LispWorks® for UNIX Editor User Guide

Version 4.1



Copyright and Trademarks

Version 4.1 November 1998

Part number: 3LBDT2A15NE

Copyright © 1994-1998 by Harlequin Group plc.

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Harlequin Group plc.

The information in this publication is provided for information only and is subject to change without notice. Harlequin Group plc and its affiliates assume no responsibility or liability for any loss or damage that may arise from the use of any information in this publication. The software described in this book is furnished under license and may only be used or copied in accordance with the terms of that license.

LispWorks is a registered trademark of Harlequin Group plc. Harlequin, Delivery, Transducer/PC, The Authoring Book, ClassWorks, and KnowledgeWorks are all trademarks of Harlequin Group plc.

Adobe and PostScript are registered trademarks of Adobe Systems Incorporated. Other brand or product names are the registered trademarks or trademarks of their respective holders.

The code for walker.lisp and compute-combination-points is excerpted with permission from PCL, Copyright © 1985, 1986, 1987, 1988 Xerox Corporation.

CLX and CLUE bear the following copyright notice, which applies to the parts of LispWorks derived therefrom:

Texas Instruments Incorporated, P.O. Box 149149, MS 2151, Austin, Texas 78714-9149

Copyright © 1987, 1988, 1989, 1990, 1991 Texas Instruments Incorporated.

Permission is granted to any individual or institution to use, modify and distribute this software, provided that this complete copyright and permission notice is maintained, intact, in all copies and documentation. Texas Instruments Incorporated provides this software "as is" without express or implied warranty.

The XP Pretty Printer bears the following copyright notice, which applies to the parts of LispWorks derived therefrom:

Copyright © 1989 by the Massachusetts Institute of Technology, Cambridge, Massachusetts.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that this copyright and permission notice appear in all copies and supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T. makes no representation about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty. M.I.T. disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall M.I.T. be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

US Government Use

The LispWorks Software is a computer software program developed at private expense and is subject to the following Restricted Rights Legend: "Use, duplication, or disclosure by the United States Government is subject to restrictions as set forth in (i) FAR 52.227-14 Alt III or (ii) FAR 52.227-19, as applicable. Use by agencies of the Department of Defense (DOD) is subject to Harlequin's customary commercial license as contained in the accompanying license agreement, in accordance with DFAR 227.7202-1(a). For purposes of the FAR, the Software shall be deemed to be `unpublished' and licensed with disclosure prohibitions, rights reserved under the copyright laws of the United States. Harlequin Incorporated, One Cambridge Center, Cambridge, Massachusetts 02142."

http://www.harlequin.com/

Europe:

Harlequin Limited Barrington Hall Barrington Cambridge CB2 5RG UK

telephone +44 1223 873 800 fax +44 1223 873 873

North America:

Harlequin Incorporated One Cambridge Center Cambridge, MA 02142 USA

telephone +1 617 374 2400 fax +1 617 252 6505

Asia Pacific:

Harlequin Australia Pty. Ltd. Level 12 12 Moore Street Canberra, ACT 2601 Australia

telephone +61 2 6206 5522 fax +61 2 6206 5525

Contents

Introduction 1

General Concepts

Window layout 5 Buffer locations 7

Text handling concepts 9

Modes 8

1

2

3

Executing commands 9 Basic editing commands 11
Command Reference 15
Aborting commands and processes 16 Executing commands 16 Help 17
Prefix arguments 22 File handling 23 Movement 32
Marks and regions 38 Deleting and killing text 41
Inserting text 45 Undoing 47
Case conversion 48 Transposition 50

Using the editor within LispWorks 2

5

Overwriting 51 Indentation 52 Filling 55 Buffers 59 Windows 62 Pages 64 Searching and replacing 66 Registers 76 Modes 77 Abbreviations 81 Keyboard macros 87 Echo area operations 89 Editor variables 92 Recursive editing 93 Key bindings 93 Running shell commands from the editor 95 Buffers, windows and the mouse 98 Miscellaneous 99

4 Editing Lisp Programs 101

Automatic entry into lisp mode 102
Functions and definitions 102
Forms 112
Lists 115
Comments 117
Parentheses 119
Symbol Documentation 120
Evaluation and compilation 121

5 Advanced Features 127

Customizing default key bindings 128 Customizing Lisp indentation 129 Programming the editor 130

Glossary 147

Index 157

1

Introduction

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

The LispWorks editor has the following features:

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

1.1 Using the editor within LispWorks

The LispWorks editor is fully integrated into the LispWorks programming environment. To produce an editor window, select **Tools > Editor** from the LispWorks podium, or **Works > Tools > Editor** from any other tool.

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 User Guide*.

1.1.1 About this manual

The *Editor User Guide* is divided into four main 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, "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 12). 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 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).

Introduction

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 typed in) or contains text from a file to be edited.

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

- a *major mode* (such as Fundamental or Lisp). 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 *character position indicator*. This value indicates the position in the buffer of the first character displayed in the pane, expressed as a percentage of the total number of characters in the buffer. A value of 10 would indicate that the first character in the pane is 10% of the way through the buffer.
- a *buffer position indicator*. This value indicates the percentage of the buffer that is currently displayed, and is separated from the character position indicator by a +. A value of 20 would indicate that a total of 20% of the buffer is currently shown.
- the pathname with which the buffer is associated.

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.

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 predefined 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 (i.e. the letters A-Z and numbers 0-9). 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 Meta

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

The two special keys referred to are the *Control* (Ctrl) key and the *Meta* key . As some keyboards do not have an *Meta* key, the *Escape* (ESC) key can be used

instead. Note that Esc must be typed *before* pressing the required character key, and not held down.

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 <code>ctrl+x</code> <code>ctrl+s</code>. Another multi-key sequence is <code>ctrl+x</code> s which saves all buffers to their relevant files. Note that in this case no Control character is required to be held down while pressing s.

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

There is a command for which you cannot use Escape instead of Meta. This is the command that breaks a process: Meta+Ctrl+C to break the current process. As there are so many different types of keyboard, if it is not possible to assert which is the Meta key on your keyboard, it may be marked with a special character, such as a diamond, or it may be one of the function keys — try fll. From this point on we refer exclusively to the Meta key in this manual.

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 to execute but are in effect only short hand versions of longer commands.

Most editor commands can be invoked explicitly by using their full names. For example, in the previous section we met the command <code>ctrl+A</code> which moves the current point to the beginning of the line. This command is called a *key binding* and is only a shorter way of executing the command <code>Beginning</code> of <code>Line</code>. To execute this command by name the user must type in <code>Meta+x</code> followed by the

full command name (Meta+x itself is only a default 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 as far as possible when Tab is used, and if the user is not sure of the rest of the command, then pressing Tab again provides a list of possible completions. The relevant 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, 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.

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

Abort the current command which may either be running or just partially typed in

2.6.2 File handling

See "File handling" on page 23.

Ctrl+X Ctrl+F file

Load file into a buffer ready for editing. If the name of a non-existent file is given, then an empty buffer is created in to which text can be inserted. Only when a save is done will the file be created.

Ctrl+x Ctrl+s Save the contents of the current buffer to the associated file. If there is no associated file, one is created with the same name as the buffer

2.6.3 Inserting text

See "Inserting text" on page 45

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

To insert a newline press Return.

2.6.4 Movement

See "Movement" on page 32.

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

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

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

Meta+Shift+< Move to the beginning of the buffer.

Meta+Shift+> Move to the end of the buffer.

2.6.5 Deleting and killing text

See "Deleting and killing text" on page 41.

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 blank line), type Ctrl+K twice

at the start of the line.

2.6.6 Undoing

See "Undoing" on page 47.

ctrl+shift+_ Undo the previous command. If ctrl+shift+_ is typed

repeatedly, previously executed commands are undone in

a "last executed, first undone" order.

2.6.7 Killing and Yanking

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

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

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

Ctrl+space Set a mark for a region. See "Marks and regions" on page

38.

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 41.

Meta+W Copy the region between the mark and the current point.

See "Deleting and killing text" on page 41.

Ctrl+Y Insert (yank) a copied region before the current point. See

"Inserting text" on page 45.

2.6.8 Help

See "Help" on page 17.

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+н к key Describe the command bound to key.

3

Command Reference

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

Commands are grouped according to functionality as follows:

- "Aborting commands and processes"
- "Executing commands"
- "Help"
- "Prefix arguments"
- "File handling"
- "Movement"
- "Marks and regions"
- "Deleting and killing text"
- "Inserting text"
- "Undoing"
- "Case conversion"

- "Transposition"
- "Overwriting"
- "Indentation"
- "Filling"

3.1 Aborting commands and processes

Abort Current Command

Key Sequence

Ctrl+G

Aborts the current command. Ctrl+G can either be used to abandon a command which has been partially typed in, or to abort the command which is currently running.

None Key Sequence

Meta+Ctrl+C

Breaks the current process. Note that this is one command where you cannot use the Escape key in place of the Meta key. As there are so many different types of keyboard, if it is not possible to assert which is the Meta key on your keyboard, it may be marked with a special character, such as a diamond, or it may be one of the function keys — try F11. Note that the process browser, documented in the *Common LispWorks User Guide* can be used to break any specified process.

None

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 95.

Extended Command

Editor Command

Meta+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 таь) 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 "Symbol Documentation" on page 120).

3.3.1 The help command

Help Editor Command

Options: See below

Key sequence: Ctrl+н option

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

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

Display a list of help options.

q or n Quit help.

a string Display a list of commands whose names con-

tain string.

c key Display the command to which key is bound.

a *command* Describe the editor *command*.

Ctrl+D command

Bring up the on-line version of this manual for

command.

g *object* Invoke the appropriate describe *object* com-

mand.

k key Describe the command to which *key* is bound.

Ctrl+k key Bring up the on-line version of this manual for

key.

describe the last 60 keys typed.

v variable Describe variable and show its current value.

ctrl+v variable Bring up the on-line version of this manual for

variable.

w command Display the key sequence to which command is

bound.

Apropos Editor Command

Arguments: string

Key sequence: ctrl+н а string

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

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

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

What Command Editor Command

Arguments: key

Key sequence: Ctrl+н с 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.

Describe Command Editor Command

Arguments: command

Key sequence: Ctrl+н D command

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

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

Document Command

Editor Command

Arguments: command

Key sequence: Ctrl+н Ctrl+D command

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

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

Generic Describe Editor Command

Arguments: object

Key sequence: Ctrl+н 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 Tab can be used to provide a completion list.

Describe Key Editor Command

Arguments: *key*

Key sequence: ctrl+н к key

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

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

Document Key Editor Command

Arguments: key

Key sequence: Ctrl+H Ctrl+K key

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

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

What Lossage Editor Command

Arguments: None

Key sequence: ctrl+н L

Displays the last 60 keys typed.

Describe Editor Variable

Editor Command

Arguments: variable

Key sequence: ctrl+н v variable

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

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

Document Variable

Editor Command

Arguments: variable

Key sequence: Ctrl+H Ctrl+V variable

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

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

Where Is Editor Command

Arguments: command

Key sequence: Сtrl+н w command

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

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

3.3.2 Other help commands

Manual Entry Editor Command

Arguments: *unix-command* Key sequence: None

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

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

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 (Ctrl+v) followed by an integer. Negative prefix arguments are allowed. A prefix argument between 0 and 9 can also be supplied using Meta+digit.

Set Prefix Argument

Editor Command

Arguments: [integer]

Key sequence: Ctrl+U [integer]

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

If no integer is given, the prefix argument defaults to the value of Prefix-Argument-Default.

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

Prefix-Argument-Default

Editor Variable

Default value: 4

The default value for the prefix argument if no integer is provided for set Prefix Argument.

None Key Sequence

Key sequence: Meta+<0-9>

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

Negative Argument

Editor Command

Arguments: None Key sequence: None

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

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

3.5 File handling

This section contains details of commands used for file handling.

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

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

There are many file handling commands which cannot be pigeon-holed so neatly and these are found in the section "Miscellaneous file operations" on page 30. 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

Creates 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) 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.

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: Ctrl+x Ctrl+v pathname

editor:visit-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.

3.5.2 Saving files

Save File Editor Command

Arguments: None

Key sequence: Ctrl+x Ctrl+s

editor:save-file-command p &optional buffer

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

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

Save All Files Editor Command

Arguments: None

Key sequence: Ctrl+x s

A Save Buffers window 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.

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.

Backup File Editor Command

Arguments: *pathname* Key sequence: None

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

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

Save All Files and Exit

Editor Command

Arguments: None

Key sequence: Ctrl+x Ctrl+C

A Save Buffer window 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 is exited, although confirmation is asked for first.

Add-Newline-at-EOF-on-Writing-File

Editor Variable

Default value: :ask-user

If the value of this variable is t, a newline is automatically added to the end of any file that lacks one when the file is being saved. If its value is nil, no newline is added in this way.

Its default value is :ask-user, so the user is prompted about adding the newline when necessary.

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 backup being the name of the original file enclosed within # characters.

Auto Save Toggle

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 Auto Save Toggle disregards the current state of auto-save.

Auto-Save-Filename-Pattern

Editor Variable

Default value: "~A#~A#"

This control string is used with the Common Lisp format function to create the filename of the auto-save file. Format is called with two arguments, the first being the directory namestring and the second being the file namestring of the pathname associated with the buffer.

By default the auto-save file is created in the same directory as the file for which it is a backup, and it has the same name with surrounding # characters.

Auto-Save-Key-Count-Threshold

Editor Variable

Default value: 256

Determines 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

Determines the time interval (in seconds) after which all modified buffers are auto-saved. If the value is zero or negative, this feature is turned off.

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, its name being the same as that file followed by a ~ character.

Backups-Wanted

Editor Variable

Default value: t

If the value of this variable is t, backups are automatically made on file saving. If its value is nil, no backups are made.

Backup-Filename-Suffix

Editor Variable

Default value: #\~

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

Backup-Filename-Pattern

Editor Variable

Default value: "~A~A~A"

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

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

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

3.5.5 Miscellaneous file operations

Print File Editor Command

Arguments: file

Key sequence: None

Prints *file*, using the print command specified by the variable lisp-works:*print-command*.

Revert File Editor Command

Arguments: None Key sequence: None

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

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

If the buffer has been modified and the value of the variable Revert-File-Confirm is t then Revert File asks for confirmation before reverting to a previous state.

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

Revert-File-Confirm Editor Variable

Default value: t

When the command Revert File is invoked, if the value of this variable is t and the buffer has been modified then confirmation is asked for before

the revert operation is performed. If its value is nil, no confirmation is asked for.

