See the \textit{LispWorks Delivery User Guide} for details.
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 100, “Key bindings” on page 107, “Editor variables” on page 106). 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 162.

- The indentation used for Lisp forms can be modified to suit the preferences of the user—see “Customizing Lisp indentation” on page 164.

- Additional editor commands can be created by combining existing commands and providing specified arguments for them—see “Programming the editor” on page 164.

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 107.

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**

```lisp
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 `:pc`, meaning that the binding is for Emacs emulation or Microsoft Windows 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

```lisp
(bind-key "Command" "Control-Right")
```
6.1 Customizing default key bindings

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 Microsoft Windows emulation.

It is best to specify the intended emulation:

(editor:bind-key "Command" "Control-Right" :global :pc)

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 "Regexp Forward Search" \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 \texttt{editor:bind-key}. See the file \emph{config/msw-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 \texttt{config/indents.lisp} in the LispWorks library directory. If you want to alter your personal Lisp indentation, put the modifying code in your \texttt{.lispworks} file.

\texttt{editor:setup-indent}

\texttt{editor:setup-indent form-name no-of-args \&optional standard special}

Modifies the indentation, in Lisp Mode, for the text following an instance of \texttt{form-name}. The arguments \texttt{no-of-args}, \texttt{standard} and \texttt{special} should all be integers. The first \texttt{no-of-args} forms following the \texttt{form-name} become indented \texttt{special} spaces if they are on a new line. All remaining forms within the scope of the \texttt{form-name} become indented \texttt{standard} spaces.

For example, the default indentation for \texttt{if} in Lisp code is established by:

\begin{verbatim}
(editor:setup-indent "if" 2 2 4)
\end{verbatim}

This determines that the first 2 forms after the \texttt{if} (that is, the \texttt{test} and the \texttt{then} clauses) get indented 4 spaces relative to the \texttt{if}, and any further forms (here, just an \texttt{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 (see \texttt{editor:defcommand}).

Note that all the code found in this chapter is included in the directory \texttt{examples/editor} in the directory defined by the variable \texttt{*lispworks-directory*}. 
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 *LispWorks CAPI 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

defcommand

Summary  Defines new editor commands.

Package   editor

Signature  defcommand name lambda-list command-doc function-doc &body forms => command-name

Arguments  name The name of the new editor command. See Description for more details.

lambda-list The lambda list of the new command, which must have at least one argument.

command-doc A string which gives a detailed description of the command.

function-doc A string which gives a brief description of the command.

forms The Lisp code for the command.

Values   command-name The symbol naming the new command. This symbol is interned in the current package.

Description This macro defines a new editor command name that can be invoked in the editor by means of Extended Command. The macro takes the specification of the command as supplied, and creates a new Lisp function command-name from it.

Note: Every editor command has an associated Lisp function named command-command. For example:

editor:self-insert-command p &optional char

For every editor command documented in this manual, the associated command-command symbol is exported from the
editor package; and for every editor command created by user code, the associated command-command symbol is interned in the current package.

Existing editor commands can be used within the body of defcommand. To make use of an existing command, the command name should be hyphenated with a command suffix added. For example, the editor command Forward Character is referred to by forward-character-command. The syntax of a call to an existing command is the same as a call to a standard Lisp function. The first argument of all command definitions is the prefix argument, and this must therefore be included in any calls made to commands from defcommand, even when prefix arguments are ignored by the command. Some commands have additional optional arguments and details of these are provided in the command descriptions throughout this manual.

The name of the command must be a string, while the name of the associated 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 the function needs as a name is computed from that string. Any spaces in the string are replaced with hyphens, and the quotes are discarded, but otherwise the symbol contains the same characters as the string.

If a specific function name, different to the one defcommand derives itself, is required, then this can be supplied explicitly, by passing a list as name. The first element of the list is the string used as the name of the command, while the last element is the symbol used to name the Lisp function.

The command-doc and function-doc variables may be empty strings if no documentation is available for them.

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 first string gives the command’s name. This is the simple form of definition, where no explicit name for the Lisp function is given.

p is not used, and is there simply because the lambda-list must have at least one element.

The second string is the command documentation, while the third is the function documentation. After these, the Lisp code defines what the command actually does.

Use **Alt+X Move Five** to invoke the command.

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)))

### 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 174).
6.3 Programming the editor

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:

\[
\begin{align*}
&\text{(editor:with-buffer-locked ((editor:current-buffer))} \\
&(\text{editor:delete-next-character-command nil)} \\
&(\text{editor:insert-character (editor:current-point)} \\
&\quad \text{#\Space}))
\end{align*}
\]

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**

\[
\begin{align*}
&\text{editor:with-buffer-locked (buffer}} \\
&\quad \text{&key for-modification check-file-modification block-interrupts) } \\
&\text{&body body => values}
\end{align*}
\]

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 edi-
tor:editor-error is signalled. 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
6.3 Programming the editor

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.

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`. 

171
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.
6.3 Programming the editor

**editor:window-buffer**  
*editor:window-buffer window*  
Returns the buffer currently associated with *window*.

**editor:buffers-start**  
*editor:buffers-start buffer*  
Returns the starting point of *buffer*.

**editor:buffers-end**  
*editor:buffers-end buffer*  
Returns the end point of *buffer*.

**editor:buffer-point**  
*editor:buffer-point buffer*  
Returns the current point in *buffer*.

**editor:use-buffer**  
*editor:use-buffer buffer &body forms*  
Makes *buffer* the current buffer during the evaluation of *forms*.

**editor:buffer-from-name**  
*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**  
*editor:make-buffer name &key modes*
Creates a new buffer called *name*. The argument *modes* is a list of modes for the new buffer. The default value for *modes* is *Fundamental*. The newly-created buffer is returned.

**editor:goto-buffer**

*editor:goto-buffer* *buffer insame-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 *insame-window* is non-nil then it is shown in the current window, otherwise a new window is created for it.

### 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 169)

*: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

<table>
<thead>
<tr>
<th>kind</th>
<th>Insert at &lt; ( n )</th>
<th>Insert at = ( n )</th>
<th>Insert at &gt; ( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>:before-insert</td>
<td>( n+len )</td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>:after-insert</td>
<td>( n+len )</td>
<td>( n+len )</td>
<td>( n )</td>
</tr>
</tbody>
</table>

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 \texttt{editor:delete-point}, and point objects are also destroyed if their buffer is killed.

\texttt{editor:point-kind} \hspace{1cm} \textit{Function}

\texttt{editor:point-kind point}

Returns the kind of the point, which is :temporary, :before-insert or :after-insert.

\texttt{editor:current-point} \hspace{1cm} \textit{Function}

\texttt{editor:current-point}

Returns the current point. See also \texttt{editor:buffer-point}.

\texttt{editor:current-mark} \hspace{1cm} \textit{Function}

\texttt{editor:current-mark &optional pop-pno-error-p}

Returns the current mark. If \texttt{pop-p} is \texttt{t}, the mark ring is rotated so that the previous mark becomes the current mark. If no mark is set and \texttt{no-error-p} is \texttt{t}, \texttt{nil} is returned; otherwise an error is signalled. The default for both of these optional arguments is \texttt{nil}.
**editor:current-mark**

**editor:current-mark point**

Sets the current mark to be `point`.

**editor:point<**

**editor:point< point1 point2**

Returns non-nil if `point1` is before `point2` in the buffer.

**editor:point<=**

**editor:point<= point1 point2**

Returns non-nil if `point1` is before or at the same offset as `point2` in the buffer.

**editor:point>**

**editor:point> point1 point2**

Returns non-nil if `point1` is after `point2` in the buffer.

**editor:point>=**

**editor:point>= point1 point2**

Returns non-nil if `point1` is after or at the same offset as `point2` in the buffer.

**editor:copy-point**

**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.
6.3 Programming the editor

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

\texttt{editor:with-point point-bindings \texttt{&rest body}}

point-bindings is a list of bindings, each of the form \texttt{(var point \{kind\})}. Each variable \texttt{var} is bound to a new point which is a copy of the point \texttt{point} though possibly with a different kind, if \texttt{kind} is supplied. If \texttt{kind} is not supplied, then the new point has \texttt{kind:temporary}.

The forms of body are evaluated within the scope of the point bindings, and then the points in each variable \texttt{var} are deleted, as if by \texttt{editor:delete-point}. Each point \texttt{var} is deleted even if there was an error when evaluating body.

The main reason for using \texttt{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 The echo area

editor:message

Function

\texttt{editor:message string \texttt{&rest args}}

A message is printed in the Echo Area. The argument \texttt{string} must be a string, which may contain formatting characters to be interpreted by \texttt{format}. The argument \texttt{args} consists of arguments to be printed within the string.

editor:clear-echo-area

Function

\texttt{editor:clear-echo-area \texttt{&optional string force}}

Clears the Echo Area. The argument \texttt{string} is then printed in the Echo Area. If \texttt{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.
6.3.6 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.
6.3.7 Files

**editor:find-file-buffer**

`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**

`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**

`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**

`editor:buffer-pathname` `buffer`
Returns the pathname of the file associated with \textit{buffer}. If no file is associated with \textit{buffer}, \texttt{nil} is returned.

6.3.7.1 File encodings in the editor

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 \texttt{output-format-default} is used. If the value of \texttt{output-format-default} is not an external-format specifier, then the external format is chosen similarly to the way \texttt{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 \texttt{Set External Format} to set interactively the external format for the current buffer, or set \texttt{Preferences... > Environment > File Encodings > Output} (which in turn sets the editor variable \texttt{output-format-default}) to provide a global default value.

In an application which writes editor buffers to file, you can do this to set the external format of a given buffer:

\begin{verbatim}
(setf (editor:buffer-external-format buffer) ef-spec)
\end{verbatim}

You can also set a global default external format for editor buffers:

\begin{verbatim}
(setf (editor:variable-value 'editor:output-format-default :global) ef-spec)
\end{verbatim}

Then \texttt{ef-spec} will be used when a buffer itself does not have an external format.

6.3.7.2 Non base-char errors

If your buffer contains a \texttt{cl:extended-char char} then Latin-1 and other encodings which support only \texttt{cl:base-char} are not appropriate. Attempts to save the buffer using such external formats will signal an error ‘\texttt{char} is not of type BASE-CHAR’. Set the external format to one which includes \texttt{char}, or delete \texttt{char} from the buffer before saving.
6.3.7.3 Choosing the encoding to use

You may want a file which is Unicode UCS-2 encoded (external format :unicode), UTF-8 encoding (:utf) or a language-specific encoding such as :shift-jis or :gbk. Or you may want an Latin-1 encoded file, in which case you could pass :latin-1-safe.

6.3.8 Inserting text

**editor:insert-string**

```
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**

```
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**

```
editor:points-to-string start end
```

Returns the string between the points start and end.
6.3 Programming the editor

6.3.9 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.

6.3.10 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)**.
6.3.11 Movement

**editor:line-end**

**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**

**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**

**editor:character-offset** *point* *n*

Moves *point* forward *n* characters. If *n* is negative, *point* moves back *n* characters.

**editor:word-offset**

**editor:word-offset** *point* *n*

Moves *point* forward *n* words. If *n* is negative, *point* moves back *n* words.

**editor:line-offset**

**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 signalled if there are not *n* further lines in the buffer.
6.3 Programming the editor

editor:form-offset

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.

6.3.12 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

editor:prompt-for-file &key direction must-exist create-directories default default-string prompt help

Prompts for a file name, and returns a pathname.
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.
6.3 Programming the editor

**editor:prompt-for-variable**

```
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`.

### 6.3.13 In-place completion

**editor:complete-with-non-focus**

```
editor:complete-with-non-focus 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 completion.

If insert-func is not supplied, the completion item is inserted. If it is not a string it is first converted by prinl-to-string.

When editor:complete-with-non-focus 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-with-non-focus 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 Lisp-Works CAPI User Guide.

6.3.14 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.
6.3 Programming the editor

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 upercasing, 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

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

editor::variable-value name &optional kind where

Returns the value of the editor variable name, where name is a symbol. An error is signalled 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 buffer 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 this function. For example, the code given below will allow buffers to be saved to file without any prompt for a missing newline.

```lisp
(setf (editor:variable-value 'editor:add-newline-at-eof-on-writing-file) nil)
```

**editor:variable-value-if-bound**

*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**

*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 signalled. If the variable is undefined and errorp is false, nil is returned. The default value of errorp is nil.

### 6.3.15 Windows

**editor:current-window**

*editor:current-window*

Returns the current window.

**editor:redisplay**

*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**

```
(editor:window-text-pane window)
```

Returns the `capi:editor-pane` associated with an editor window.

### 6.3.16 Examples

#### 6.3.16.1 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)))
```

#### 6.3.16.2 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)))

6.3.16.3 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/6-0-0-0/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 LispWorks 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.
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.

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 be 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 Microsoft Windows platform. Keys, cursors, behavior with selected text and other functionality differs. We use the term Microsoft Windows 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 Alt 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 Alt+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 Alt), 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.

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 location of a point so that it 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 two modes associated with it: a major mode and a minor mode. A buffer must have one major mode, but can have zero or more minor modes associated with it. 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 location in a buffer where editor commands take effect. The current point is generally the location 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 others 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 \texttt{Space}, \texttt{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 \textit{pasting}. 
Symbols

# files 24
? Help on Parse 103
~ files 24, 33

A
Abbrev Expand Only 97
Abbrev Mode 95
abbrev mode 92, 95
Abbreviated Complete Symbol 130
abbreviation
add global 96
add global expansion 96
add mode 95
add mode expansion 96
append to file 99
delete all 98
delete global 98
delete mode 97
edit 99
editor definition 95
expand 97
list 98
read from file 100
save to file 99
undo last expansion 97
abbreviation commands 95
abbrev-pathname-defaults 99
Abort Recursive Edit 107
aborting editor commands 11, 16

aborting processes 11, 16
Add Global Word Abbrev 96
Add Mode Word Abbrev 95
add-newline-at-eof-on-writing-file 29
Alt key 9
Alt+* Insert Double Quotes For Selection 137
Alt+# Insert Multi Line Comment For Selection 139
Alt+{ ( Insert Parentheses For Selection 142
Alt+)} Move Over ) 142
Alt+, Continue Tags Search 123
Alt+. Find Source 118
Alt+/ Dynamic Completion 55
Alt+; Indent for Comment 139
Alt<= Beginning of Buffer 41
Alt<= Beginning Of Parse 104
Alt== Function Arglist 131
Alt=> End of Buffer 42
Alt+? Find Tag 121
Alt+@ Mark Word 44
Alt+[ Backward Paragraph 39
Alt\ Delete Horizontal Space 49
Alt+] Forward Paragraph 39
Alt+’ Word Abbrev Prefix Point 97
Alt+A Backward Sentence 39
Alt+B Backward Word 37
Alt+B Echo Area Backward Word 104
Alt+C Capitalize Word 57
Alt+Ctrl+. Rotate Active Finders 124
Alt+Ctrl+; Kill Comment 140
Alt+Ctrl+@ Mark Form 134
Alt+Ctrl+\ Indent Region 61
Alt+Ctrl+A Beginning of Defun 117
Alt+Ctrl+B Backward Form 134
Alt+Ctrl+D Down List 138
Alt+Ctrl+Delete Backward Kill Form 135
Alt+Ctrl+E End of Defun 117
Alt+Ctrl+F Forward Form 134
Alt+Ctrl+H Mark Defun 117
Alt+Ctrl+I Complete Symbol 129, 134
Alt+Ctrl+K Forward Kill Form 135
Alt+Ctrl+L Select Previous Buffer 68
Alt+Ctrl+N Forward List 137
Alt+Ctrl+P Backward List 137
Alt+Ctrl+Q Indent Form 134
Alt+Ctrl+R Regexp Reverse Search 85
Alt+Ctrl+S Regexp Forward Search 85
Alt+Ctrl+Shift+A Show Documentation 144
Alt+Ctrl+Shift+L Circulate Buffers 68
Alt+Ctrl+Space Pop Mark 44
Alt+Ctrl+T Transpose Forms 136
Alt+Ctrl+U Backward Up List 138
Alt+Ctrl+W Append Next Kill 52
Alt+Ctrl+X Evaluate Defun 146
Alt+Ctrl+Z Exit Recursive Edit 107
Alt+D Kill Next Word 50
Alt+Delete E...
Index