Process File Options

Editor Command

Arguments: None Key sequence: None

Reprocess the file options associated with the current buffer. This involves resetting all the modes to be the ones specified either by the modeline at the top of the file or, if this does not exist, by the name of the file. See "Modes" on page 77.

Insert File Editor Command

Arguments: pathname

Key sequence: Ctrl+x I pathname

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

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

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

Delete File Editor Command

Arguments: *pathname* Key sequence: None

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

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.

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 prefix argument repeats these commands a certain number of times in the opposite logical direction. For example, the command <code>ctrl+U 10 Ctrl+B</code> moves the cursor 10 characters backwards, but the command <code>ctrl+U -10 Ctrl+B</code> moves the cursor 10 characters forward.

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

Forward Character Editor Command

Arguments: None

Key sequence: Ctrl+F or Right Arrow on some keyboards

Moves the current point forward one character.

Backward Character Editor Command

Arguments: None

Key sequence: Ctrl+B or Left Arrow on some keyboards

Moves the current point backward one character.

Forward Word Editor Command

Arguments: None Key sequence: Meta+F

Moves the current point forward one word.

Backward Word Editor Command

Arguments: None Key sequence: Meta+B

Moves the current point backward one word.

Beginning of Line

Editor Command

Arguments: None Key sequence: Ctrl+A

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

End of Line Editor Command

Arguments: None Key sequence: Ctrl+E

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

Next Line Editor Command

Arguments: None

Key sequence: ctrl+n or Down 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*.

Forward Sentence

Editor Command

Arguments: None Key sequence: Meta+E

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

Backward Sentence

Editor Command

Arguments: None Key sequence: Meta+A

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

Forward Paragraph

Editor Command

Arguments: None Key sequence: Meta+]

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

Backward Paragraph

Editor Command

Arguments: None Key sequence: Meta+[Moves the current point to the start of the current paragraph. If the current point is already at the start of a paragraph, then it is moved to the beginning of the previous paragraph.

Scroll Window Down

Editor Command

Arguments: None Key sequence: Ctrl+v

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

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

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

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

Scroll Window Up

Editor Command

Arguments: None Key sequence: Meta+v

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

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

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

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

Scroll-Overlap Editor Variable

Default value: 2

Determines the number of lines of overlap when scroll window pown and scroll window up are used with no prefix argument.

Line to Top of Window

Editor Command

Arguments: None Key sequence: None

Moves the current line to the top of the window.

Top of Window

Editor Command

Arguments: None Key sequence: None

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

Bottom of Window

Editor Command

Arguments: None Key sequence: None

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

Beginning of Buffer

Editor Command

Arguments: None

Key sequence: Meta+Shift+<

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

End of Buffer Editor Command

Arguments: None

Key sequence: Meta+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:

editor:what-cursor-position-command.

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 23 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 98—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 Editor Command

Arguments: None

Key sequence: Ctrl+Space or Middle Mouse Button

Sets the mark to the current point, in effect, pushing the current point on to the top of the mark ring.

With a prefix argument equal to the value of the Prefix-Argument-Default, Pop and Goto Mark is invoked (a prefix argument equal to the Prefix-Argument-Default can be achieved by typing Ctrl+U with no following integer).

With a prefix argument equal to the square of the Prefix-Argument-Default (achieved by typing Ctrl+U Ctrl+U before invoking Set Mark), the mark is deleted from the mark ring, and the next point on the mark ring then becomes the mark, but the current point is not moved.

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, this mark is deleted from the mark ring. Successive invocations of this command clear the mark ring.

Pop Mark Editor Command

Arguments: None

Key sequence: Meta+Ctrl+Space

Pops the top mark from the mark ring, de-activating the current region.

Exchange Point and Mark

Editor Command

Arguments: None

Key sequence: Ctrl+X Ctrl+X

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

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

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

Mark 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: мета+н

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

Mark Whole Buffer Editor Command

Arguments: None

Key sequence: ctrl+х н

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 than Region-Query-size lines, 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 the print command specified by the variable lispworks:*print-command*.

3.8 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 a 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.

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

3.8.1 Deleting Text

Delete Next Character

Editor Command

Arguments: None Key sequence: 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.

Delete Horizontal Space

Editor Command

Arguments: None Key sequence: Meta+\

Deletes all spaces on the line surrounding the current point.

Just One Space

Editor Command

Arguments: None

Key sequence: Meta+Space

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

Delete Blank Lines

Editor Command

Arguments: None

Key sequence: Ctrl+Shift+X Ctrl+Shift+O

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

Delete Region Editor Command

Arguments: None Key sequence: None

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

3.8.2 Killing text

All 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).

Kill Next Word Editor Command

Arguments: None Key sequence: Meta+D

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

Kill Previous Word Editor Command

Arguments: None

Key sequence: Meta+Delete

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

Kill Line Editor Command

Arguments: None Key sequence: Ctrl+k

Kills the characters from the current point up to the end of the current line.

If the line is empty then the line is deleted.

Backward Kill Line

Editor Command

Arguments: None Key sequence: None

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

Forward Kill Sentence

Editor Command

Arguments: None Key sequence: Meta+K

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

Backward Kill Sentence

Editor Command

Arguments: None

Key sequence: Ctrl+x Delete

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

Kill Region Editor Command

Arguments: None Key sequence: Ctrl+w Kills the region between the current point and the mark.

Save Region Editor Command

Arguments: None Key sequence: Meta+W

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

3.9 Inserting text

This section contains details of commands used to insert text from the kill ring—see "Deleting and killing text" on page 41—and various other commands used to insert text and lines into the buffer.

Un-Kill Editor Command

Arguments: None Key sequence: Ctrl+Y

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

A prefix argument (Ctrl+v 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: Meta+Y

Replaces the text that has just been un-killed with the item that is next on the kill ring. It is therefore possible to recover text other than that which was most recently killed by typing Ctrl+Y followed by Meta+Y the

required number of times. If un-Kill was not the previous command, an error is 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.

Open Line Editor Command

Arguments: None Key sequence: Ctrl+0

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: key

Key sequence: Ctrl+Q key

Inserts key 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 ^].

Self Insert Editor Command

Arguments: None Key sequence: key

editor:self-insert-command p &optional char

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

Dynamic Completion

Editor Command

Arguments: None Key sequence: Meta+/

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

Expand File Name

Editor Command

Arguments: None

Key sequence: Meta+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.

3.10 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.

3.11 Case conversion

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

Lowercase Word Editor Command

Arguments: None Key sequence: Meta+L

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

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

Uppercase Word Editor Command

Arguments: None Key sequence: Meta+U

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

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

Capitalize Word Editor Command

Arguments: None Key sequence: Meta+C

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

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

Lowercase Region

Editor Command

Arguments: None

Key sequence: Ctrl+X Ctrl+L

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

Uppercase Region

Editor Command

Arguments: None

Key sequence: Ctrl+X Ctrl+U

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

Capitalize Region

Editor Command

Arguments: None Key sequence: None

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

3.12 Transposition

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

Transpose Characters

Editor Command

Arguments: None Key sequence: Ctrl+T

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

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

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

Transpose Words

Editor Command

Arguments: None Key sequence: Meta+T

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

A positive prefix argument causes the current or previous word to be shifted forwards the required number of words. A negative prefix argu-

ment 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.

3.13 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.

3.14 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, a single tab is inserted at the current point which is spaces-For-Tab wide.

In Lisp mode, the current line is indented according to the structure of the current Lisp form.

In both cases, a prefix argument causes a single tab to be inserted at the start of the appropriate number of lines (starting from the current line).

Spaces-For-Tab Editor Variable

Default value: 8

Determines the width of the whitespace used for tabs.

Indent Region Editor Command

Arguments: None

Key sequence: Meta+Ctrl+\

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

In Text mode a block of whitespace, which is spaces-For-Tab wide, is inserted at the start of each line within the region.

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

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

Indent Rigidly Editor Command

Arguments: None

Key sequence: Ctrl+X Tab or Ctrl+X Ctrl+I

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

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

Delete Indentation Editor Command

Arguments: None

Key sequence: Meta+Shift+^

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

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

Back to Indentation Editor Command

Arguments: None Key sequence: меtа+м

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.

3.15 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.

3.15.1 Fill commands

Fill Paragraph Editor Command

Arguments: None Key sequence: Meta+Q

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

A prefix argument causes Fill-Column to be set at the required value for the current operation.

Fill Region Editor Command

Arguments: None Key sequence: Meta+G

Fills the region from the current point to the mark.

A prefix argument causes Fill-Column to be set at the required value for the current operation.

Fill-Column Editor Variable

Default value: 79

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

Set Fill Column Editor Command

Arguments: None

Key sequence: Ctrl+x F

Sets the value of Fill-Column, for the current buffer, as the column of the current point.

A prefix argument causes Fill-column to be set at the required value.

Fill-Prefix Editor Variable

Default value: nil

Defines a string which is excluded when each line of the current buffer is re-formatted using the filling commands. For example, if the Fill-Prefix is set to ;;, then these characters at the start of a line is skipped over when the text is re-formatted, so it is possible to re-format Lisp comments. If the Fill-Prefix is nil, no characters are excluded when text is filled.

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

Set Fill Prefix Editor Command

Arguments: None Key sequence: Ctrl+x .

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

Center Line Editor Command

Arguments: None Key sequence: None

Centers the current line with reference to the current value of Fill-Column.

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

3.15.2 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.

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.

3.16 Buffers

This section contains details of commands used to manipulate buffers.

Select Buffer Editor Command

Arguments: buffer-name

Key sequence: ctrl+х в buffer-name

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

Select Buffer Other Window

Editor Command

Arguments: *buffer-name* Key sequence: None

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

Select Previous Buffer

Editor Command

Arguments: None

Key sequence: Meta+Ctrl+L

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

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

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.

Insert Buffer Editor Command

Arguments: *buffer-name* Key sequence: None

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

Rename Buffer Editor Command

Arguments: *new-name* Key sequence: None

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

Print Buffer Editor Command

Arguments: None Key sequence: None

Prints the current buffer, using the print command specified by the variable lispworks:*print-command*.

Set Buffer Read-Only

Editor Command

Arguments: None

Key sequence: Ctrl+x Ctrl+Q

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

Check Buffer Modified

Editor Command

Arguments: None

Key sequence: Ctrl+x shift+~

Checks whether the current buffer is modified or not.

Buffer Not Modified Editor Command

Arguments: None

Key sequence: Meta+Shift+~

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

Makes the current buffer not modified.

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

3.17 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.

Refresh Screen Editor Command

Arguments: None Key sequence: Ctrl+L

Moves the current line to the center of the current window, and then redisplays 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.

3.18 Pages

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

Previous Page Editor Command

Arguments: None

Key sequence: Ctrl+x [

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

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

Next Page Editor Command

Arguments: None

Key sequence: Ctrl+x]

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

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

Goto Page Editor Command

Arguments: None Key sequence: None

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

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

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.

3.19 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. The third part provides details of commands used for replacing one string with another.

3.19.1 Searching

Most of the search commands perform straight forward 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.

Ctrl+S	If the search string is empty, repeats the last
	incremental search, otherwise repeats a for-
	ward search for the current search string.
	If the search string cannot be found starts the

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.

If the search string is empty, invokes a nonincremental search, otherwise exits the search,

leaving the current point at the last find.

Ctrl+G Aborts the search, returning the current point

to its original location.

If the search string cannot be found, cancels the last character typed (equivalent to

Delete).

Ctrl+Q Quotes the next character typed.

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.

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

Quotes the next character typed.

If the search string is amnty repeats the last

Forward Search Editor Command

Delete).

Arguments: string

Key sequence: Ctrl+s Esc string

Ctrl+Q

C+~1 +D

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

Searches forwards from the current point for *string*. When a match is found, the current 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.

The argument *the-point* is the point from which to start the search. The default is the current point.

Reverse Search Editor Command

Arguments: string

Key sequence: Ctrl+R Esc string

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

Searches backwards from the current point for *string*. When a match is found, the current 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.

the-point is the point from which to start the search. The default is the current point.

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*. Only files with the suffix .lisp or .lsp are searched. 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: *ignorable-file-suffices*. Use the key sequence Meta+, to find subsequent definitions of the search string.

System Search

Editor Command

Arguments: *system string* Key sequence: None

Searches *system* for *string*. Use the key sequence Meta+, to find subsequent definitions of the search string.

Default-Search-Kind

Editor Variable

Default value: :string-insensitive

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

Count Occurrences string

Editor Command

Arguments: None Default binding: None

editor:count-occurrences-command p &optional string

Counts the number of occurrences of string.

It is also possible to search a set of files programatically using the seach-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.

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 cur-

sor on the occurrence. Meta+-, makes the search continue until the next occurrence.

3.19.2 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 new-
	line. For example, c.r matches any three char-
	acter 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.
- 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.
- ? Matches the previous regexp either 0 or 1 times. For example, ca?r matches either the string cr or car, and nothing else.
- ^ Matches the next regexp as long as it is at the beginning of a line. For example, ^foo matches the string foo as long as it is at the beginning of a line.

\$ Matches the previous regexp as long as it is at
the end of a line. For example, £00\$ matches
the string £00 as long as it is at the end of a
line.

Contains a character set to be used for matching, where the other special characters mentioned do not apply. The empty string is automatically part of the character set. For example, [a.b] matches either a or . or b or the empty string. The regexp c[ad]*r matches strings beginning with c and ending with r, with any number of a's and d's in-between.

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

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

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

\(, \) Provides a grouping construct. For example, ab\(cd\|ef\) matches either abcd or abef.

Regexp Forward Search

Editor Command

Arguments: string

[]

Key sequence: Meta+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: Meta+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.

3.19.3 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.

Query Replace Editor Command

Arguments: target replacement

Key sequence: Meta+Shift+% target replacement

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

Replaces occurrences of *target* string by *replacement* string, starting from the current point, but only after querying the user. Each time *target* is found, an action must be indicated from the keyboard.

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

The following key sequences are used to control Query Replace:

Space Of y	Replace <i>target</i> by <i>replacement</i> and move to the next occurrence of <i>target</i> .
Delete	Skip <i>target</i> without replacing it and move to the next occurrence of <i>target</i> .
•	Replace target by replacement and then exit.
1	Replace all subsequent occurrences of <i>target</i> by <i>replacement</i> without prompting.
Ctrl+R	Enter recursive edit. This allows the current occurrence of <i>target</i> to be edited. When this editing is completed, <code>Exit Recursive Edit</code> should be invoked. The next instance of <i>target</i> 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: *ignorable-file-suffices*. Each time target is found, an action must be indicated from the keyboard. For details of possible actions see Query Replace.

System Query Replace

Editor Command

Arguments: system target replacement

Key sequence: None

Replaces occurrences of *target* string by *replacement* string, for each file in *system*, but only after querying the user. Each time *target* is found, an action must be indicated from the keyboard. For details of possible actions see Query Replace.

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.

3.20 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.

Save Position 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.

Jump to Saved Position

Editor Command

Arguments: name

Key sequence: Ctrl+x J name

Moves the current point to a location previously saved in the register called *name*.

Kill Register Editor Command

Arguments: *name*Key sequence: None

Kills the register called name.

List Registers Editor Command

Arguments: None Key sequence: None

Lists all existing registers.

Put Register Editor Command

Arguments: name

Key sequence: Ctrl+x x name

Saves the region between the mark and the current point to the register called *name*.

Get Register Editor Command

Arguments: name

Key sequence: Ctrl+x G name

Copies the region from the register called *name* to the current point.

3.21 Modes

A buffer can be in two kinds of mode at once: *major* and *minor*. The following two sections give a description of each, along with details of some commands which alter the modes.

In most cases, the current buffer can be put in a certain mode using the mode name as an Editor Command.

3.21.1 Major modes

The major modes govern how certain commands behave. 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 3-52.

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.
- Lisp mode. Used for editing Lisp programs and is automatically loaded if the file name ends in .lisp or .lsp.
- *Shell mode.* Used for running interactive shells.

The major mode of most buffers may be altered explicitly by using the commands described below.

By default, Lisp mode is the major mode whenever you edit a file with a .lisp or .lsp extension. If you have files of source code with extensions other than .lisp or .lsp, however, put the following code in your .lispworks file, substituting the extensions shown for those of your own files:

```
(define-file-type-hook
  ("lispworks" "lisp" "slisp" "el" "lsp" "mcl" "cl")
  (buffer type)
  (declare (ignore type))
  (setf (buffer-major-mode buffer) "Lisp"))
```

This automatically makes Lisp mode the major mode if you edit any file with a .lisp, .slisp, .el, .lsp, .mcl, or .cl extension.

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.

3.21.2 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 51.
- 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 55.
- Abbrev mode. Allows abbreviation definitions to be expanded automatically—see "Abbreviations" on page 81.
- Execute mode. Used by the Listener to make history commands available (see the Common LispWorks User Guide).

3.21.3 Defining modes

New modes can be defined using the defmode macro.

defmode Function

Summary Defines new editor modes.

Package editor

Signature defmode name 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 variable definitions.

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 mil.

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 5, "Customizing default key bindings", 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 of dotted pairs, where the first element of the pair is the (string) name of the variable to be changed, and the second is the new value.

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. First, the defmode call itself:

```
(defmode "Foo" :setup-function 'setup-foo-mode)
```

The next piece of code makes .foo files invoke Foo-mode:

```
(define-file-type-hook ("foo") (buffer type)
  (declare (ignore type))
  (setf (buffer-minor-mode buffer "Foo") t))
```

The next line defines the set-up function:

Now, any files loaded into the Editor with the suffix .foo invoke the Foo minor mode.

3.22 Abbreviations

Abbreviations (*abbrevs*) can be defined by the user, such that if an abbreviation is typed at the keyboard followed by a word terminating character (such as space or ,), the expansion is found and used to replace the abbreviation. Typing can thereby be saved for frequently used words or sequences of characters.

There are two kinds of abbreviations: *global abbreviations*, which are expanded in all major modes; and *mode abbreviations*, which are expanded only in defined major modes.

Abbreviations (both global and mode) are only expanded automatically when *Abbrev mode* (a minor mode) is on. The default is for abbrev mode to be off.

All abbreviations that are defined can be saved in a file and reloaded during later editor sessions.

Abbrev Mode Editor Command

Arguments: None Key sequence: None

Switches abbrev mode on if it is currently off, and off if it is currently on. Only when in abbrev mode are abbreviations automatically expanded.

Add Mode Word Abbrev

Editor Command

Arguments: abbrev

Key sequence: Ctrl+x Ctrl+A abbrev

Defines a mode abbreviation for the word before the current point.

A positive prefix argument defines an abbreviation for the appropriate number of words before the current point. A zero prefix argument defines an abbreviation for all the text in the region between the mark and the current point. A negative prefix argument deletes an abbreviation.

Inverse Add Mode Word Abbrev

Editor Command

Arguments: expansion

Key sequence: Ctrl+x Ctrl+H expansion

Defines the word before the current point as a mode abbreviation for *expansion*.

Add Global Word Abbrev

Editor Command

Arguments: abbrev

Key sequence: Ctrl+x + abbrev

Defines a global abbreviation for the word before the current point.

A positive prefix argument defines an abbreviation for the appropriate number of words before the current point. A zero prefix argument defines an abbreviation for all the text in the region between the mark and the current point. A negative prefix argument deletes an abbreviation.

Inverse Add Global Word Abbrev

Editor Command

Arguments: expansion

Key sequence: Ctrl+x - expansion

Defines the word before the current point as a global abbreviation for *expansion*.

Make Word Abbrev

Editor Command

Arguments: abbrev expansion mode

Key sequence: None

editor:make-word-abbrev-command \ensuremath{p} &optional \ensuremath{abbrev} expansion \ensuremath{mode}

Defines an abbreviation for *expansion* without reference to the current point. The default value for *mode* is global.

Abbrev Expand Only

Editor Command

Arguments: None Key sequence: None

Expands the word before the current point into its abbreviation definition (if it has one). If the buffer is currently in abbrev mode then this is done automatically on meeting a word defining an abbreviation.

Word Abbrev Prefix Point

Editor Command

Arguments: None Key sequence: Meta+'

Allows the prefix before the current point to be attached to the following abbreviation. For example, if the abbreviation value is bound to valuation, typing re followed by Meta+', followed by value, results in the expansion revaluation.

Unexpand Last Word

Editor Command

Arguments: None Key sequence: None

Undoes the last abbreviation expansion. If this command is typed twice in succession, the previous abbreviation is restored.

Delete Mode Word Abbrev

Editor Command

Arguments: *abbrev*Key sequence: None

editor:delete-mode-word-abbrev-command p &optional abbrev mode

Deletes a mode abbreviation for the current mode. A prefix argument causes all abbreviations defined in the current mode to be deleted.

The argument *mode* is the name of the mode for which the deletion is to be applied. The default is the current mode.

Delete Global Word Abbrev

Editor Command

Arguments: *abbrev* Key sequence: None

editor:delete-global-word-abbrev-command p &optional abbrev

Deletes a global abbreviation. A prefix argument causes all global abbreviations currently defined to be deleted.

Delete All Word Abbrevs

Editor Command

Arguments: None Key sequence: None

Deletes all currently defined abbreviations, both global and mode.

List Word Abbrevs

Editor Command

Arguments: None Key sequence: None

Displays a list of all the currently defined abbreviations in an Abbrev window.

Word Abbrev Apropos

Editor Command

Arguments: *search-string* Key sequence: None

editor:word-abbrev-apropos-command p &optional search-string

Displays a list of all the currently defined abbreviations which contain *search-string* in their abbreviation definition or mode. The list is displayed in an Abbrev window.

Edit Word Abbrevs

Editor Command

Arguments: None Key sequence: None

Allows recursive editing of currently defined abbreviations. The abbreviation definitions are displayed in an Edit Word Abbrevs buffer, from where they can be can be added to, modified, or removed. This buffer can then either be saved to an abbreviations file, or Define Word Abbrevs can be used to define any added or modified abbreviations in the buffer. When editing is complete, Exit Recursive Edit should be invoked.

Write Word Abbrev File

Editor Command

Arguments: *filename* Key sequence: None

editor:write-word-abbrev-file-command p &optional filename

Saves the currently defined abbreviations to *filename*. If no file name is provided, the default file name defined by the editor variable Abbrev-

Pathname-Defaults is used.

Append to Word Abbrev File

Editor Command

Arguments: *filename* Key sequence: None

editor:append-to-word-abbrev-file-command p &optional filename

Appends all abbreviations that have been defined or redefined since the last save to *filename*. If no file name is provided, the default file name defined by the editor variable Abbrev-Pathname-Defaults is used.

Abbrev-Pathname-Defaults

Editor Variable

Default value: abbrev.defns

Defines the default file name for saving the abbreviations that have been defined in the current buffer.

Read Word Abbrev File

Editor Command

Arguments: *filename* Key sequence: None

editor:read-word-abbrev-file-command p & optional filename

Reads previously defined abbreviations from *filename*. The format of each abbreviation must be that used by write word Abbrev File and Insert Word Abbrevs.

Insert Word Abbrevs

Editor Command

Arguments: None Key sequence: None

Inserts into the current buffer, at the current point, a list of all currently defined abbreviations. This is similar to write Word Abbrev File, except that the abbreviations are written into the current buffer rather than a file.

Define Word Abbrevs

Editor Command

Arguments: None Key sequence: None

Defines abbreviations from the definition list in the current buffer. The format of each abbreviation must be that used by write word Abbrev File and Insert Word Abbrevs.

3.23 Keyboard macros

Keyboard macros enable a sequence of commands to be turned into a single operation. For example, if it is found that a particular sequence of commands is to be repeated a large number of times, they can be turned into a keyboard macro, which may then be repeated the required number of times by using Prefix Arguments.

Note that keyboard macros are only available for use during the current editing session.

Define Keyboard Macro

Editor Command

Arguments: None

Key sequence: Ctrl+x Shift+(

Begins the definition of a new keyboard macro. All the commands that are subsequently invoked are executed and at the same time combined into the newly defined macro. Any text typed into the buffer is also included in the macro. The definition is ended with <code>End Keyboard Macro</code>, and the sequence of commands can then be repeated with <code>Last Keyboard Macro</code>.

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.

Exit from this keyboard macro immediately.

3.24 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.

3.24.1 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 to produce a popup list of all possible completions.

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 sequence: ?

Displays a popup list of all possible completions of the text in the echo

area.

3.24.2 Repeating echo area commands

The Echo Area commands are recorded in a history ring so that they can be easily repeated. Details of these commands are given below.

Previous Parse Editor Command

Arguments: None Key sequence: Meta+P

Moves to the previous command in the Echo Area history ring. If the current input is not empty and the contents are different from what is on the top of the ring, then this input is pushed onto the top of the ring before the new input is inserted.

Next Parse Editor Command

Arguments: None Key sequence: Meta+N

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.

3.24.3 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: Meta+B

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

Beginning Of Parse

Editor Command

Arguments: None Key sequence: Ctrl+A

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

3.24.4 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: Meta+Delete

Kills the previous word entered in the Echo Area.

Kill Parse Editor Command

Arguments: None Key sequence: 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+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+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.

3.25 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.

3.26 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 <code>Query Replace</code>, the <code>ctrl+R</code> option can be used to edit the current instance of the target string (that is, enter a recursive edit). Details of commands used to exit a recursive edit are given below.

Exit Recursive Edit Editor Command

Argument: None

Key sequence: Meta+Ctrl+Z

Exits a level of recursive edit, returning to the original command. An error is signalled if not in a recursive edit.

Abort Recursive Edit Editor Command

Argument: None Key sequence: Ctrl+1

Aborts a level of recursive edit, quitting the unfinished command immediately. An error is signalled if not in a recursive edit.

3.27 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 128.

Bind Key Editor Command

Argument: command key-sequence bind-type

Key sequence: None

Binds *command* (full command names must be used) to *key-sequence*. *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".

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.

Delete Key Binding

Editor Command

Argument: key-sequence bind-type

Key sequence: None

Removes a key binding, so that the key sequence no longer invokes any command. The argument *bind-type* can be either buffer, global or mode. If a *bind-type* of buffer or mode is selected, the name of the buffer or mode required must then be entered. The default value for *bind-type* is "global".

It is necessary to enter the kind of binding, because a single key sequence may sometimes be bound differently in different buffers and modes.

Illegal Editor Command

Argument: None Key sequence: None

Signals an editor error with the message "Illegal command in the current mode" accompanied by a beep. It is sometimes useful to bind key sequences to this command, to ensure the key sequence is not otherwise bound.

Do Nothing Editor Command

Argument: None Key sequence: None

Does nothing. This is therefore similar to Illegal, except that there is no

beep and no error message.

3.28 Running shell commands from the editor

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

Shell Command Editor Command

Argument: command

Key sequence: Meta+! command

Executes the single shell command *command*. The output from the command is displayed in a Shell Output buffer. A prefix argument causes the output from the shell command to be sent to the *terminal-io* stream rather than the Shell Output buffer.

Run Command Editor Command

Argument: *command* Key sequence: None

Similar to shell command, but runs the 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-push-RegExp or shell-pop-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 ~programs/lisp is issued, the next time the editor command wfind file is issued, the default directory offered is ~programs/lisp. 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-push-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-pop-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.

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.

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.

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.

3.29 Buffers, windows and the mouse

3.29.1 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 (in UNIX this is usually achieved by clicking the middle mouse button).

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).

3.29.2 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 38—but also note that a region must be defined *either* by using the mouse *or* by using editor key sequences, as the region may become unset if a combination of the two is used. For example, using Ctrl+space to set a mark and then using the mouse to go to the start of the required region unsets the mark.

left-button

Moves the current point to the position of the mouse pointer.

shift-left-button

Moves the current point to the location of the mouse pointer and sets the mark to be the end of the new current form.

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 sets the current mark to the position of the mouse pointer.

right-button

Brings up a popup window, from which a number of useful commands can be invoked. The options include **Cut**, **Copy**, and **Paste**.

shift-right-button

Inserts the text from the location of the mouse pointer to the end of that form at the current point.

3.30 Miscellaneous

Bug Report Editor Command

Argument: None Key sequence: None

Opens a window containing the LispWorks bug reporting template. This template can then be filled in and sent to Harlequin via the electronic mail system.

Room Editor Command

Argument: None Key sequence: None

Displays information on the current status of the memory allocation for

the host computer.

4

Editing Lisp Programs

There are a whole set of editor commands designed to facilitate editing of Lisp programs. These commands are designed to understand the syntax of the Lisp language and therefore allow movement over Lisp constructs, indentation of code, operations on parentheses and definition searching. Lisp code can also be evaluated and compiled directly from the editor.

To use some of these commands the current buffer should be in Lisp mode. See 'Modes' on page 77.

Commands are grouped according to functionality as follows:

- "Functions and definitions"
- "Forms"
- "Lists"
- "Comments"
- "Parentheses"
- "Symbol Documentation"
- "Evaluation and compilation"

4.1 Automatic entry into lisp mode

Some source files begin with the line

```
;; -*- Mode:Common-Lisp;
or the line
;; -*- Mode:Lisp;
```

A buffer is automatically set to be in lisp mode when such a file is displayed.