Beginning of Line 37
Beginning Of Parse 104
Beginning of Parse or Line 105
Bind Key 107
Bind String to Key 108
binding
  editor definition 10
binding keys 107
bind-key 162
bind-string-to-key 163
Bottom of Window 41
Break Definition 126
Break Definition on Exit 126
Break Function 126
Break Function on Exit 126
breaking processes 17
break-on-editor-error 113
buffer
  changed definitions in 130
    circulate 68
    compile 151
    compile changed definitions 152
    compile if necessary 151
    create 69
    editor definition 6
    evaluate 148
    evaluate changed definitions 148
    file options 35
      insert 70
      kill 36, 68
      list 69
    mark whole 45
    modified check 70
    move to beginning 41
    move to end 42
    new 69
    not modified 71
    read only 70
    rename 70
    revert 34
    save 27
    search all 80
    select 67
    select in other window 68
    select previous 68
    set package 145
    buffer 168
Buffer Changed Definitions 130
buffer commands 67
buffer functions 168, 191
Buffer Not Modified 71
buffer-from-name 173
*buffer-list* 172
buffer-name 172
buffer-pathname 180
buffer-point 173
buffers and windows 112
buffers-end 173
buffers-start 173
buffer-value 191
bug
  reporting 113, 114
Build Application 114
button
  mouse bindings in editor 112
C
calling editor functions 165
Capitalize Region 58
Capitalize Word 57
case conversion commands 56
case-replace 87
CD 110
Center Line 65
table of contents 171
change-buffer-lock-for-
  modification 171
character
  backward 37
  delete expanding tabs 48
  delete next 48
  delete previous 48
  forward 37
  insert with overwrite 60
  overwrite previous 60
  transposition 58
character-offset 184
Check Buffer Modified 70
check-disk-version-consistent 180
Circulate Buffers 68
class
  describe 132
Class Browser tool 132
clear-echo-area 172
colors
  Lisp syntax 116
command
  abort 16
  completion 10, 18, 102
  description 20, 21
  execution 9, 17, 165
  history 22
  key sequence for 22
  key sequences 23
  listing with apropos 19
Index

repetition 11, 23
shell 109
commands
abbreviation 95
aborting commands 11, 16
aborting processes 11, 16
buffer 67
case conversion 56
compilation 145, 150
cut and paste 13
deleting text 13, 47
echo area 102
editing Lisp programs 115
editor variable 106
evaluation 145, 146, 149
file handling 11, 24
filling 63
help 14, 18
indentation 60
inserting text 12, 52
key binding 107
keyboard macro 100
killing text 13, 47
Lisp comment 138
Lisp documentation 143
Lisp form 134
Lisp function and definition 117
Lisp list 137
movement 12, 36
overwriting 59
pages 74
parentheses 141, 143
recursive editing 106
register 89
replacing 76
running shell from editor 109
searching 76
transposition 58
undoing 13, 55
window 71
comment
create 139
kill 140
move to 139
comment commands 138
comment-begin 141
comment-column 141
comment-end 141
comments
inserting 139
comment-start 141
compilation commands 145, 150
 compilation messages
finding the source code 153
compile
buffer 151
buffer changed definitions 152
buffer if necessary 151
changed definitions 152
file 151
form 150
region 151
system 153
system changed definitions 153
Compile Buffer 151
Compile Buffer Changed Definitions 152
Compile Buffer File 151
Compile Changed Definitions 152
Compile Defun 150
Compile File 151
Compile Region 151
Compile System 153
Compile System Changed Definitions 153
compile-buffer-file-confirm 152
Complete Field 103
Complete Input 102
Complete Symbol 129, 134
complete-with-non-focus 187
completion
dynamic word 55
in-place 187
of abbreviated symbols 130
of commands 10, 18, 102
of filenames 55
of symbols 129, 134
configuration files 158, 161
Confirm Parse 103
Continue Tags Search 123
Control key 9
count keys
insert into buffer 54
Copy to Cut Buffer 112
Copy to Register 90
copy-point 176
Count Lines Page 75
Count Lines Region 46
Count Matches 85
Count Occurrences 85
Count Words Region 45
Create Buffer 69
Create Tags Buffer 121
Ctrl key 9
Ctrl+] Abort Recursive Edit 107
Ctrl+‘ Function Arglist Display 131
Ctrl+A Beginning of Line 37
Ctrl+A Beginning Of Parse or Line 105
Ctrl+B Backward Character 37
Ctrl+B Echo Area Backward Character 104
Ctrl+Break, break gesture 17
Ctrl+C Ctrl+C Insert Selected Text 106
Ctrl+D Delete Next Character 48
Ctrl+D End of Line 38
Ctrl+E Forward Character 37
Ctrl+G, abort current command 16
Ctrl+H A Apropos 16
Ctrl+H A Apropos Command 19
Ctrl+H B Describe Bindings 23
Ctrl+H C What Command 20
Ctrl+H Ctrl+D Document Command 20
Ctrl+H Ctrl+V Document Variable 22
Ctrl+H D Describe Command 14, 20
Ctrl+H G Generic Describe 21
Ctrl+H Help 18
Ctrl+H K Describe Key 14, 21
Ctrl+H L What Lassage 22
Ctrl+H V Describe Editor Variable 22
Ctrl+H W Where Is 22
Ctrl+I Kill Line 51
Ctrl+L Refresh Screen 73
Ctrl+M Next Line 38
Ctrl+O Open Line 54
Ctrl+P Insert Parse Default 105
Ctrl+P Previous Line 38
Ctrl+Q Quoted Insert 54
Ctrl+R Return Default 106
Ctrl+R Reverse Incremental Search 77
Ctrl+R Esc Forward Search 78
Ctrl+R Incremental Search 76
Ctrl+Shift+_ Undo 13, 56
Ctrl+Shift+A Function Argument List 132
Ctrl+Shift+B Compile Buffer 151
Ctrl+Shift+C Compile Defun 150
Ctrl+Shift+D Function Documentation 144
Ctrl+Shift+E Evaluate Region 147
Ctrl+Shift+M Macroexpand Form 136
Ctrl+Shift+R Compile Region 151
Ctrl+Space Set Mark 43
Ctrl+T Transpose Characters 58
Ctrl+U Kill Parse 105
Ctrl+V Set Prefix Argument 23
Ctrl+V Scroll Window Down 39
Ctrl+W Kill Region 52
Ctrl+X - Inverse Add Global Word Abbrev 96
Ctrl+X & Search Region Matching Patterns 81
Ctrl+X ( Define Keyboard Macro 101
Ctrl+X ) End Keyboard Macro 101
Ctrl+X * Search Files 81
Ctrl+X + Add Global Word Abbrev 96
Ctrl+X . Set Fill Prefix 65
Ctrl+X / Point to Register 89
Ctrl+X ; Set Comment Column 138
Ctrl+X [ Previous Page 74
Ctrl+X ] Next Page 74
Ctrl+X - Check Buffer Modified 70
Ctrl+X 0 Delete Window 72
Ctrl+X 1 Delete Next Window 72
Ctrl+X 2 New Window 71
Ctrl+X B Select Buffer 67
Ctrl+X C Go Back 47
Ctrl+X Ctrl+A Add Mode word Abbrev 95
Ctrl+X Ctrl+B List Buffers 69
Ctrl+X Ctrl+C Save All Files and Exit 29
Ctrl+X Ctrl+E Evaluate Last Form 147
Ctrl+X Ctrl+F Wfind File 26
Ctrl+X Ctrl+H Inverse Add Mode Word Abbrev 96
Ctrl+X Ctrl+I Indent Rigidly 62
Ctrl+X Ctrl+L Lowercase Region 57
Ctrl+X Ctrl+O Delete Blank Lines 49
Ctrl+X Ctrl+P Mark Page 75
Ctrl+X Ctrl+Q Toggle Buffer Read-Only 70
Ctrl+X Ctrl+S Save File 27
Ctrl+X Ctrl+T Transpose Lines 59
Ctrl+X Ctrl+U Uppercase Region 57
Ctrl+X Ctrl+V Find Alternate File 26
Ctrl+X Ctrl+W Write File 28
Ctrl+X Ctrl+X Exchange Point and Mark 44
Ctrl+X Delete Backward Kill Sentence 51
Ctrl+X E Last Keyboard Macro 101
Ctrl+X F Set Fill Column 64
Ctrl+X G Insert Register 90
Ctrl+X H Mark Whole Buffer 45
Ctrl+X I Insert File 35
Ctrl+X J Jump to Register 89
Ctrl+X K Kill Buffer 68
Ctrl+X L Count Lines Page 75
Index