Alternatively, if you have files of Common Lisp code with extension other than .lisp, add the following code to your .lispworks file, substituting the extensions shown for your own. This ensures that Lisp mode is the major mode whenever a file with one of these extensions is viewed in the editor:

```
(editor:define-file-type-hook
    ("lispworks" "lisp" "slisp" "el" "lsp" "mcl" "cl")
    (buffer type)
    (declare (ignore type))
    (setf (editor:buffer-major-mode buffer) "Lisp"))
```

4.2 Functions and definitions

4.2.1 Movement, marking and indentation

Beginning of Defun

Editor Command

Argument: None

Key sequence: Meta+Ctrl+A

Moves the current point to the beginning of the current top-level form. A positive prefix argument causes the point to be moved to the beginning of the form the required number of levels up.

End of Defun Editor Command

Argument: None

Key sequence: Meta+Ctrl+E

Moves the current point to the end of the current top-level form. A positive prefix argument causes the point to be moved to the end of the form the required number of levels down.

Mark Defun Editor Command

Argument: None

Key sequence: Meta+Ctrl+H

Puts the mark at the end of the current top-level form and the current point at the beginning of the form. The definition thereby becomes the current region. If the current point is initially located between two top-level forms, then the mark and current point are placed around the next top-level form.

Defindent Editor Command

Argument: *no-of-args* Key sequence: None

Defines the number of arguments of the current function or defining form to be specially indented if they fall on a new line. The indent is defined for the first string of the form. So, if defun is the first string of the current form, Defindent defines the argument indentation for all defuns in that buffer.

4.2.2 Definition searching

Definition searching involves taking a function name 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 the LispWorks cross-referencer or by a Tags buffer. If cross-referencing or Tags information has not been produced, then the following commands do not work.

Cross-referencing information is produced by turning source debugging on while compiling and loading the relevant definitions (see toggle-source-debugging in the *LispWorks Reference Manual*).

Tag information is set up by the editor itself, and can be saved to a file for future use. For each file in a defined system, the tag file contains a relevant file name entry, followed by names and positions of each defining form in that file. Before tag searching can take place, there must exist a buffer containing the required tag information. You can specify a previously saved tag file as the current tag buffer, or you can create a new one using Create Tags Buffer. GNU Emacs tag files are fully compatible with LispWorks editor tag files.

Find Source Editor Command

Argument: function [tags-file]

Key sequence: Meta+. function [tags-file]

Tries to find the source code for *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.

If the source code for *function* is found, the file in which it is contained is displayed in a new buffer. When there is more than one definition, Find source finds the first definition, and Meta+, (Continue Tags Search) finds subsequent definitions.

Find source first looks for definitions that have been loaded or evaluated (i.e. using cross-referencing information). If it cannot find the definition here, it looks at the current tags file by issuing the command Find Tag (for this reason, Find source is recommended rather than Find Tag).

If there is no current tags buffer, Find source 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 documentation on Tags commands given below).

See the chapter on the DSPEC package in the *LispWorks Reference Manual* for information on how to use the dpsec:*active-finders* variable to control how this command operates. The file config/a-dot-lispworks.lisp contains an example setting for this variable.

Find Source for Dspec

Editor Command

Argument: function [tags-file]

Key sequence: None

Similar to Find source, but takes a dspec instead of a name as its argument.

For example, given a generic function foo of one argument, with methods specializing on bar and baz,

Find Source foo

will find each method definition in turn (with the continuation via Meta+), whereas

Find Source for Dspec (method foo (bar))

immediately finds the definition of the method on bar.

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) in the Find Definitions view of the Editor.

Create Tags Buffer

Editor Command

Argument: None Key sequence: None

Creates a buffer containing tag search information, for all the .lisp files in the current directory. If you want to use this information at a later date then save this buffer to a file (preferably a file called tags in the current directory).

The format of the information contained in this buffer is compatible with that of GNU Emacs tags files.

A prefix argument causes the user to be prompted for the name of a file containing a list of files, to be used for constructing the tags table.

Find Tag Editor Command

Argument: *string* [*tags-file*] Key sequence: Meta+?

Tries to find the source code for a symbol containing a partial or complete match with *string* by examining the tags contained in the current tags buffer and loading the correct file when a match is found. The symbol under the current point is offered as a default value for *string*.

If the source code for a match with *string* is found, the file in which it is contained is displayed in a new buffer. When there is more than one definition, Find Tag finds the first definition, and Meta+, (Continue Tags Search) finds subsequent definitions.

If there is no current tags buffer, 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 Reference Manual* for information on how to use the dpsec:*active-finders* variable to control how this command operates. The file config/a-do-lispworks.lisp contains an example setting for this variable.

See also Find source and Find source for Dspec, which find the source code for a function.

Tags Search Editor Command

Argument: string [tags-file] Key sequence: None

Exhaustively searches each file mentioned in the current tags buffer for string. If string is found, it is displayed in a new buffer containing the relevant file. When there is more than one definition, Tags Search finds the first definition, and Meta+, (Continue Tags Search) finds subsequent definitions.

If there is no current tags buffer, Tags search prompts for the name of a tags file. The default is a file called TAGS in the current directory. If there is

no such file, you can create one using Create Tags Buffer. If you want to search a different directory, specify the name of a tags file in that directory.

Continue Tags Search

Editor Command

Argument: None Key sequence: Meta+,

Searches for the next match for the last tag-based search. This command is only applicable if issued immediately after a Find Source, Find Tag or Tags Search command.

Tags Query Replace

Editor Command

Argument: target replacement [tags-file]

Key sequence: None

Replaces occurrences of *target* string by *replacement* string, for each file mentioned in the current tags buffer, 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 on page 74.

If there is no current tags file, 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

Argument: file

Key sequence: None

Creates a new tags buffer containing the tag information provided by *file*. The default is a file called **TAGS** in the current directory. The tag information contained in the new buffer is subsequently used for tag searches.

4.2.3 Tracing functions

The commands described in this section require that LispWorks is producing cross-referencing information. This information is produced by turning source

debugging on while compiling and loading the relevant definitions (see toggle-source-debugging in the *LispWorks Reference Manual*).

Trace Function Editor Command

Argument: *function* Key sequence: None

Traces *function*. The symbol under the current point is offered as a default value for *function*. A prefix argument automatically causes this default value to be used.

Trace Function Inside Definition

Editor Command

Argument: *function* Key sequence: None

Like Trace Function, but the function is only traced within the definition that contains the cursor.

Untrace Function Editor Command

Argument: *function* Key sequence: None

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

Traces the function defined by the current top-level form.

Untrace Definition Editor Command

Argument: None Key sequence: None

Untraces the function defined by the current top-level form.

4.2.4 Function callers and callees

The commands described in this section, require that LispWorks is producing cross-referencing information. This information is produced by turning source debugging on while compiling and loading the relevant definitions (see toggle-source-debugging in the *LispWorks Reference Manual*).

List Callers Editor Command

Argument: function Key sequence: None

Produces a Collection Viewer window listing those functions that call *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.

List Callees Editor Command

Argument: *function* Key sequence: None

Produces a Collection Viewer window listing those functions that are 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.

Show Paths To Editor Command

Argument: *function*Key sequence: None

Produces a Collection Graph Viewer window showing the callers of *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.

Show Paths From Editor Command

Argument: *function* Key sequence: None

Produces a Collection Graph Viewer window showing the function calls from *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.

4.2.5 Miscellaneous

Buffer Changed Definitions

Editor Command

Argument: None Key sequence: None

Displays a list of definitions that have been changed in the current buffer during the current LispWorks session. All definitions that have been modified or created (but not deleted) are displayed in a File Contents Collection window. Modified text between definitions (i.e. comments) are also included. 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.

Function Arglist

Editor Command

Argument: function

Key sequence: Meta+= function

Prints the arguments expected by *function* in the Echo Area. The symbol under the current point is offered as a default value for *function*. A prefix argument automatically causes this default value to be used.

Function Argument List

Editor Command

Argument: function

Key sequence: Ctrl+Shift+A function

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-function-ignores* in which case the second symbol in the form is offered as the default. A prefix argument automatically causes this default value to be used.

Describe Class Editor Command

Argument: *class*Key sequence: None

Displays a description of *class* in a class description window. The symbol under the current point is offered as a default value for *class*. A prefix argument automatically causes this default value to be used.

Describe Generic Function

Editor Command

Argument: function Key sequence: None

Displays a description of *function* in a Generic Function Description window. 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.

Complete Symbol

Editor Command

Argument: None

Key sequence: Meta+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 in a popup window.

4.3 Forms

4.3.1 Movement, marking and indentation

Forward Form Editor Command

Argument: None

Key sequence: Meta+Ctrl+F

Moves the current point to the end of the next form. A positive prefix argument causes the point to be moved the required number of forms forwards.

Backward Form Editor Command

Argument: None

Key sequence: Meta+Ctrl+B

Moves the current point to the beginning of the previous form. A positive prefix argument causes the point to be moved the required number of forms backwards.

Mark Form Editor Command

Argument: None Key sequence: None

Puts the mark at the end of the current form. The current region is that area from the current point to the end of form. A positive prefix argument puts the mark at the end of the relevant form.

Indent Form Editor Command

Argument: None

Key sequence: Meta+Ctrl+Q

If the current point is located at the beginning of a form, the whole form is indented in a manner that reflects the structure of the form. This command can therefore be used to format a whole definition so that the structure of the definition is apparent.

4.3.2 Killing forms

Forward Kill Form Editor Command

Argument: None

Key sequence: Meta+Ctrl+K

Kills the text from the current point up to the end of the current form. A positive prefix argument causes the relevant number of forms to be killed forwards. A negative prefix argument causes the relevant number of forms to be killed backwards.

Backward Kill Form Editor Command

Argument: None

Key sequence: Meta+Ctrl+Delete

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: Ctrl+x R

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.

4.3.3 Macro-expansion of forms

Macroexpand Form

Editor Command

Argument: None

Key sequence: Ctrl+Shift+M

Macro-expands the form after the current point. The output is sent to the Output window. A prefix argument causes the output to be displayed in the current buffer.

Walk Form Editor Command

Argument: None

Key sequence: Meta+Shift+M

Produces an exhaustive macro-expansion 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.

4.3.4 Miscellaneous

Transpose Forms

Editor Command

Argument: None

Key sequence: Meta+Ctrl+T

Transposes the forms immediately preceding and following the current point. A zero prefix argument causes the forms at the current point and the current mark to be transposed. A positive prefix argument causes the form at or preceding the current point to be transposed with the form the relevant number of forms forward. A negative prefix argument causes the form at or preceding the current point to be transposed with the form the relevant number of forms backward.

Find Unbalanced Parentheses

Editor Command

Arguments: None Key sequence: None

Moves the point to the end of the last properly matched form, thereby allowing you to identify any parentheses in your code that are unbalanced.

4.4 Lists

4.4.1 Movement

Forward List Editor Command

Argument: None

Key sequence: Meta+Ctrl+N

Moves the current point to the end of the current list. A positive prefix argument causes the point to be moved the required number of lists forwards.

Backward List Editor Command

Argument: None

Key sequence: Meta+Ctrl+P

Moves the current point to the beginning of the current list. A positive prefix argument causes the point to be moved the required number of lists backwards.

Forward Up List

Editor Command

Argument: None Key sequence: None

Moves the current point to the end of the current list by finding the first closing parenthesis that is not matched by an opening parenthesis after the current point.

Backward Up List

Editor Command

Argument: None

Key sequence: Meta+Ctrl+U

Moves the current point to the beginning of the current list by finding the first opening parenthesis that is not matched by a closing parenthesis before the current point.

Down List Editor Command

Argument: None

Key sequence: Meta+Ctrl+D

Moves the current point to a location down one level in the current list structure. A positive prefix argument causes the current point to be moved down the required number of levels.

4.4.2 Miscellaneous

Extract List Editor Command

Argument: None Key sequence: None

Extracts the current list and replaces the surrounding list by the current

list. Synonymous with Kill Backward Up List.

4.5 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.

Indent for Comment

Editor Command

Argument: None Key sequence: Meta+;

Creates a new comment or moves to the beginning of an existing comment, indenting it appropriately (see set Comment Column).

If the current point is in a line already containing a comment, that comment is indented as appropriate, and the current point is moved to the beginning of the comment. An existing double semicolon comment is aligned as for a line of code. An existing triple semicolon comment or a comment starting in column 0, is not moved.

A prefix argument causes comments on the next relevant number of lines to be indented. The current point is moved down the relevant number of lines.

If characters not associated with the comment extend past the comment column, a space is added before starting the comment.

Up Comment Line

Editor Command

Argument: None Key sequence: Meta+P

Moves to the previous line and then performs an Indent for Comment.

Down Comment Line

Editor Command

Argument: None Key sequence: Meta+N

Moves to the next line and then performs an Indent for Comment.

Indent New Comment Line

Editor Command

Argument: None

Key sequence: Meta+J or Meta+Newline

Ends the current comment and starts a new comment on the next line, using the indentation and number of comment start characters from the previous line's comment. If Indent New Comment Line is performed when the current point is not in a comment line, it simply acts as a Return.

Kill Comment Editor Command

Argument: None

Key sequence: Meta+Ctrl+;

Kills the comment on the current line and moves the current point to the next line. If there is no comment on the current line, the point is simply moved onto the next line. A prefix argument causes the comments on the relevant number of lines to be killed and the current point to be moved appropriately.

4.6 Parentheses

Insert () Editor Command

Argument: None

Key sequence: Meta+Shift+(

Inserts a pair of parentheses, positioning the current point after the opening parenthesis. A positive prefix argument causes the parentheses to be placed around the appropriate number of following forms.

Move Over) Editor Command

Argument: None

Key sequence: Meta+Shift+)

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.

4.7 Symbol Documentation

Describe Symbol Editor Command

Argument: *symbol* Key sequence: None

Displays a description (that is, value, property list, package, and so on) of *symbol* in a Help window. The symbol under the current point is offered as a default value for *string*. A prefix argument automatically causes this default value to be used.

Show Documentation

Editor Command

Argument: *symbol* Key sequence: None

Displays the documentation for the Lisp symbol symbol in a Help window (if any documentation exists).

Show Documentation Function

Editor Command

Argument: function

Key sequence: Meta+Ctrl+Shift+A

Displays the documentation for the Lisp function function in a Help window (if any documentation exists).

Show Documentation Variable

Editor Command

Argument: variable

Key sequence: Meta+Ctrl+Shift+V

Displays the documentation for the Global Lisp variable variable in a Help window (if any documentation exists).

Show Documentation Class

Editor Command

Argument: *class*Key sequence: None

Displays the documentation for the Lisp class class in a Help window (if

any documentation exists).