Ctrl+X M Select Go Back 47
Ctrl+X O Next Ordinary Window 71
Ctrl+X O Next Window 71
Ctrl+X P Go Forward 47
Ctrl+X Q Keyboard Macro Query 101
Ctrl+X S Save All Files 27
Ctrl+X Tab Indent Rigidly 62
Ctrl+X X Copy to Register 90
Ctrl+Y Un-Kill 14, 52
Ctrl-C Ctrl-C Interrupt Shell Subjob 111
Ctrl-C Ctrl-D Shell Send Eof 111
Ctrl-C Ctrl-Z Stop Shell Subjob 111
current point
editor definition 7
current-buffer 172
current-mark 175
current-package 145
current-point 175
current-window 191
customising
editor commands 161
indention of Lisp forms 161, 164
key bindings 158, 161, 162
cut and paste commands 13
d
debugger
using in editor 148
default
binding 10
external format 27, 30
prefix argument 23, 24
default-auto-save-on 32
default-buffer-element-type 69
default-modes 92
default-search-kind 82
defcommand macro 166
Defindent 117
Define Keyboard Macro 101
Define Word Abbrevs 100
define-editor-mode-variable 189
define-editor-variable 188
definition
break 126
disable 153
defining 117
find 118
find buffer changes 130
searching for 118
trace 125
trace inside 125
untrace 125
defmode function 93
Delete All Word Abbrevs 98
Delete Blank Lines 49
DELETE Delete Previous Character 48
DELETE Echo Area Delete Previous Character 105
Delete File 35
Delete File and Kill Buffer 36
Delete Global Word Abbrev 98
Delete Horizontal Space 49
Delete Indentation 62
Delete Key Binding 108
Delete Matching Lines 79
Delete Mode Word Abbrev 97
Delete Next Character 48
Delete Next Window 72
Delete Non-Matching Lines 80
Delete Previous Character 48
Delete Previous Character Expanding Tabs 48
Delete Region 49
Delete Selection Mode 55
Delete Window 72
delete-point 177
deleting text 48
deleting text commands 13, 47
deletion
editor definition 47
of selection 55
of surrounding form 135
delimiter
sentence 9
Describe Bindings 23
Describe Class 132
Describe Command 20
Describe Editor Variable 22
Describe Generic Function 133
Describe Key 21
Describe Method Call 133
Describe Symbol 143
Describe System 133
Diff 88
Diff Ignoring Whitespace 89
directory
change 110
query replace 87
search 80
Directory Query Replace 87
Directory Search 80
Disassemble Definition 153
**Index**