4.8 Evaluation and compilation

The commands described below allow the user to evaluate (interpret) or compile Lisp code that exists as text in a buffer. In some cases, the code may be used to modify the performance of the Editor itself.

4.8.1 General Commands

Current-Package

Editor variable

Default value: USER

Defines the value of the current package.

Set Buffer Package

Editor Command

Argument: package Key sequence: None

Set the package to be used by Lisp evaluation and compilation while in this buffer. Not to be used in the Listener, which uses the value of

package instead.

Set Buffer Output

Editor Command

Argument: *stream*Key sequence: None

Sets the output stream that evaluation results in the current buffer are sent to.

4.8.2 Evaluation commands

Evaluate Defun Editor Command

Argument: None

Key sequence: Meta+Ctrl+X

Evaluates the current top-level form. If the current point is between two

forms, the previous form is evaluated.

Re-evaluate Defvar Editor Command

Argument: None Key sequence: None

Evaluates the current top-level form if it is a $\tt defvar.$ If the current point is between two forms, the previous form is evaluated. The form is treated as

if the variable is not bound.

Evaluate Expression

Editor Command

Argument: expression

Key sequence: Esc+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 Region

Editor Command

Argument: None Key sequence: None

Evaluates the Lisp forms in the region between the current point and the mark.

Evaluate Buffer Editor Command

Argument: None Key sequence: None Evaluates the Lisp forms in the current buffer.

Load File Editor Command

Argument: file

Key sequence: None

Loads *file* into the current eval server, so that all Lisp forms in the file are evaluated.

Toggle Error Catch

Editor Command

Argument: None Key sequence: None

Toggles error catching for expressions evaluated in the editor. By default, if there is an error in an expression evaluated in the editor, a Notifier window is opened which provides the user with a number of options, including debug, re-evaluation and aborting of the editor command. However, this behavior can be changed by using Toggle Error Catch, so that in the event of an error, the error message is printed in the Echo Area, and the user is given no restart or debug options.

Evaluate Buffer Changed Definitions

Editor Command

Argument: None Key sequence: None

Evaluates definitions that have been changed in the current buffer during the current LispWorks session (use Buffer Changed Definitions on page 110 to see which definitions have changed). A prefix argument equal to the value of Prefix-Argument-Default causes evaluation of definitions changed since last evaluated. A prefix argument of 1 causes evaluation of definitions changed since last saved.

Evaluate Changed Definitions

Editor Command

Argument: None Key sequence: None Evaluates definitions in all Lisp buffers that have been changed during the current LispWorks session. The effect of prefixes is the same as for Evaluate Buffer Changed Definitions.

Evaluate System Changed Definitions

Editor Command

Argument: *system* Key sequence: None

Evaluates definitions that have been changed in *system* during the current LispWorks session.

4.8.3 Compilation commands

Compile Defun Editor Command

Argument: None Key sequence: None

Compiles the current top-level form. If the current point is between two forms, the previous form is evaluated.

Compile Region

Editor Command

Argument: None Key sequence: None

Compiles the Lisp forms in the region between the current point and the mark.

Compile File Editor Command

Argument: file

Key sequence: None

Compiles all Lisp forms in 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: None

Compiles the Lisp forms in the current buffer.

Compile Buffer File

Editor Command

Argument: None Key sequence: None

Compiles the file in the current buffer if its associated binary file is older than the source file or does not exist. For a file with a .lisp suffix, the corresponding binary file has a .fsl suffix. When the binary file is up to date, the user is asked if the source should be compiled anyway. When compile-Buffer-File-Confirm is non-nil, 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 non-nil, 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 on page 110 to see which definitions have changed). A prefix argument equal to the value of Prefix-Argument-Default causes compilation of definitions

changed since last compiled. A prefix argument of 1 causes compilation of definitions changed since last saved.

Compile Changed Definitions

Editor Command

Argument: None Key sequence: None

Compiles definitions in all Lisp buffers that have been changed during the current LispWorks session. The effect of prefixes is the same as for Compile Buffer Changed Definitions.

Compile System

Editor Command

Argument: *system* Key sequence: None

Compiles all definitions in system.

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*.

Advanced Features

The editor can be customized, both interactively and programmatically, to suit the users requirements.

The chapter "Command Reference" provides details of commands used to customize the editor for the duration of an editing session (see "Keyboard macros" on page 87, "Key bindings" on page 93, "Editor variables" on page 92). 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 128.
- The indentation used for Lisp forms can be modified to suit the preferences of the user—see "Customizing Lisp indentation" on page 129.
- Additional editor commands can be created by combining existing commands and providing specified arguments for them—see "Programming the editor" on page 130.

Note that the default configuration files mentioned in this chapter were used when LispWorks was released by the Harlequin Group Limited. 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.

5.1 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 93.

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 key bindings can be found in the file config/key-binds.lisp in the LispWorks library directory.

editor:bind-key Function

editor:bind-key name key &optional kind

Binds the command *name* to the key sequence or combination *key. kind* can take the value <code>:global</code>, <code>:mode</code>, or <code>:buffer</code>. The default is <code>:global</code>, as most key bindings are required to work throughout the editor. <code>:buffer</code> applies the binding only to a specified buffer. <code>:mode</code> applies the binding only to buffers in a specified major mode.

If this function is called interactively, you will be prompted as necessary for the kind of binding, the buffer or the major mode. Tab completion may be used at any stage.

Note that the function editor:bind-key is a programmatic version the editor command Bind Key.

The following examples, which are used to implement some existing key bindings, illustrate how key sequences can be specified using bind-key.

```
(in-package 'editor)
(bind-key "Forward Character" #\control-\f)
(bind-key "Forward Word" #\meta-\f)
(bind-key "Save File" '#(#\control-\x #\control-\s))
(bind-key "Regexp Forward Search" #\meta-control-\s)
(bind-key "Complete Field" #\space :mode "Echo Area")
(bind-key "Backward Character" "left")
(bind-key "Forward Word" #("control-right"))
```

editor:bind-string-to-key

Function

```
editor:bind-string-to-key string key &optional kind
```

Binds the text string *string* to the keyboard shortcut *key* without the need to create a macro explicitly. The *kind* option is as for editor:bind-key.

5.2 Customizing Lisp indentation

The indentation used for Lisp forms can be modified to suit the preferences of the user.

The default indentations can be found in the file config/indents.lisp in the LispWorks library directory. If you want to alter your personal Lisp indentation, put the modifying code in your .lispworks file.

editor:setup-indent

Function

editor:setup-indent form-name no-of-args &optional standard special

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

For example, the default indentation for if in Lisp code is established by:

```
(editor:setup-indent "if" 2 2 4)
```

This determines that the first 2 forms after the if (i.e. the test and the then clauses) get indented 4 spaces relative to the if, and any further forms (here, just an else clause) are indented by 2 spaces.

5.3 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 editor:defcommand).

Note that all the code found in this chapter is included in the directory examples/editor in the directory defined by the variable *lispworks-directory*.

The following sections describe editor functions that are not interactive editor commands. All the exist

ing commands are in the editor package, and new commands should also be created in that package.

Macro

5.3.1 Defining commands

defcommand

uciconimana			Wacio
	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 long-form command description.
		function-doc	A string which gives a short-form function description.
		forms	The Lisp code for the command.
	Values	command-name	The symbol name of the new command.

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 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.

```
(defcommand "Move Five" (p)
  "Moves the current point forward five characters.
  Any prefix argument is ignored."
  "Moves five characters forward."
  (forward-character-command 5))
```

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.

5.3.2 Buffers

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:buffers-name Function

editor:buffers-name buffer

Returns the name of buffer.

editor:window-buffer Function

editor:window-buffer window

Returns the buffer currently associated with window.

editor:buffers-start Function

editor:buffers-start buffer

Returns the starting point of buffer.

editor:buffers-end Function

editor:buffers-end buffer

Returns the end point of buffer.

editor:use-buffer Macro

editor:use-buffer buffer &body forms

Makes *buffer* the current buffer during the evaluation of *forms*.

editor:buffer-from-name Function

editor:buffer-from-name name

Returns the buffer called *name* (which should be a string). If there is no buffer with that name, nil is returned.

editor:make-buffer Function

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 Function

editor:goto-buffer buffer in-same-window

Makes *buffer* the current buffer. If *buffer* is currently being shown in a window then the cursor is moved there. If *buffer* is not currently in a window and *in-same-window* is non-nil then it is shown in the current window, otherwise a new window is created for it.

5.3.3 The echo area

editor:message Function

editor:message string &rest args

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

editor:clear-echo-area Function

editor:clear-echo-area &optional string force

Clears the Echo Area. The argument *string* is then printed in the Echo Area. If *force* is non-nil, the Echo Area is cleared immediately, with no delay. Otherwise, there may be a delay for the user to read any existing message.

5.3.4 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:editor-error Function

editor:editor-error string &rest args

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 may contain formatting characters to be interpreted by format. As with editor:message, *args* can consist of arguments to be printed within the string.

5.3.5 Files

editor:find-file-buffer Function

editor:find-file-buffer pathname &optional check-function

Returns a buffer associated with the file *pathname*, reading the file into a new buffer if necessary. The second value returned is **T** if a new buffer is created, and nil otherwise. If the file already exists in a buffer, its consis-

tency is first checked by means of *check-function*. If no value is supplied for *check-function*, editor:check-disk-version-consistent is used.

editor:fast-save-all-buffers

Function

editor:fast-save-all-buffers &optional ask

Saves all modified buffers which are associated with a file. If *ask* is non-nil then confirmation is asked for before saving each buffer. If *ask* is not set, all buffers are saved without further prompting.

Unlike the editor command save All Files this function can be run without any window interaction. It is thus suitable for use in code which does not intend to allow the user to leave any buffers unsaved, and from the console if it is necessary to save buffers without re-entering the full window system.

editor:check-disk-version-consistent

Function

editor:check-disk-version-consistent pathname buffer

Checks that the date of the file *pathname* is not more recent than the last time *buffer* was saved. If *pathname* is more recent, the user is prompted on how to proceed. Returns t if there is no need to read the file from disk and nil if it should be read from disk.

editor:buffer-pathname

Function

editor:buffer-pathname buffer

Returns the pathname of the file associated with *buffer*. If no file is associated with *buffer*, nil is returned.

5.3.6 Inserting text

editor:insert-string point string &optional start end

Function

editor:insert-string point string &optional start end

Inserts *string* at *point* in the current buffer. The arguments *start* and *end* specify the indices within *string* of the substring to be inserted. The default values for *start* and *end* are 0 and length string respectively.

editor:points-to-string

Function

editor:points-to-string start end

Returns the string between the points start and end.

5.3.7 Lisp

editor:*find-likely-function-ignores*

Variable

editor: *find-likely-function-ignores*

Contains a list of symbols likely to be found at the beginning of a form (i.e. apply, funcall, defun, defmethod, defgeneric).

5.3.8 Movement

editor:line-end Function

editor:line-end point

Moves *point* to be located immediately before the next newline character, or the end of the buffer if there are no following newline characters.

editor:line-start Function

editor:line-start point

Moves *point* to be located immediately after the previous newline character, or the start of the buffer if there are no previous newline characters.

editor:character-offset

Function

editor:character-offset point n

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

editor:word-offset Function

editor:word-offset point n

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

editor:line-offset Function

editor:line-offset point n &optional to-offset

Moves *point* n lines forward, to a location *to-offset* characters into the line. If n is negative, *point* moves back n lines. If *to-offset* is nil (its default value), an attempt is made to retain the current offset. An error is signalled if there are not n further lines in the buffer.

editor:form-offset Function

editor:form-offset point n &optional form depth

Moves *point* forward n Lisp forms. If n is negative, point moves back n forms. If *form* is t (its default value) then atoms are counted as forms, otherwise they are ignored. Before point is moved forward n forms, it first jumps out *depth* levels. The default value for *depth* is 0.

5.3.9 Points

editor:current-point

Function

editor:current-point

Returns the current point.

editor:current-mark Function

editor:current-mark &optional pop-p no-error-p

Returns the current mark. If pop-p is t, the mark is popped off the point ring. If no mark is set and no-error-p is t, nil is returned; otherwise an error is signalled. The default for both of these optional arguments is nil.

editor:set-current-mark

Function

editor:set-current-mark point

Sets the current mark to be *point*.

editor:buffer-point

Function

editor:buffer-point buffer

Returns the current point in *buffer*.

editor:point<

Function

editor:point< point1 point2</pre>

Returns non-mil if *point1* is before *point2* in the buffer.

editor:point>

Function

editor:point> point1 point2

Returns non-mil if *point1* is after *point2* in the buffer.

editor:copy-point

Function

editor:copy-point point &optional kind new-point

Makes and returns a copy of *point*. The argument *kind* can take the value :before, :after, or :temporary. If *new-point* is supplied, the copied point is bound to that as well as being returned.

editor: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 the current 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 the current 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*.

5.3.10 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 key fields are common to a number of prompting functions.

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.

:must-exist Specifies whether the value that is input by the user must

be an existing value or not. If :must-exist is non-ni1, 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).

:help Provides a help message that is printed if the user types

"?".

editor:prompt-for-file

Function

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

Prompts for a file name, and returns a pathname. See above for an explanation of the keywords.

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 default-string help

Prompts for an integer. See above for an explanation of the keywords.

editor:prompt-for-string

Function

editor:prompt-for-string &key prompt default default-string clear-echo-area help

Prompts for a string. No checking is done on the input. The keyword *clear-echo-area* controls whether or not the echo area is cleared (that is, whether the text being replaced is visible or not). The default for this keyword is t. See above for an explanation of the remaining keywords.

editor:prompt-for-variable

Function

editor:prompt-for-variable &key must-exit prompt default default-string help

Prompts for a variable. See above for an explanation of the keywords. The default value of *must-exist* is t.

5.3.11 Variables

editor:variable-value Function

editor:variable-value name &optional kind where

Returns the value of the variable *name*. An error is signalled if the variable is undefined. The argument *kind* can take the value :current, :global or :mode. The default value of *kind* is :current. The argument *where* should be the name of a buffer.

For example, the code given below will, by default, return the value :ask-user.

```
(editor:variable-value
  "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.

```
(setf
  (editor:variable-value
    "Add Newline At Eof On Writing File")
    nil)
```

editor:variable-value-if-bound

Function

editor:variable-value-if-bound name &optional kind where

Returns the value of the variable name. If the variable is not bound, NIL is returned. The argument kind can take the value :current, :global or :mode. The default value of kind is :current. Note that where should be the name of a buffer.

5.3.12 Windows

editor:current-window

Function

editor:current-window

Returns the current window.

editor:redisplay

Function

editor:redisplay

Redisplays any window that appears to need it. In general, the contents of a window may not be redisplayed until there is an event to provoke it.

5.3.13 Examples

5.3.13.1 Example 1

The following simple example creates a new editor command called what Line.

```
(in-package 'editor)
(defcommand "What 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 (current-point))
           (svpoint (copy-point cpoint))
           (count 0))
          (beginning-of-buffer-command nil)
          (loop
             (if (point > cpoint svpoint)
                 (return))
             (unless (next-line-command)
                 (return))
             (incf count))
          (move-point cpoint svpoint)
          (message "Line Number: ~S " count)))
```

5.3.13.2 Example 2

This example creates a new editor command called **Goto** Line which moves the current point to the specified line number.

5.3.13.3 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.

To test this example, two buffers named ± 1 and ± 2 should be created. Then, to copy all the text from ± 1 to the end of ± 2 :

Advanced Features

Glossary

Abbrev

An abbrev (abbreviation) is a user defined text string which, when typed into a buffer, may be expanded into another string. Typing can therefore be saved by defining short strings to be expanded into frequently used longer words or phrases.

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 string into its full name. Completion can used for expanding command names, file names and 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 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.

Escape Key

The Escape key (Esc) has its own functionality but is mostly used in place of the Meta key when no such key exists on a keyboard. Esc must be typed *before* pressing the required character key.

Extended Command

Most editor commands can be invoked explicitly by using their full command names, preceded by the Meta+x key sequence. A command issued in such a way is known as an extended command.

Fill Prefix

The fill prefix is a string which is ignored when filling takes place. For example, if the fill prefix is ;;, then these characters at the start of a line are skipped over when the text is re-formatted.

Filling

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

Global Abbrev

A global abbrev is an abbrev which can be expanded in all major modes.

History Ring

The history ring records Echo Area commands so that they can easily be repeated.

Incremental Search

An incremental search is a search which is started as soon as the first character of the search string is typed.

Indentation

Indentation is the blank space at the beginning of a line. Lisp, like many other programming languages, has conventions for the indentation of code to make it more readable. The editor is designed to facilitate such indentation.

Insertion

Insertion is the process of inputting text into a buffer.

Keyboard Macro

A keyboard macro allows a sequence of editor commands to be turned into a single operation. Keyboard macros are only available for the duration of an editing session.

Key Sequence

A key sequence is a sequence of characters used to issue, or partly issue, an editor command. A single key sequence usually involves holding down one of two specially defined keys (i.e. Ctrl and Meta), while at the same time pressing another key which is usually a character 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

There are many different types of keyboard, and the Meta key may not be marked with "Meta". It may be marked with a special character, such as a diamond, or it may be one of the function keys — try f11. 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.

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: <code>space Tab @ - '</code>

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 (i.e. it actually lies *between* two characters). Many types of commands (i.e. 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 space, tab or Newline.

Window

A window is an object used by the window manager to display data. When the editor is called up, an editor window is created and displayed.

Window Ring

A window ring is used to hold details of all windows currently open.

Word

A word is a continuous string of alphanumeric characters (i.e. the letters A–Z and numbers 0–9). In most modes, any character which is not alphanumeric is treated as a word delimiter.

Yanking

Yanking means inserting a previously killed item of text from the kill ring at a required location. This is often known as *pasting*.

Index

Symbols	argument
# files 23	listing for function 110
? Help on Parse 90	prefix 22
~ files 23, 29	attribute
11105 20, 20	description 20
A	listing with apropos 18
	Auto Fill Linefeed 58
Abbrev Expand Only 83	Auto Fill Mode 58
Abbrev Mode 82	Auto Fill Return 58
abbrev mode 79, 82	Auto Fill Space 58
abbreviation	Auto Save Toggle 28
add global 83	auto-fill mode 57, 79
add global expansion 83	auto-save file 27
add mode 82	Auto-Save-Checkpoint-Frequency 28
add mode expansion 82	Auto-Save-Filename-Pattern 28
append to file 86	Auto-Save-Key-Count-Threshold 28
delete all 85	·
delete global 84	В
delete mode 84	Back to Indentation 54
edit 85	Backup File 26
editor definition 81	backup file 26, 29
expand 83	Backup-Filename-Pattern 29
list 85	Backup-Filename-Suffix 29
read from file 86	Backups-Wanted 29
save to file 86	Backward Character 32
undo last expansion 84	Backward Form 112
abbreviation commands 81	Backward Kill Form 113
Abbrev-Pathname-Defaults 86	Backward Kill Line 44
Abort Recursive Edit 93	Backward Kill Sentence 44
aborting editor commands 11, 16	Backward List 116
aborting processes 11, 16	Backward Paragraph 34
Add Global Word Abbrev 83	Backward Sentence 34
Add Mode Word Abbrev 82	Backward Up List 116
Add-Newline-at-EOF-on-Writing-File 27	Backward Word 33
Append to Word Abbrev File 86	Beginning of Buffer 36
Apropos 18	2-5 01 Dailor 00

Beginning of Defun 102	mouse bindings in editor 98
Beginning of Line 33	
Beginning Of Parse 91	C
Bind Key 94	Capitalize Region 50
binding	Capitalize Word 49
editor definition 10	case conversion commands 48
binding keys 93	Case-Replace 76
bind-key 128	CD 96
Bottom of Window 36	Center Line 57
breaking processes 16	character
buffer	backward 32
changed definitions in 110	delete expanding tabs 42
compile 125	delete next 41
compile changed definitions 125	delete previous 42
compile if necessary 125	forward 32
create 60	insert with overwrite 52
editor definition 6	overwrite previous 52
evaluate 122	transposition 50
evaluate changed definitions 123	character-offset 137
file options 31	Check Buffer Modified 61
insert 61	check-disk-version-consistent 136
kill 60	class
list 60	describe 111
mark whole 40	documentation 121
modified check 61	clear-echo-area 134
move to beginning 36	command
move to end 37	abort 16
not modified 62	completion 10, 17, 89
print 61	description 19, 20
read only 61	execution 9, 16
rename 61	history 21
save 25	key sequence for 21
search all 70	listing with apropos 18
select 59	repetition 11, 22
select in other window 59	shell 95
select previous 59	commands
set package 121	abbreviation 81
Buffer Changed Definitions 110	aborting commands 11, 16
buffer commands 59	aborting processes 11, 16
buffer functions 132, 143	buffer 59
Buffer Not Modified 62	case conversion 48
buffer-from-name 133	compilation 121, 124
buffer-list 132	cut and paste 13
buffer-pathname 136	deleting text 13, 41
buffer-point 139	echo area 89
buffers and windows 98	editing Lisp programs 101
buffers-end 133	editor variable 92
buffers-start 133	evaluation 121, 122
bug	file handling 12, 23
reporting 99	filling 55
Bug Report 99	help 14, 17
button	indentation 52

inserting text 12, 45	dynamic word 47
key binding 93	of commands 10, 17, 89
keyboard macro 87	of symbols 111
killing text 13, 41	configuration files 127
Lisp comment 117	Confirm Parse 89
Lisp form 112	Continue Tags Search 107
Lisp function and definition 102	Control key 9
Lisp list 115	control keys
Lisp symbol documentation 120	insert into buffer 46
movement 12, 32	Copy to Cut Buffer 98
overwriting 51	copy-point 139
pages 64	Count Lines Page 65
parentheses 119	Count Lines Region 40
recursive editing 93	Count Occurrences 71
register 76	Count Words Region 40
replacing 66	Create Buffer 60
running shell from editor 95	Create Tags Buffer 105
searching 66	Ctrl key 9
transposition 50	Ctrl+] Abort Recursive Edit 93
undoing 13, 47	Ctrl+A Beginning of Line 33
window 62	Ctrl+A Beginning Of Parse 91
comment	Ctrl+B Backward Character 32
create 117	Ctrl+B Echo Area Backward
kill 118	Character 91
move to 117	Ctrl+D Delete Next Character 41
comment commands 117	Ctrl+E End of Line 33
compilation commands 121, 124	Ctrl+F Forward Character 32
compile	Ctrl+G, abort current command 16
buffer 125	Ctrl+H A Apropos 14,18
buffer changed definitions 125	Ctrl+H C What Command 19
buffer if necessary 125	Ctrl+H Ctrl+D Document Command 19
changed definitions 126	Ctrl+H Ctrl+K Document Key 20
file 124	Ctrl+H Ctrl+V Document Variable 21
form 124	Ctrl+H D Describe Command 14, 19
region 124	Ctrl+H G Generic Describe 20
system 126	Ctrl+H Help 17
system changed definitions 126	Ctrl+H K Describe Key $14,20$
Compile Buffer 125	Ctrl+H L What Lossage 21
Compile Buffer Changed Definitions 125	Ctrl+H V Describe and Show
Compile Buffer File 125	Variable 21
Compile Changed Definitions 126	Ctrl+H W Where Is 21
Compile Defun 124	Ctrl+K Kill Line 44
Compile File 124	Ctrl+L Refresh Screen 64
Compile Region 124	Ctrl+N Next Line 33
Compile System 126	Ctrl+P Insert Parse Default 92
Compile System Changed	Ctrl+P Previous Line 33
Definitions 126	Ctrl+Q Quoted Insert 46
Compile-Buffer-File-Confirm 125	Ctrl+R Esc Reverse Search 69
Complete Field 89	Ctrl+R Return Default 92
Complete Input 89	Ctrl+R Reverse Incremental Search 67
Complete Symbol 111	Ctrl+S Esc Forward Search 68
completion 47	Ctrl+S Incremental Search 67

Ctrl+Shift+_ Undo 13,48	Ctrl+X I Insert File 31
Ctrl+Shift+A Function Argument	Ctrl+X J Jump to Saved Position 76
List 111	Ctrl+X K Kill Buffer 60
Ctrl+Shift+M Macroexpand Form 114	Ctrl+X L Count Lines Page 65
Ctrl+Shift+X Ctrl+Shift+O Delete	Ctrl+X O Next Window 62
Blank Lines 42	Ctrl+X Q Keyboard Macro Query 88
Ctrl+Space Set Mark 38	Ctrl+X S Save All Files 26
Ctrl+T Transpose Characters 50	Ctrl+X Tab Indent Rigidly 54
Ctrl+U Kill Parse 92	Ctrl+X X Put Register 77
Ctrl+U Set Prefix Argument 22	Ctrl+Y Un-Kill 14,45
Ctrl+V Scroll Window Down 35	Ctrl-C Ctrl-C Interrupt Shell
Ctrl+W Kill Region 44	Subjob 97
Ctrl+X - Inverse Add Global Word	Ctrl-C Ctrl-D Shell Send Eof 97
Abbrev 83	Ctrl-C Ctrl-Z Stop Shell Subjob 97
Ctrl+X (Define Keyboard Macro 87	current point
Ctrl+X) End Keyboard Macro 88	editor definition 7
Ctrl+X + Add Global Word Abbrev 83	current-buffer 132
Ctrl+X . Set Fill Prefix 57	current-mark 139
Ctrl+X / Save Position 76	Current-Package 121
Ctrl+X ; 117	current-point 138
Ctrl+X [Previous Page 64	current-window 143
Ctrl+X] Next Page 65	Cursor Position 37
Ctrl+X ~ Check Buffer Modified 61	customising
Ctrl+X 0 Delete Window 63	editor 127
Ctrl+X 1 Delete Next Window 63	editor commands 127
Ctrl+X 2 New Window 62	indentation of Lisp forms 127, 129
Ctrl+X b Select Buffer 59	key bindings 127, 128
Ctrl+X Ctrl+A Add Mode word Abbrev 82	cut and paste commands 13
Ctrl+X Ctrl+B List Buffers 60	_
Ctrl+X Ctrl+C Save All Files and	D
Exit 27	debugger
Ctrl+X Ctrl+F Wfind File 25	using in editor 123
Ctrl+X Ctrl+H Inverse Add Mode Word	default
Abbrev 82	binding 10
Ctrl+X Ctrl+I Indent Rigidly 54	prefix argument 22, 23
Ctrl+X Ctrl+L Lowercase Region 49	Default-Search-Kind 71
Ctrl+X Ctrl+P Mark Page 65	defcommand macro 130
Ctrl+X Ctrl+Q Set Buffer Read-	Defindent 103
Only 61	Define Keyboard Macro 87
Ctrl+X Ctrl+S Save File 25	Define Word Abbrevs 87
Ctrl+X Ctrl+T Transpose Lines 51	definition
Ctrl+X Ctrl+U Uppercase Region 49	disassemble 126
Ctrl+X Ctrl+V Visit File 25	editing 102
Ctrl+X Ctrl+W Write File 26	find 104
Ctrl+X Ctrl+X Exchange Point and	find buffer changes 110
Mark 39	searching for 103
Ctrl+X Delete Backward Kill	trace 108
Sentence 44	untrace 109
Ctrl+X E Last Keyboard Macro 88	defmode function 80
Ctrl+X F Set Fill Column 56	Delete All Word Abbrevs 85
Ctrl+X G Get Register 77	Delete Blank Lines 42
Ctrl+X H Mark Whole Buffer 40	DELETE Delete Previous Character 42

DELETE Echo Area Delete Previous	next command 90
Character 91	previous command 90
Delete File 31	prompting the user 140
Delete Global Word Abbrev 84	repeating commands in 90
Delete Horizontal Space 42	terminate entry 89
Delete Indentation 54	Echo Area Backward Character 91