<table>
<thead>
<tr>
<th>Command/Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do Nothing</td>
<td>109</td>
</tr>
<tr>
<td>Document Command</td>
<td>20</td>
</tr>
<tr>
<td>Document Key</td>
<td>21</td>
</tr>
<tr>
<td>Document Variable</td>
<td>22</td>
</tr>
<tr>
<td>documentation commands</td>
<td>143</td>
</tr>
<tr>
<td>double-quotes</td>
<td>137</td>
</tr>
<tr>
<td>Down Comment Line</td>
<td>140</td>
</tr>
<tr>
<td>Down List</td>
<td>138</td>
</tr>
<tr>
<td>dspec</td>
<td>144</td>
</tr>
<tr>
<td>Dynamic Completion</td>
<td>55</td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>echo area</td>
<td></td>
</tr>
<tr>
<td>complete text</td>
<td>102</td>
</tr>
<tr>
<td>completing commands in</td>
<td>102</td>
</tr>
<tr>
<td>deleting and inserting text in</td>
<td>105</td>
</tr>
<tr>
<td>editor definition</td>
<td>102</td>
</tr>
<tr>
<td>help on parse</td>
<td>103</td>
</tr>
<tr>
<td>movement in</td>
<td>104</td>
</tr>
<tr>
<td>next command</td>
<td>104</td>
</tr>
<tr>
<td>previous command</td>
<td>103</td>
</tr>
<tr>
<td>prompting the user</td>
<td>185</td>
</tr>
<tr>
<td>repeating commands in</td>
<td>103</td>
</tr>
<tr>
<td>terminate entry</td>
<td>103</td>
</tr>
<tr>
<td>Echo Area Backward Character</td>
<td>104</td>
</tr>
<tr>
<td>Echo Area Backward Word</td>
<td>104</td>
</tr>
<tr>
<td>echo area commands</td>
<td>102</td>
</tr>
<tr>
<td>Echo Area Delete Previous Character</td>
<td>105</td>
</tr>
<tr>
<td>echo area functions</td>
<td>178, 192</td>
</tr>
<tr>
<td>Echo Area Kill Previous Word</td>
<td>105</td>
</tr>
<tr>
<td>Edit Calleees</td>
<td>128</td>
</tr>
<tr>
<td>Edit Callers</td>
<td>128</td>
</tr>
<tr>
<td>Edit Editor Command</td>
<td>120</td>
</tr>
<tr>
<td>Edit Recognized Source</td>
<td>153</td>
</tr>
<tr>
<td>Edit Word Abbrevs</td>
<td>99</td>
</tr>
<tr>
<td>editor</td>
<td></td>
</tr>
<tr>
<td>customising</td>
<td>161</td>
</tr>
<tr>
<td>delete-region-command</td>
<td>49</td>
</tr>
<tr>
<td>programming</td>
<td>164</td>
</tr>
<tr>
<td>editor commands</td>
<td></td>
</tr>
<tr>
<td>Abbrev Expand Only</td>
<td>97</td>
</tr>
<tr>
<td>Abbrev Mode</td>
<td>95</td>
</tr>
<tr>
<td>Abbreviated Complete Symbol</td>
<td></td>
</tr>
<tr>
<td>Alt+X 130</td>
<td></td>
</tr>
<tr>
<td>Abort Recursive Edit Ctrl+1</td>
<td>107</td>
</tr>
<tr>
<td>Add Global Word Abbrev Ctrl+X + 96</td>
<td></td>
</tr>
<tr>
<td>Add Mode Word Abbrev Ctrl+X</td>
<td></td>
</tr>
<tr>
<td>Ctrl+A 95</td>
<td></td>
</tr>
<tr>
<td>Append Next Kill Alt+Ctrl+W</td>
<td>52</td>
</tr>
<tr>
<td>Append to File</td>
<td>29</td>
</tr>
<tr>
<td>Append to Word Abbrev File</td>
<td>99</td>
</tr>
<tr>
<td>Apropos</td>
<td>143</td>
</tr>
<tr>
<td>Apropos Command Ctrl+H A</td>
<td>19</td>
</tr>
<tr>
<td>Auto Fill Linefeed</td>
<td>67</td>
</tr>
<tr>
<td>Auto Fill Mode</td>
<td>66</td>
</tr>
<tr>
<td>Auto Fill Return</td>
<td>67</td>
</tr>
<tr>
<td>Auto Fill Space</td>
<td>66</td>
</tr>
<tr>
<td>Auto Save Toggle</td>
<td>32</td>
</tr>
<tr>
<td>Back to Indentation</td>
<td>63</td>
</tr>
<tr>
<td>Backup File</td>
<td>29</td>
</tr>
<tr>
<td>Backward Character Ctrl+B</td>
<td>37</td>
</tr>
<tr>
<td>Backward Form Alt+Ctrl+B</td>
<td>134</td>
</tr>
<tr>
<td>Backward Kill Form</td>
<td></td>
</tr>
<tr>
<td>Alt+Ctrl+Delete</td>
<td>135</td>
</tr>
<tr>
<td>Backward Kill Line</td>
<td>51</td>
</tr>
<tr>
<td>Backward Kill Sentence Ctrl+X</td>
<td></td>
</tr>
<tr>
<td>Delete 51</td>
<td></td>
</tr>
<tr>
<td>Backward List Alt+Ctrl+P</td>
<td>137</td>
</tr>
<tr>
<td>Backward Paragraph Alt+f</td>
<td>39</td>
</tr>
<tr>
<td>Backward Search</td>
<td>79</td>
</tr>
<tr>
<td>Backward Sentence Alt+A</td>
<td>39</td>
</tr>
<tr>
<td>Backward Up List Alt+Ctrl+U</td>
<td>138</td>
</tr>
<tr>
<td>Backward Word Alt+B</td>
<td>37</td>
</tr>
<tr>
<td>Beginning of Buffer Alt+&lt; 41</td>
<td></td>
</tr>
<tr>
<td>Beginning of Defun Alt+Ctrl+X 117</td>
<td></td>
</tr>
<tr>
<td>Beginning of Line Ctrl+A</td>
<td>37</td>
</tr>
<tr>
<td>Beginning Of Parse Alt+&lt; 104</td>
<td></td>
</tr>
<tr>
<td>Beginning of Parse or Line Ctrl+A 105</td>
<td></td>
</tr>
<tr>
<td>Bind Key</td>
<td>107</td>
</tr>
<tr>
<td>Bind String to Key</td>
<td>108</td>
</tr>
<tr>
<td>Bottom of Window</td>
<td>41</td>
</tr>
<tr>
<td>Break Definition</td>
<td>126</td>
</tr>
<tr>
<td>Break Definition on Exit</td>
<td>126</td>
</tr>
<tr>
<td>Break Function</td>
<td>126</td>
</tr>
<tr>
<td>Break Function on Exit</td>
<td>126</td>
</tr>
<tr>
<td>Buffer Changed Definitions</td>
<td>130</td>
</tr>
<tr>
<td>Buffer Not Modified Alt+Shift++ 71</td>
<td></td>
</tr>
<tr>
<td>Build Application</td>
<td>114</td>
</tr>
<tr>
<td>Capitalize Region</td>
<td>58</td>
</tr>
<tr>
<td>Capitalize Word Alt+c</td>
<td>57</td>
</tr>
<tr>
<td>CD</td>
<td>110</td>
</tr>
<tr>
<td>Center Line</td>
<td>65</td>
</tr>
<tr>
<td>Check Buffer Modified Ctrl+X - 70</td>
<td></td>
</tr>
<tr>
<td>Circulate Buffers Alt+Ctrl+Shift+L 68</td>
<td></td>
</tr>
<tr>
<td>Compile Buffer Changed Definitions</td>
<td>152</td>
</tr>
<tr>
<td>Compile Buffer File</td>
<td>151</td>
</tr>
<tr>
<td>Compile BufferCtrl+Shift+B</td>
<td>151</td>
</tr>
<tr>
<td>Compile Changed Definitions</td>
<td>152</td>
</tr>
<tr>
<td>Compile Defun Ctrl+Shift+C</td>
<td>150</td>
</tr>
<tr>
<td>Compile File</td>
<td>151</td>
</tr>
<tr>
<td>Compile Region Ctrl+Shift+R</td>
<td>151</td>
</tr>
</tbody>
</table>

211
Find Source for Dspec 119
Find Tag Alt+? 121
Find Unbalanced Parentheses 143
Find Unwritable Character 31
Forward Character Ctrl+F 37
Forward Form Alt+Ctrl+F 134
Forward Kill Form Alt+Ctrl+K 135
Forward Kill Sentence Alt+K 51
Forward List Alt+Ctrl+N 137
Forward Paragraph Alt+] 39
Forward Kill Sentence Alt+K 51
Forward List Alt+Ctrl+N 137
Forward Paragraph Alt+] 39
Forward List Alt+K 39
Forward Up List 138
Forward Word Alt+F 37
Function Arglist Alt+= 131
Function Arglist Display Ctrl+’ 131
Function Argument List Ctrl+Shift+A 132
Function Documentation Ctrl+Shift+D 144
Fundamental Mode 91
Generic Describe Ctrl+H G 21
Get Register 90
Go Back Ctrl+X C 47
Go Forward Ctrl+X F 47
Goto Line 38
Goto Page 74
Goto Point 42
Help Ctrl+H 18
Help on Parse? 103
Illegal 108
Incremental Search Ctrl+S 76
Indent for Comment Alt+; 139
Indent Form Alt+Ctrl+Q 134
Indent New Comment Line Alt+J or Alt+Newline 140
Indent New Line 63
Indent or Complete Symbol 129
Indent Region Alt+Ctrl+L 61
Indent Rigidly 62
Indent Rigidly Ctrl+X Tab, Ctrl+X Ctrl+I 62
Indent Selection or Complete Symbol TAB 129
Indent TAB 61
Insert () 141
Insert Buffer 70
Insert Cut Buffer 112
Insert Double Quotes For Selection Alt+* 137
Insert File Ctrl+X I 35
Insert Multi Line Comment For Selection Alt+# 139
Insert Page Directory 75
Insert Parentheses For Selection Alt+( 142
Insert Parse Default Ctrl+P 105
Insert Register Ctrl+X G 90
Insert Selected Text Ctrl+C Ctrl+V 106
Insert Word Abbrevs 100
Interrupt Shell Subjob Ctrl-C Ctrl-C 111
Inverse Add Global Word Abbrev Ctrl+X 96
Inverse Add Mode Word Abbrev Ctrl+X Ctrl+H 96
Jump to Register Ctrl+X J 89
Jump to Saved Position 89
Just One Space Alt+Space 49
Keyboard Macro Query Ctrl+X Q 101
Kill Backward Up List 135
Kill Buffer Ctrl+X K 68
Kill Comment Alt+Ctrl+; 140
Kill Line Ctrl+X 51
Kill Next Word Alt+D 50
Kill Parse Ctrl+U 105
Kill Previous Word Alt+Delete 51
Kill Region Ctrl+X W 52
Kill Register 89
Last Keyboard Macro Ctrl+X E 101
Line to Top of Window 41
Lisp Insert ) 142
Lisp Mode 92
List Buffers Ctrl+X Ctrl+B 69
List Calleees 127
List Callers 127
List Definitions 120
List Definitions For Dspec 121
List Matching Lines 79
List Registers 90
List Unwritable Characters 31
List Word Abbrevs 98
Load File 148
Lowercase Region Ctrl+X Ctrl+L 57
Lowercase Word Alt+L 56
Macroexpand Form Ctrl+Shift+M 136
Make Word Abbrev 96
Mark Defun Alt+Ctrl+H 117
Mark Form Alt+Ctrl+@ 134
Mark Page Ctrl+X Ctrl+P 75
Mark Paragraph Alt+H 45
Mark Sentence 45
Mark Whole Buffer Ctrl+X H 45
Mark Word Alt+@ 44
Move Over) Alt+ 142
Move to Window Line Alt+Shift+R 41
Name Keyboard Macro 101
Negative Argument 24
New Buffer 69
New Line RETURN 53
New Window Ctrl+X 2 71
Next Breakpoint 154
Next Line Ctrl+R 38
Next Ordinary Window Ctrl+X O 71
Next Page Ctrl+X 74
Next Parse Alt+N 104
Next Window Ctrl+O 71
Open Line Ctrl+O 54
Overwrite Delete Previous Character 60
Overwrite Mode 60
Point to Register Ctrl+X / 89
Pop Mark Alt+Ctrl+Space 44
Previous Breakpoint 154
Previous Line Ctrl+P 38
Previous Page Ctrl+X f 74
Previous Parse Alt+P 103
Previous Window 72
Print File 34
Print Region 46
Process File Options 35
Put Register 90
Query Replace Alt+Shift+% 86
Query Replace Regexp 88
Quote Tab 63
Quoted Insert Ctrl+Q 54
Read Word Abbrev File 100
Re-evaluate Defvar 147
Reevaluate Defvar 147
Refresh Screen Ctrl+L 73
Regexp Forward Search Alt+Ctrl+S 85
Regexp Reverse Search Alt+Ctrl+R 85
Register to Point 89
Rename Buffer 70
Rename File 36
Replace Regexp 88
Replace String 86
Report Bug 113
Report Manual Bug 114
Return Default Ctrl+R 106
Reverse Incremental Search Ctrl+R 77
Reverse Search 79
Revert Buffer 34
Room 114
Rotate Active Finders Ctrl+. 124
Rotate Active Finders Alt+Ctrl+. 124
Rotate Kill Ring Alt+Y 53
Run Command 109
Save All Files and Exit Ctrl+X Ctrl+C 29
Save All Files Ctrl+X S 27
Save File Ctrl+X Ctrl+S 27
Save Position 89
Save Region Alt+M 52
Scroll Next Window Down 72
Scroll Next Window Up 72
Scroll Window Down Ctrl+V 39
Scroll Window Up Alt+V 40
Search All Buffers 80
Search Files Ctrl+X • 81
Search Files Matching Patterns Ctrl+X & 81
Search System 82
Select Buffer Ctrl+X B 67
Select Buffer Other Window 68
Select Go Back Ctrl+X M 47
Select Previous Buffer Alt+Ctrl+L 68
Self Insert 54
Self Overwrite 60
Set Buffer Output 145
Set Buffer Package 145
Set Comment Column Ctrl+X ; 138
Set External Format 50
Set Fill Column Ctrl+X F 64
Set Fill Prefix Ctrl+X . 65
Set Mark Ctrl+Space 43
Set Prefix Argument Ctrl+U 23
Set Variable 106
Shell Send Eof Ctrl-C Ctrl-D 111
Show Documentation 144
Show Documentation for Dspec 144
Show Paths From 128
Show Paths To 127
Skip Whitespace 42
Stop Shell Subjob Ctrl-C Ctrl-Z 111
System Query Replace 87
System Search 81
Tags Query Replace 123
Tags Search 122
Text Mode 91
Toggle Auto Save 31
Toggle Breakpoint 154
Toggle Buffer Read-Only Ctrl+X Ctrl+Q 70
Toggle Count Newlines 73
Index

Toggle Error Catch 148
Top of Window 41
Trace Definition 125
Trace Definition Inside Definition 125
Trace Function 124
Trace Function Inside Definition 125
Transpose Characters Ctrl+T 58
Transpose Forms Alt+Ctrl+T 136
Transpose Lines Ctrl+X Ctrl+T 59
Transpose Regions 59
Transpose Words Alt+T 58
Undefine 155
Undefine Buffer 155
Undefine Command 155
Undefine Region 156
Undo Ctrl+Shift+ 56
Unexpand Last Word 97
Un-Kill Ctrl+R 52
Untrace Definition 125
Untrace Function 125
Up Comment Line Alt+P 140
Uppercase Region Ctrl+X Ctrl+U 57
Uppercase Word Alt+U 57
View Page Directory 75
View Source Search 120
Visit File 26
Visit Other Tags File 124
Visit Tags File 123
Walk Form Alt+Shift+M 136
Wfind File Ctrl+X Ctrl+F 26
What Command Ctrl+H C 20
What Cursor Position 42
What Line 38
What Lossage Ctrl+R L 22
Where Is Ctrl+W W 22
Where is Point 42
Word Abbrev Apropos 98
Word Abbrev Prefix Point Alt+‘ 97
Write File Ctrl+X Ctrl+W 28
Write Region 28
Write Word Abbrev File 99
Editor errors
  debugging 113
  editor functions
    bind-key 162
    bind-string-to-key 163
    buffer-from-name 173
    buffer-name 172
    buffer-pathname 180
    buffer-point 173
    buffers-end 173
    buffers-start 173
    buffer-value 191
    change-buffer-lock-for-modification 171
    character-offset 184
    check-disk-version-consistent 180
    clear-echo-area 178
    complete-with-non-focus 187
    copy-point 176
    current-buffer 172
    current-mark 175
    current-point 175
    current-window 191
    define-editor-mode-variable 189
    define-editor-variable 188
    delete-point 177
    editor-error 179
    editor-variable-documentation 190
    end-line-p 177
    fast-save-all-buffers 180
    find-file-buffer 180
    form-offset 185
    goto-buffer 174
    insert-string 182
    kill-ring-string 182
    line-end 184
    line-offset 184
    line-start 184
    make-buffer 173
    message 178
    move-point 177
    point< 176
    point<= 176
    point> 176
    point>> 176
    point-kind 175
    points-to-string 182
    process-character 165
    prompt-for-buffer 186
    prompt-for-file 185
    prompt-for-integer 186
    prompt-for-string 186
    prompt-for-variable 187
    redisplay 191
    same-line-p 177
    set-current-mark 176
    set-interrupt-keys 163
    setup-indent 164
    start-line-p 177
    variable-value 190
    variable-value-if-bound 191
    window-buffer 173
    window-text-pane 192
Index

Expand File Name 55
  expression evaluate 147
Extended Command 10, 18
  extended-char type 181
external format
default 27, 30
  setting 30
  unwritable character 31
  unwritable characters 31
external formats 181
Extract List 136

F
  fast-save-all-buffers 180
file
  auto-saving 31
  backup 29, 33
  compile 151
  delete 35, 36
  editor definition 6
  evaluate 148
  expand name 55
  find alternate 26
  finding 25
  insert into buffer 35
  options for buffer 35
  print 34
  rename 36
  save 27, 29
  set external format 30
  unwritable character 31
  unwritable characters 31
  write 28
file encodings 181
file functions 191
file handling commands 11, 24
filename completion 55
files
  search 81
Fill Paragraph 64
Fill Region 64
fill-column 64
filling commands 63
fill-prefix 65
Find Alternate File 26
Find Command Definition 120
Find File 25
Find Mismatch 143
Find Source 118
Find Source for Dspec 119
Find Tag 121
Find Unbalanced Parentheses 143
Find Unwritable Character 31
find-file-buffer 180
finding editor source code 120
*find-likely-function-ignores* 183
form
  compile 150
  evaluate 146, 149
  evaluate last 147, 149
  indent 134
  kill backwards 135
  kill forwards 135
  macro-expand 136
  mark 134
  move to beginning 134
  move to end 134
  transposition 136
form commands 134
form-offset 185
Forward Character 37
Forward Form 134
Forward Kill Form 135
Forward Kill Sentence 51
Forward List 137
Forward Paragraph 39
Forward Search 78
Forward Sentence 39
Forward Up List 138
Forward Word 37
function
  argument list 131
  break 126
  describe generic 133
  documentation 144
  edit callees 128
  edit callers 128
  editing 117
  find definition 118
  indentation 117
  list callees 127, 128
  list callers 127
  mark 117
  move to beginning 117
  move to end 117
  trace 124
  trace inside 125
  untrace 125
Function Arglist 131
Function Arglist Display 131
Function Argument List 132
Function Call Browser tool 127, 128
Function Documentation 144
functions
  buffer 168, 191
  calling 165
  defmode 93
  echo area 178, 192
  editor error 179
  editor, see editor functions
  file 191
  inserting text 182
  Lisp editor 183
  movement 184, 191
  point 174
  prompt 185
  search-files 82
  variable 188
  window 191
Fundamental Mode 91
fundamental mode 91