Delete Key Binding 94	Echo Area Backward Word 91
Delete Matching Lines 69	echo area commands 89
Delete Mode Word Abbrev 84	Echo Area Delete Previous Character 91
Delete Next Character 41	echo area functions 134, 143
Delete Next Window 63	Echo Area Kill Previous Word 91
Delete Non-Matching Lines 70	Edit Word Abbrevs 85
Delete Previous Character 42	editor
Delete Previous Character Expanding	customising 127
Tabs 42	$ ext{delete-region-command}$
Delete Region 43	programming 130
Delete Window 63	editor commands 46
deleting text 41	Abbrev Expand Only 83
deleting text commands 13, 41	Abbrev Mode 82
deletion	Abort Recursive Edit Ctrl+] 93
editor definition 41	Add Global Word Abbrev Ctrl+x + 83
delimiter	Add Mode Word Abbrev Ctrl+X
sentence 9	Ctrl+A 82
Describe and Show Variable 21	Append to Word Abbrev File 86
Describe Class 111	Apropos Ctrl+H A 18
Describe Command 19	Auto Fill Linefeed LINEFEED 58
Describe Generic Function 111	Auto Fill Mode 58
Describe Key 20	Auto Fill Return RETURN 58
Describe Symbol 120	Auto Fill Space SPACE 58
directory	Auto Save Toggle 28
change 96	Back to Indentation Meta+M 54
query replace 75	Backup File 26
search 70	Backward Character Ctrl+B 32
Directory Query Replace 75	Backward Form Meta+Ctrl+B 112
Directory Search 70	Backward Kill Form
Disassemble Definition 126	Meta+Ctrl+Delete 113
Do Nothing 95	Backward Kill Line 44
Document Command 19	Backward Kill Sentence Ctrl+X
Document Key 20	Delete 44
Document Variable 21	Backward List Meta+Ctrl+P 116
Down Comment Line 118	Backward Paragraph Meta+[34
Down List 116	Backward Sentence Meta+A 34
Dynamic Completion 47	Backward Up List Meta+Ctrl+U 116
2 James Completion 1.	Backward Word Meta+B 33
E	Beginning of Buffer Meta+< 36
-	Beginning of Defun Meta+Ctrl+A 102
echo area	Beginning of Line Ctrl+A 33
complete text 89	Beginning Of Parse Ctrl+A 91
completing commands in 89	Bind Key 94
deleting and inserting text in 91	Bottom of Window 36
editor definition 89	Buffer Changed Definitions 110
help on parse 90	Buffer Not Modified Meta+Shift+~ 62
movement in 91	

Bug Report 99	v 21
Capitalize Region 50	Describe Class 111
Capitalize Word Meta+C 49	Describe Command Ctrl+H D 19
CD 96	Describe Generic Function 111
Center Line 57	Describe Key Ctrl+н к 20
Check Buffer Modified Ctrl+X ~ 61	Describe Symbol 120
Compile Buffer 125	Directory Query Replace 75
Compile Buffer Changed	Directory Search 70
Definitions 125	Disassemble Definition 126
Compile Buffer File 125	Do Nothing 95
Compile Changed Definitions 126	Document Command Ctrl+H
Compile Defun 124	Ctrl+D 19
Compile File 124	Document Key Ctrl+H Ctrl+K 20
Compile Region 124	Document Variable Ctrl+H Ctrl+V 21
Compile System 126	Down Comment Line Meta+N 118
Compile System Changed	Down List Meta+Ctrl+D 116
Definitions 126	Dynamic Completion Meta+/ 47
Complete Field SPACE 89	Echo Area Backward Character
Complete Input TAB 89	Ctrl+B 91
• •	Echo Area Backward Word Meta+B 91
Complete Symbol Meta+Ctrl+I 111 Confirm Parse RETURN 89	Echo Area Delete Previous Character
	DELETE 91
Convito Cut Buffer 09	
Count Lines Page Start vs. 7, 65	Echo Area Kill Previous Word
Count Lines Page Ctrl+X L 65	Meta+Delete 91
Count Lines Region 40	Edit Word Abbrevs 85
Count Occurrences 71	End Keyboard Macro Ctr1+x) 88
Count Words Region 40	End of Buffer Meta+> 37
Create Buffer 60	End of Defun Meta+Ctrl+E 102
Create Tags Buffer 105	End of Line Ctrl+E 33
Cursor Position 37	Evaluate Buffer 122
Defindent 103	Evaluate Buffer Changed
Define Keyboard Macro Ctrl+x (87	Definitions 123
Define Word Abbrevs 87	Evaluate Changed Definitions 123
Delete All Word Abbrevs 85	Evaluate Defun Meta+Ctrl+X 122
Delete Blank Lines Ctrl+Shift+X	Evaluate Expression
Ctrl+Shift+O 42	Escape+Escape 122
Delete File 31	Evaluate Region 122
Delete Global Word Abbrev 84	Evaluate System Changed
Delete Horizontal Space Meta+\ 42	Definitions 124
Delete Indentation Meta+Shift+^ 54	Exchange Point and Mark Ctrl+X
Delete Key Binding 94	Ctrl+X 39
Delete Matching Lines 69	Exit Recursive Edit Meta+Ctr1+Z 93
Delete Mode Word Abbrev 84	Expand File Name Meta+Tab 47
Delete Next Character Ctrl+D 41	Extended Command Meta+x 10, 17
Delete Next Window Ctrl+x 1 63	Extract List 117
Delete Non-Matching Lines 70	Fill Paragraph Meta+Q 55
Delete Previous Character DELETE 42	Fill Region Meta+G 56
Delete Previous Character Expanding	Find Dspec Meta+. 104
Tabs 42	Find File 24
Delete Region 43	Find Source for Dspec 105
Delete Window Ctrl+x 0 63	Find Tag Meta+? 106
Describe and Show Variable Ctrl+H	Forward Character Ctrl+F 32

Forward Form Meta+Ctrl+F 112 Kill Previous Word Meta+Delete 43 Forward Kill Form Meta+Ctrl+K 113 Kill Region Ctrl+W 44 Forward Kill Sentence Meta+K 44 Kill Register 77 Forward List Meta+Ctrl+N 115 Last Keyboard Macro Ctrl+x E 88 Forward Paragraph Meta+] 34 Line to Top of Window 36 Forward Search Ctrl+S Esc 68 Lisp Insert) 119 Forward Sentence Meta+E 34 Lisp Mode 79 Forward Up List 116 List Buffers Ctrl+X Ctrl+B 60 Forward Word Meta+F 32 List Callees 109 Function Arglist Meta+= 110 List Callers 109 **Function Argument List List Matching Lines** 69 Ctrl+Shift+A 111 List Registers 77 Fundamental Mode 78 List Word Abbrevs 85 Generic Describe Ctrl+H G 20 Load File 123 Lowercase Region Ctrl+X Ctrl+L 49 Get Register Ctrl+x G 77 Goto Line 34 Lowercase Word Meta+L 48 Goto Page 65 Macroexpand Form Ctrl+Shift+M 114 Goto Point 37 Make Word Abbrev 83 Help Ctrl+H 17 Manual Entry 22 Help on Parse ? 90 Mark Defun Meta+Ctrl+H 103 Illegal 94 Mark Form 112 Incremental Search Ctrl+s 67 Mark Page Ctrl+X Ctrl+P 65 Indent for Comment Meta+; 117 Mark Paragraph Meta+H 40 Indent Form Meta+Ctrl+Q 113 Mark Sentence 39 Mark Whole Buffer Ctrl+x H 40 Indent New Comment Line Meta+J or Meta+Newline 118 Move Over) Meta+) 119 **Indent New Line 55** Name Keyboard Macro 88 Indent Region Meta+Ctrl+\ 53 Negative Argument 23 Indent Rigidly Ctrl+X Tab, Ctrl+X New Line return 46 New Window Ctrl+x 2 62 Ctrl+I 54 Indent TAB 53 Next Line Ctrl+N 33 Insert () Meta+(119 Next Page Ctrl+x] 65 Next Parse Meta+N 90 **Insert Buffer** 61 **Insert Cut Buffer** 98 Next Window Ctrl+x 0 62 Insert File Ctrl+x I 31 **Overwrite Delete Previous** Character 52 Insert Page Directory 66 Insert Parse Default Ctrl+P 92 Overwrite Mode 52 **Insert Word Abbrevs 87** Pop and Goto Mark 39 Interrupt Shell Subjob Ctrl-C Ctrl-Pop Mark Meta+Ctrl+Space 39 c 97 Previous Line Ctrl+P 33 **Inverse Add Global Word Abbrev** Previous Page Ctrl+x [64 Previous Parse Meta+P 90 Ctrl+x - 83 Inverse Add Mode Word Abbrev Previous Window 63 Ctrl+X Ctrl+H 82 Print Buffer 61 Jump to Saved Position Ctrl+X J 76 Print File 30 Just One Space Meta+Space 42 Print Region 41 Keyboard Macro Query Ctrl+x Q 88 **Process File Options** 31 Kill Buffer Ctrl+x K 60 Put Register Ctrl+x x 77 Kill Comment Meta+Ctrl+; 118 Query Replace Meta+Shift+% 74 Kill Line Ctrl+K 44 Quote Tab 55 Kill Next Word Meta+D 43 Quoted Insert Ctrl+Q 46 Kill Parse Ctrl+U 92 Read Word Abbrev File 86

Re-evaluate Defvar 122	Show Paths To 109
Refresh Screen Ctrl+L 64	Show Variable 92
Regexp Forward Search	Skip Whitespace 37
Meta+Ctrl+S 73	Stop Shell Subjob Ctrl-C Ctrl-Z 97
Regexp Reverse Search Meta+Ctrl+R 74	System Query Replace 76
Rename Buffer 61	System Search 70
Rename File 31	Tags Query Replace 107
Replace String 74	Tags Search 106
Return Default Ctrl+R 92	Text Mode 79
Reverse Incremental Search Ctrl+R 67	Toggle Error Catch 123
Reverse Search Ctrl+R Esc 69	Top of Window 36
Revert File 30	Trace Definition 108
Revert-File-Confirm 30	Trace Function 108
Room 100	Transpose Characters Ctrl+T 50
Rotate Kill Ring Meta+Y 45	Transpose Forms Meta+Ctrl+T 115
Run Command 95	Transpose Lines Ctrl+X Ctrl+T 51
Save All Files and Exit Ctrl+X	Transpose Regions 51
Ctrl+C 27	Transpose Words Meta+T 50
Save All Files Ctrl+x s 26	Undo Ctrl+Shift+_ 48
Save File Ctrl+X Ctrl+S 25	Unexpand Last Word 84
Save Position Ctrl+x / 76	Un-Kîll Ctrl+Y 45
Save Region Meta+W 45	Untrace Definition 109
Scroll Next Window Down 63	Untrace Function 108
Scroll Next Window Up 63	Up Comment Line Meta+P 118
Scroll Window Down Ctrl+V 35	Uppercase Region Ctrl+x Ctrl+U 49
Scroll Window Up Meta+V 35	Uppercase Word Meta+U 49
Search All Buffers 70	View Page Directory 66
Select Buffer Ctrl+X b 59	View Source Search 105
Select Buffer Other Window 59	Visit File Ctrl+X Ctrl+V 25
Select Previous Buffer Meta+Ctrl+L 59	Visit Tags File 107
Self Insert 47	Walk Form Meta+Shift+M 114
Self Overwrite 52	Wfind File Ctrl+X Ctrl+F 25
Set Buffer Output 121	What Command Ctrl+H C 19
Set Buffer Package 121	What Lossage Ctrl+H L 21
Set Buffer Read-Only Ctrl+X	Where Is Ctrl+H w 21
Ctrl+Q 61	Where is Point 37
Set Comment Column Ctrl+x ; 117	Word Abbrev Apropos 85
Set Fill Column Ctrl+X F 56	Word Abbrev Prefix Point Meta+' 84
Set Fill Prefix Ctrl+x . 57	Write File Ctrl+X Ctrl+W 26
Set Mark Ctrl+Space 38	Write Region 26
Set Prefix Argument Ctrl+U 22	Write Word Abbrev File 86
	editor functions
Shell 95	bind-key 128
Shell Command Meta-! 95	buffer-from-name 133
Shell Send Eof Ctrl-C Ctrl-D 97	buffer-pathname 136
Show Documentation 120	buffer-point 139
Show Documentation Class 121	buffers-end 133
Show Documentation Function	buffers-start 133
Meta+Ctrl+Shift+A 120	character-offset 137
Show Documentation Variable	check-disk-version-consistent 136
Meta+Ctrl+Shift+V 120	clear-echo-area 134
Show Paths From 110	copy-point 139

current-buffer 132	Current-Package 121
current-mark 139	Default-Search-Kind 71
current-point 138	Fill-Column 56
current-window 143	Fill-Prefix 56
editor-error 135	Prefix-Argument-Default 23
end-line-p 140	Region-Query-Size 41
fast-save-all-buffers 136	Revert-File-Confirm 30
find-file-buffer 135	Ring-Size 48
form-offset 138	Scroll-Overlap 36
goto-buffer 134	Shell-cd-RegExp 96
insert-string 136	Shell-pop-RegExp 97
line-end 137	Shell-push-RegExp 96
line-offset 138	Spaces-For-Tab 53
line-start 137	editor-error 135
make-buffer 133	End Keyboard Macro 88
message 134	End of Buffer 37
move-point 140	End of Defun 102
point< 139	End of Line 33
point> 139	end-line-p 140
points-to-string 137	error
prompt-for-buffer 141	catching evaluation 123
prompt-for-file 141	editor 134
prompt-for-integer 141	error functions 134
prompt-for-string 142	Escape key 9
prompt-for-variable 142	Escape+Escape Evaluate
redisplay 143	Expression 122
same-line-p 140	evaluate
save-excursion 140	buffer 122
setup-indent 129	buffer changed definition 123
start-line-p 140	changed definitions 123
variable-value 142	defvar 122
variable-value-if-bound 143	expression 122
window-buffer 133	file 123
word-offset 138	form 122
editor macros	region 122
use-buffer 133	system changed definitions 124
editor package 130	Evaluate Buffer 122
editor variable 92	Evaluate Buffer Changed Definitions 123
editor variables	Evaluate Changed Definitions 123
buffer-list 132	Evaluate Defun 122
find-likely-function-ignores 137	Evaluate Expression 122
Abbrev-Pathname-Defaults 86	Evaluate Region 122
Add-Newline-at-EOF-on-Writing-	Evaluate System Changed
File 27	Definitions 124
Auto-Save-Checkpoint-Frequency 28	evaluation commands 121, 122
Auto-Save-Filename-Pattern 28	examples
Auto-Save-Key-Count-Threshold 28	programming the editor 143
Backup-Filename-Pattern 29	Exchange Point and Mark 39
Backup-Filename-Suffix 29	execute mode 79
Backups-Wanted 29	executing editor commands 9, 16
Case-Replace 76	Exit Recursive Edit 93
Compile-Buffer-File-Confirm 125	Expand File Name 47

expression	Forward Character 32
evaluate 122	Forward Form 112
Extended Command 10, 17	Forward Kill Form 113
Extract List 117	Forward Kill Sentence 44
Introduction 117	Forward List 115
F	Forward Paragraph 34
	Forward Search 68
fast-save-all-buffers 136	Forward Sentence 34
file	
auto-saving 27	Forward Up List 116 Forward Word 32
backup 26, 29	
compile 124	function
delete 31	argument list 110
editor definition 6	describe generic 111
evaluate 123	documentation 120
expand name 47	editing 102
finding 24	find definition 103
insert into buffer 31	indentation 103
options for buffer 31	list callees 109, 110
print 30	list callers 109
rename 31	mark 103
revert 30	move to beginning 102
save 25, 27	move to end 102
visit 25	trace 108
write 26	untrace 108
file functions 143	Function Arglist 110
file handling commands 12, 23	Function Argument List 111
filename completion 47	functions
Fill Paragraph 55	buffer 132, 143
Fill Region 56	${\tt defmode}\ 80$