G
Generic Describe 21
generic function
describe 133
Generic Function Browser tool 133
Get Register 90
global abbreviation
editor definition 95
Go Back 47
Go Forward 47
Goto Line 38
Goto Page 74
Goto Point 42
goto-buffer 174

H
Help 18
help commands 14, 18
Help on Parse 103
highlight-matching-parens 142
history of commands 22
history ring 103

I
Illegal 108
Incremental Search 76
Indent 61
indent
form 134
Indent for Comment 139
Indent Form 134
Indent New Comment Line 140
Indent New Line 63
Indent or Complete Symbol 129
Indent Region 61
Indent Rigidly 62
Indent Selection 62
Indent Selection or Complete Symbol 129
indentation
  customising 161, 164
  define for Lisp forms 117
  define for Lisp functions 117
  delete 62
  move back to 63
  indentation commands 60
  indenting 183
  *indent-with-tabs* 183
In-place completion 187
input-format-default 27
Insert () 141
Insert Buffer 70
Insert Cut Buffer 112
Insert Double Quotes For Selection 137
Insert File 35
Insert Multi Line Comment For Selection 139
Insert Page Directory 75
Insert Parentheses For Selection 142
Insert Parse Default 105
Insert Register 90
Insert Selected Text 106
Insert Word Abbrevs 100
inserting text commands 12, 52
inserting text functions 182
insert-string 182
Interrupt Shell Subjob 111
Inverse Add Global Word Abbrev 96
Inverse Add Mode Word Abbrev 96

J
Jump to Register 89
Jump to Saved Position 89
Just One Space 49

K
key
  Alt 9
  command description 20
  Control 9
description 21
  Escape 9
  key binding 107
customising 158, 161, 162
key sequence
  - editor definition 9
  - for command 22
key sequences
  - for commands 23
keyboard macro
  - begin definition of 101
  - editor definition 100
  - end definition of 101
  - execute 101
  - name 101
keyboard macro commands 100
**Keyboard Macro Query** 101
**Kill Backward Up List** 135
**Kill Buffer** 68
**Kill Comment** 140
**Kill Line** 51
**Kill Next Word** 50
**Kill Parse** 105
**Kill Previous Word** 51
**Kill Region** 52
**Kill Register** 89
  - kill ring 47, 50, 52
  - rotate 53
killing
  - editor definition 47
  - killing text 50
  - killing text commands 13, 47
  - kill-ring-string 182

**L**
**Last Keyboard Macro** 101
  - line
    - beginning 37
    - centre 65
    - count for page 75
    - count for region 46
    - delete blank 49
    - delete matching 79
    - delete non-matching 80
    - end 38
    - goto 38
    - indent new 63
    - indentation 129
    - kill 51
    - kill backward 51
    - length 64
    - list matching 79
    - move to top of window 41
    - next 38
    - open new 53
    - previous 38
  - transposition 59
  - what line 38
  - line count 75
  - **Line to Top of Window** 41
  - line-end 184
  - **LINEFEED Auto Fill Linefeed** 67
  - line-offset 184
  - line-start 184
Lisp
  - editor commands 115
  - Lisp comment commands 138
  - Lisp documentation commands 143
  - Lisp editor functions 183
  - Lisp form commands 134
  - **Lisp Insert** 142
  - Lisp list commands 137
  - **Lisp Mode** 92
  - Lisp mode 91
  - LispWorks IDE tools
    - Application Builder 114
    - Class Browser 132
    - Editor 128
    - Function Call Browser 127, 128
    - Generic Function Browser 133
    - Listener 49, 50, 92, 111, 149
    - Output Browser 50
    - Process Browser 17
    - Search Files 81, 82
    - Shell 109, 111
    - Symbol Browser 143
list
  - extract 135
  - kill backward up 135
  - move down one level 138
  - move to end 137, 138
  - move to start 137, 138
**List Buffers** 69
**List Callees** 127
**List Callers** 127
list commands 137
**List Definitions** 120
**List Definitions For Dspec** 121
**List Matching Lines** 79
**List Registers** 90
**List Unwritable Characters** 31
**List Word Abbrevs** 98
Listener tool 49, 50, 92, 111, 149
**Load File** 148
locations 46
**Lowercase Region** 57
**Lowercase Word** 56

219
Index

M

macro keyboard 100
Macroexpand Form 136
macro-expansion 136
macros defcommand 166
major mode editor definition 8, 90
Make Word Abbrev 96
make-buffer 173
manual on-line editor 20, 21, 22
mark editor definition 7
exchange with point 44
form 134
move current point to 44
paragraph 45
pop 44
sentence 45
set 43
word 44
See also locations
Mark Defun 117
Mark Form 134
Mark Page 75
Mark Paragraph 45
mark ring 43
Mark Sentence 45
Mark Whole Buffer 45
Mark Word 44
message 178
method call describe 133
minor mode editor definition 8, 92
mode editor definition 8, 90
indention in 60
mode abbreviation editor definition 95
mode line editor definition 6
modes abbrev 92, 95
auto-fill 66, 92
execute 92
fundamental 91
Lisp 91
overwrite 60, 92
shell 91
text 91
mouse editor bindings 112
Move Over) 142
Move to Window Line 41
movement command 12, 36
locations 46
movement functions 184, 191
move-point 177
MS windows keys
using 157

N
Name Keyboard Macro 101
Negative Argument 24
New Buffer 69
New in LispWorks 6.0
buffer-value 191
Build Application 114
change-buffer-lock-for-
modification 171
define-editor-mode-variable 189
define-editor-variable 188
Describe Method Call 133
diff ignoring whitespace 89
editor-variable-documentation 190
Find Unwritable Character 31
Function Arglist Display 131
Indent or Complete Symbol 129
indent selection or complete symbol 129
Insert Double Quotes For Selection 137
Insert Multi Line Comment For Selection 139
Insert Parentheses For Selection 142
List Unwritable Characters 31
Make Directory 36
Mark Word 44
Next Breakpoint 154
Previous Breakpoint 154
Toggle Count Newlines 73
New Line 53
New Window 71
newline adding to end of file 29
Next Breakpoint 154
Next Line 38
Next Ordinary Window 71
Next Page 74
Next Parse 104
Index

Next Window  71

O
Open Line  54
Output Browser tool  50
output-format-default  30
Overwrite Delete Previous Character  60
Overwrite Mode  60
overwrite mode  60, 92
overwriting commands  59

P
package
   editor  164
      set  145
Page
display first lines  75
   editor definition  74
goto  74
   insert first lines into buffer  75
mark  75
   next  74
previous  74
Page commands  74
Pane
editor definition  5
Paragraph
backward  39
   editor definition  9
fill  64
forward  39
mark  45
Parentheses
   inserting a pair of  141, 142
Parentheses commands  141
Pending delete  55
Point
   editor definition  7
      exchange with mark  44
   goto  42
   move to window line  41
   position of  42
   save to register  89
      where is  42
Point  174
Point behavior  174
Point functions  174
point ring, see mark ring
Point to Register  89
Point  174
Point<=  176
Point<  176
Point-kind  175
Points and text modification  174
Points-to-string  182
Pop and Goto Mark  44
Pop Mark  44
Prefix
   fill  65
   prefix argument  11, 23
   prefix-argument-default  24
Previous Breakpoint  154
Previous Line  38
Previous Page  74
Previous Parse  103
Previous Window  72
print
   file  34
      region  46
Print File  34
Print Region  46
Process
   breaking  17
Process Browser tool  17
Process File Options  35
process-character  165
Programming the editor  164
   calling functions  165
      examples  192
   prompt functions  185
   prompt-for-buffer  186
   prompt-for-file  185
   prompt-for-integer  186
   prompt-for-string  186
   prompt-for-variable  187
   prompt-regexp-string  111
Put Register  90

Q
Query Replace  86
Query replace  86
   directory  87
   regexp  88
   system  87
   tags  123
Query Replace Regexp  88
Quote Tab  63
Quoted Insert  54

R
Read Word Abbrev File  100
recursive editing 106
redisplay 191
Re-evaluate Defvar 147
Reevaluate Defvar 147
Refresh Screen 73
regexp
query replace 88
replace 88
Regexp Forward Search 85
Regexp Reverse Search 85
region
append 29
capitalize 58
compile 151
delete 49
determining 44
determine 147, 150
evaluate 147, 150
generate from register 90
indent 61
indent rigidly 62
kill 52
line count 46
lowercase 57
print 46
save 52
transposition 59
upper case 57
word count 45
write 28
region-query-size 46
register
editor definition 89
generate region 90
kill 89
list 90
move to saved position 89
record position 90
save current point to 89
save position 90
register commands 89
Register to Point 89
regular expression 83
count occurrences of 85
regular expression search 83
Rename Buffer 70
Rename File 36
repeating a command 11, 23
replace
case sensitivity 87
query 86
regexp 88
string 86
Replace Regexp 88
Replace String 86
replacing 86
replacing commands 76
Report Bug 113
Report Manual Bug 114
RETURN Auto Fill Return 67
RETURN Confirm Parse 103
Return Default 106
RETURN New Line 53
Reverse Incremental Search 77
Reverse Search 79
Revert Buffer 34
revert-buffer-confirm 35
ring
history 103
kill 47, 50, 52
mark 43
undo 55
window 71
Room 114
Rotate Active Finders 124
Rotate Kill Ring 53
Run Command 109
S
same-line-p 177
Save All Files 27
Save All Files and Exit 29
Save File 27
Save Position 89
Save Region 52
save-all-files-confirm 28
save-excursion 177
screen
refresh 73
scroll button
size 73
Scroll Next Window Down 72
Scroll Next Window Up 72
Scroll Window Down 39
Scroll Window Up 40
scroll
size 73
scroll-overlap 40
search
all buffers 80
backward 79
case sensitivity 82
directory 80
files 81
forward 78
incremental backward 77
incremental forward 76
regexp backward 85
regexp forward 85
regular expression 83
system 81, 82
Search All Buffers 80
Search Files 81
Search Files Matching Patterns 81
Search Files tool 81, 82
Search System 82
search-files function 82
searching 76
searching commands 76
Select Buffer 67
Select Buffer Other Window 68
Select Go Back 47
Select Previous Buffer 68
selection
indent 62
indenting 129
Self Insert 54
Self Overwrite 60
sentence
backward 39
delimiter 9
delimiter definition 8
forward 39
kill backward 51
kill forward 51
mark 45
terminator 9
Set Buffer Output 145
Set Buffer Package 145
Set Comment Column 138
Set External Format 30
Set Fill Column 64
Set Fill Prefix 65
Set Mark 43
Set Prefix Argument 23
Set Variable 106
set-current-mark 176
set-interrupt-keys 163
setup-indent 164
shell command
from editor 109
shell mode 91
Shell Send Eof 111
Shell tool 109, 111
shell-pop-regexp 110
shell-push-regexp 110
Show Documentation for Dspec 144
Show Documentation 144
Show Paths From 128
Show Paths To 127
Show Variable 106
simple-char type 69
Skip Whitespace 42
source finding
active finders list 124
dspec 119
tags 121
tags files 122, 123
source recording 118
*source-found-action* 183
space
delete horizontal 49
just one 49
SPACE Auto Fill Space 66
SPACE Complete Field 103
spaces-for-tab 61
start-line-p 177
Stop Shell Subjob 111
string
count occurrences of 85
insert 182
replace 86
search 76
symbol
apropos 143
browser 143
completion 129, 130, 134
describe 143
Symbol Browser tool 143
Syntax coloring 116
system
compile 153
compile changed definitions 153
describe 133
evaluate changed definitions 149
query replace 87
search 81, 82
System Query Replace 87
System Search 81
T
TAB
for command completion 102
for indentation 61, 129
for symbol completion 129
Tab
for command completion 10, 18
tab
insert 63
width 61
TAB Complete Input 102
TAB Indent 61
TAB Indent Selection or Complete Symbol 129
tag
continue search 123
create buffer 121
editor definition 118
find 121
query replace 123
search 122
visit file 123
Tags Query Replace 123
Tags Search 122
temporary files 33
terminator
sentence 9
text handling concepts 8
Text Mode 91
text mode 91
Toggle Auto Save 31
Toggle Breakpoint 154
Toggle Buffer Read-Only 70
Toggle Count Newlines 73
Toggle Error Catch 148
Tools menu
Preferences 2
Top of Window 41
Trace Definition 125
Trace Definition Inside Definition 125
Trace Function 124
Trace Function Inside Definition 125
tracing functions 124
Transpose Characters 58
Transpose Forms 136
Transpose Lines 59
Transpose Regions 59
Transpose Words 58
transposition commands 58
U
Undefine 155
undefined
buffer 155
command 155
definition 155
region 156
Undefine Buffer 155
Undefine Command 155
Undefine Region 156
Undo 56
undo ring 55
size 56
undoing editor commands 13, 55
undo-ring-size 56
Unexpand Last Word 97
Un-Kill 52
Untrace Definition 125
Untrace Function 125
Up Comment Line 140
Uppercase Region 57
Uppercase Word 57
use-buffer 173
V
variable
change value of 106
description 21, 22
text handling with apropos 19
show value of 106
variable functions 188
variables
*buffer-list* 172
*find-likely-function-ignores* 183
*indent-with-tabs* 183
*source-found-action* 183
indenting 183
variable-value 190
variable-value-if-bound 191
View Page Directory 75
View Source Search 120
Visit File 26
Visit Other Tags File 124
Visit Tags File 123
W
Walk Form 136
Wfind File 26
What Command 20
What Cursor Position 42
What Line 38
What Lossage 22
Where Is 22
Where Is Point 42
whitespace
Index

skip 42
window
  delete 72
  delete next 72
  editor definition 5
  mode line 73
  move line to top of 41
  move to bottom 41
  move to top 41
  new 71
  next 71
  previous 72
  scroll down 39
  scroll next down 72
  scroll next up 72
  scroll overlap 40
  scroll up 40
  scroller 73
window commands 71
window functions 191
window ring 71
window-buffer 173
windows
  and the Editor 112
  copy 112
  paste 112
window-text-pane 192
with-buffer-locked 169
with-point 178
with-point-locked 169, 170
word
  backward 37
  capitalize 57
  count for region 45
  dynamic completion 55
  editor definition 8
  forward 37
  kill next 50
  kill previous 51
  lowercase 56
  mark 44
  transposition 58
  uppercase 57
Word Abbrev Apropos 98
Word Abbrev Prefix Point 97
word-offset 184
Write File 28
Write Region 28
Write Word Abbrev File 99

Y
yank 52