Fill-Column 56	echo area 134, 143
filling commands 55	editor error 134
Fill-Prefix 56	editor, see editor functions
Find File 24	file 143
Find Source 104	inserting text 136
Find Source for Dspec 105	Lisp editor 137
Find Tag 106	movement 137, 143
Find Unbalanced Parentheses 115	point 138
find-file-buffer 135	prompt 140
find-likely-function-ignores 137	search-files 71
form	variable 142
compile 124	window 143
evaluate 122	Fundamental Mode 78
indent 113	fundamental mode 78
kill backwards 113	
kill forwards 113	G
macro-expand 114	Generic Describe 20
mark 112	generic function
move to beginning 112	describe 111
move to beginning 112 move to end 112	Get Register 77
transposition 115	global abbreviation
form commands 112	editor definition 82
form-offset 138	Goto Line 34

Goto Page 65	Control 9
Goto Point 37	description 20
goto-buffer 134	Escape 9
**	Meta 9
Н	key binding 93
Help 17	customising 127, 128
help commands 14, 17	key sequence
Help on Parse 90	editor definition 9 for command 21
history of commands 21	
history ring 90	keyboard macro
_	begin definition of 87 editor definition 87
I	end definition of 88
Illegal 94	execute 88
Incremental Search 67	name 88
Indent 53	keyboard macro commands 87
indent	Keyboard Macro Query 88
form 113	Kill Buffer 60
Indent for Comment 117	Kill Comment 118
Indent Form 113	Kill Line 44
Indent New Comment Line 118	Kill Next Word 43
Indent New Line 55	Kill Parse 92
Indent Region 53	Kill Previous Word 43
Indent Rigidly 54	Kill Region 44
indentation	Kill Register 77
customising 127, 129	kill ring 41, 43, 45
define for Lisp forms 103	rotate 45
define for Lisp functions 103	killing
delete 54	editor definition 41
move back to 54	killing text 43
indentation commands 52	killing text commands 13, 41
Insert () 119	
Insert Buffer 61	L
Insert Cut Buffer 98	Last Keyboard Macro 88
Insert File 31	line
Insert Page Directory 66	beginning 33
Insert Parse Default 92	centre 57
Insert Word Abbrevs 87	count for page 65
inserting text commands 12, 45	count for region 40
inserting text functions 136	delete blank 42
insert-string 136	delete matching 69
Interrupt Shell Subjob 97	delete non-matching 70
Inverse Add Global Word Abbrev 83	end 33
Inverse Add Mode Word Abbrev 82	goto 34
т	indent new 55
J	kill 44
Jump to Saved Position 76	kill backward 44
Just One Space 42	length 56
**	list matching 69
K	move to top of window 36
key	next 33
command description 19	open new 46

previous 33	exchange with point 39
transposition 51	form 112
line count 65	move current point to 39
Line to Top of Window 36	paragraph 40
line-end 137	pop 39
LINEFEED Auto Fill Linefeed 58	sentence 39
line-offset 138	set 38
line-start 137	Mark Defun 103
Lisp	Mark Form 112
editor commands 101	Mark Page 65
Lisp comment commands 117	Mark Paragraph 40
Lisp editor functions 137	mark ring 38
Lisp form commands 112	Mark Sentence 39
Lisp Insert) 119	Mark Whole Buffer 40
Lisp list commands 115	message 134
Lisp Mode 79	Meta key 9
Lisp mode 78	Meta-! Shell Command 95
Lisp symbol documentation	Meta+(Insert () 119
commands 120	Meta+) Move Over) 119
list	Meta+, Continue Tags Search 107
extract 117	Meta+. Find Source 104
move down one level 116	Meta+/ Dynamic Completion 47
move to end 115, 116	Meta+; Indent for Comment 117
move to start 116	Meta+< Beginning of Buffer 36
List Buffers 60	Meta+= Function Arglist 110
List Callees 109	Meta+> End of Buffer 37
List Callers 109	Meta+? Find Tag 106
list commands 115	Meta+[Backward Paragraph 34
List Matching Lines 69	Meta+\ Delete Horizontal Space 42
List Registers 77	Meta+] Forward Paragraph 34
List Word Abbrevs 85	Meta+' Word Abbrev Prefix Point 84
Load File 123	Meta+A Backward Sentence 34
Lowercase Region 49	Meta+B Backward Word 33
Lowercase Word 48	Meta+B Echo Area Backward Word 91
Lowercuse Word 10	Meta+C Capitalize Word 49
M	Meta+Ctrl+; Kill Comment 118
	Meta+Ctrl+\ Indent Region 53
macro	Meta+Ctrl+A Beginning of Defun 102
keyboard 87	Meta+Ctrl+B Backward Form 112
Macroexpand Form 114	Meta+Ctrl+C, break current process 16
macro-expansion 114	Meta+Ctrl+D Down List 116
macros	Meta+Ctrl+Delete Backward Kill
defcommand 130	Form 113
major mode	
editor definition 8, 78	Meta+Ctrl+E End of Defun 102
Make Word Abbrev 83	Meta+Ctrl+F Forward Form 112
make-buffer 133	Meta+Ctrl+H Mark Defun 103
man Unix command 22	Meta+Ctrl+I Complete Symbol 111
manual	Meta+Ctrl+K Forward Kill Form 113
on-line editor 19, 20, 21	Meta+Ctrl+L Select Previous
Manual Entry 22	Buffer 59
mark	Meta+Ctrl+N Forward List 115
editor definition 8	Meta+Ctrl+P Backward List 116

110	J. R
Meta+Ctrl+Q Indent Form 113	mode line editor definition 6
Meta+Ctrl+R Regexp Reverse Search 74	modes
Meta+Ctrl+S Regexp Forward Search 73	
Meta+Ctrl+Shift+A Show Documentation	abbrev 79, 82
Function 120	auto-fill 57, 79
Meta+Ctrl+Shift+C, break selected	execute 79
process 16	fundamental 78
Meta+Ctrl+Shift+V Show Documentation	Lisp 78
Variable 120	overwrite 52, 79 shell 78
Meta+Ctrl+Space Pop Mark 39	
Meta+Ctrl+T Transpose Forms 115	text 78
Meta+Ctrl+U Backward Up List 116	mouse
Meta+Ctrl+X Evaluate Defun 122	editor bindings 98
Meta+Ctrl+Z Exit Recursive Edit 93	Move Over) 119
Meta+D Kill Next Word 43	movement functions, 127, 142
Meta+Delete Echo Area Kill Previous	movement functions 137, 143
Word 91	move-point 140
Meta+Delete Kill Previous Word 43	NT
Meta+E Forward Sentence 34	N
Meta+F Forward Word 32	Name Keyboard Macro 88
Meta+G Fill Region 56	Negative Argument 23
Meta+H Mark Paragraph 40	New Line 46
Meta+J Indent New Comment Line 118 Meta+K Forward Kill Sentence 44	New Window 62
Meta+L Lowercase Word 48	newline
Meta+M Back to Indentation 54	adding to end of file 27
Meta+N Down Comment Line 118	Next Line 33
Meta+N Next Parse 90	Next Ordinary Window 62
Meta+Newline Indent New Comment	Next Page 65
Line 118	Next Parse 90
Meta+P Previous Parse 90	Next Window 62
Meta+P Up Comment Line 118	
Meta+Q Fill Paragraph 55	0
Meta+Shift+% Query Replace 74	Open Line 46
Meta+Shift+ Delete Indentation 54	Overwrite Delete Previous Character 52
Meta+Shift+~ Buffer Not Modified 62	Overwrite Mode 52
Meta+Shift+M Walk Form 114	overwrite mode 52, 79
Meta+Space Just One Space 42	overwriting commands 51
Meta+T Transpose Words 50	
Meta+Tab Expand File Name 47	P
Meta+U Uppercase Word 49	package
Meta+V Scroll Window Up 35	editor 130
Meta+W Save Region 45	set 121
Meta+X Extended Command 10, 17	page
Meta+Y Rotate Kill Ring 45	display first lines 66
minor mode	editor definition 64
editor definition 8, 79	goto 65
mode	insert first lines into buffer 66
editor definition 8, 77	mark 65
indentation in 52	next 65
mode abbreviation	previous 64
editor definition 82	page commands 64

pane	${f Q}$
editor definition 5	Query Replace 74
paragraph	query replace 74
backward 34	directory 75
editor definition 9	system 76
fill 55	tags 107
forward 34	Quote Tab 55
mark 40	Quoted Insert 46
parentheses commands 119	4 40104 2115011 10
point	R
editor definition 7	=-
exchange with mark 39	Read Word Abbrev File 86
goto 37	recursive editing 93
position of 37	redisplay 143 Re-evaluate Defvar 122
save to register 76	
where is 37	Refresh Screen 64
point functions 138	Regexp Forward Search 73
point ring, see mark ring	Regexp Reverse Search 74
point< 139	region
point> 139	capitalize 50
points-to-string 137	compile 124
Pop and Goto Mark 39	delete 43
Pop Mark 39	determining 39
prefix	editor definition 8 evaluate 122
fill 56	fill 56
prefix argument 11, 22	
Prefix-Argument-Default 23	get from register 77
Previous Line 33	indent 53
Previous Page 64	indent rigidly 54 kill 44
Previous Parse 90	line count 40
Previous Window 63	lowercase 49
print	print 41
buffer 61	save 45
file 30	save 43
region 41	transposition 51
Print Buffer 61	uppercase 49
Print File 30	word count 40
Print Region 41	write 26
process	Region-Query-Size 41
breaking 16	register
Process File Options 31	editor definition 76
programming the editor 130	get region 77
examples 143	kill 77
prompt functions 140	list 77
prompt-for-buffer 141	move to saved position 76
prompt-for-file 141	save current point to 76
prompt-for-integer 141	save editient point to 70
prompt-for-string 142	register commands 76
prompt-for-variable 142	regular expression 72
Put Register 77	regular expression 72 regular expression search 72
	Rename Buffer 61

Rename File 31	regular expression 72
repeating a command 11, 22	system 70
replace	Search All Buffers 70
case sensitivity 76	search-files function 71
query 74	searching 66
string 74	searching commands 66
Replace String 74	Select Buffer 59
replacing 74	Select Buffer Other Window 59
replacing commands 66	Select Previous Buffer 59
RETURN Auto Fill Return 58	Self Insert 47
RETURN Confirm Parse 89	Self Overwrite 52
Return Default 92	sentence
RETURN New Line 46	backward 34
Reverse Incremental Search 67	delimiter 9
Reverse Search 69	editor definition 9
Revert File 30	forward 34
ring	kill backward 44
history 90	kill forward 44
kill 41, 43, 45	mark 39
mark 38	terminator 9
undo 47	Set Buffer Output 121
window 62	Set Buffer Package 121
Ring-Size 48	Set Buffer Read-Only 61
Room 100	Set Comment Column 117
Rotate Kill Ring 45	Set Fill Column 56
Run Command 95	Set Fill Prefix 57
	Set Mark 38
S	Set Prefix Argument 22
same-line-p 140	Set Variable 93
Save All Files 26	setup-indent 129
Save All Files and Exit 27	Shell 95
Save File 25	Shell Command 95
Save Position 76	shell command
Save Region 45	from editor 95
save-excursion 140	shell mode 78
screen	Shell Send Eof 97
refresh 64	Shell-cd-RegExp 96
Scroll Next Window Down 63	Shell-pop-RegExp 97
Scroll Next Window Up 63	Shell-push-RegExp 96
Scroll Window Down 35	Show Documentation 120
Scroll Window Up 35	Show Documentation Class 121
Scroll-Overlap 36	Show Documentation Function 120
search	Show Documentation Variable 120
all buffers 70	Show Paths From 110
backward 69	Show Paths To 109
case sensitivity 71	Show Variable 92
directory 70	Skip Whitespace 37
forward 68	space
incremental backward 67	delete horizontal 42
incremental forward 67	just one 42
regexp backward 74	SPACE Auto Fill Space 58
regexp forward 73	SPACE Complete Field 89
O	

Spaces-For-Tab 53 start-line-p 140	tracing functions 107 Transpose Characters 50
Stop Shell Subjob 97	Transpose Forms 115
string	Transpose Lines 51
count occurrences of 71	Transpose Regions 51
insert 136	Transpose Words 50
replace 74	transposition commands 50
search 66	
symbol	U
completion 111	Undo 48
describe 120	undo ring 47
documentation 120	size 48
symbol documentation commands 120	undoing editor commands 13, 47
system	Unexpand Last Word 84
compile 126	Unix command
compile changed definitions 126	man 22
evaluate changed definitions 124	Un-Kill 45
query replace 76	Untrace Definition 109
search 70	Untrace Function 108
System Query Replace 76	Up Comment Line 118
System Search 70	Uppercase Region 49
_	Uppercase Word 49
T	use-buffer 133
TAB	
for command completion 89	V
for indentation 53	variable
Tab	change value of 93
for command completion 10, 17	description 20, 21
tab	documentation for global 120
insert 55	editor 92
width 53	listing with apropos 18
TAB Complete Input 89	show value of 92
TAB Indent 53	variable functions 142
tag	variable-value 142
continue search 107	variable-value-if-bound 143
create buffer 105	View Page Directory 66
editor definition 103	View Source Search 105
find 106	Visit File 25
query replace 107	Visit Tags File 107
search 106	o .
visit file 107	W
Tags Query Replace 107	Walk Form 114
Tags Search 106	Wfind File 25
terminator	What Command 19
sentence 9	What Lossage 21
text handling concepts 9	Where Is 21
Text Mode 79	Where Is Point 37
text mode 78	whitespace
Toggle Error Catch 123	skip 37
Top of Window 36	window
Trace Definition 108 Trace Function 108	delete 63
HACE FUHCHOH 100	

```
delete next 63
  editor definition 5
  move line to top of 36
  move to bottom 36
  move to top 36
  new 62
  next 62
  previous 63
  scroll down 35
  scroll next down 63
  scroll next up 63
  scroll overlap 36
  scroll up 35
window commands 62
window functions 143
window ring 62
window-buffer 133
windows
  and the Editor 98
  copy 98
  paste 98
word
  backward 33
  capitalize 49
  count for region 40
  dynamic completion 47
  editor definition 9
  forward 32
  kill next 43
  kill previous 43
  lowercase 48
  transposition 50
  uppercase 49
Word Abbrev Apropos 85
Word Abbrev Prefix Point 84
word-offset 138
Write File 26
Write Region 26
Write Word Abbrev File 86
Y
```

yank 45

Index