

# **IDLWAVE User Manual**

---

Emacs major mode and shell for IDL  
Edition 6.0, Feb, 2006

by **J.D. Smith & Carsten Dominik**

---

This is edition 6.0 of the *IDLWAVE User Manual* for IDLWAVE version 6.0, Feb, 2006.

Copyright © 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, with the Front-Cover texts being “A GNU Manual”, and with the Back-Cover Texts as in (a) below. A copy of the license is included in the section entitled “GNU Free Documentation License” in the Emacs manual.

(a) The FSF’s Back-Cover Text is: “You have freedom to copy and modify this GNU Manual, like GNU software. Copies published by the Free Software Foundation raise funds for GNU development.”

This document is part of a collection distributed under the GNU Free Documentation License. If you want to distribute this document separately from the collection, you can do so by adding a copy of the license to the document, as described in section 6 of the license.

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>IDLWAVE in a Nutshell</b>	<b>3</b>
<b>3</b>	<b>Getting Started (Tutorial)</b>	<b>4</b>
3.1	Lesson I: Development Cycle	4
3.2	Lesson II: Customization	6
3.3	Lesson III: User and Library Catalogs	8
<b>4</b>	<b>The IDLWAVE Major Mode</b>	<b>9</b>
4.1	Code Formatting	9
4.1.1	Code Indentation	9
4.1.2	Continued Statement Indentation	9
4.1.3	Comment Indentation	10
4.1.4	Continuation Lines and Filling	11
4.1.5	Syntax Highlighting	12
4.1.6	Octals and Highlighting	12
4.2	Routine Info	12
4.3	Online Help	14
4.3.1	Help with HTML Documentation	15
4.3.2	Help with Source	17
4.4	Completion	18
4.4.1	Case of Completed Words	19
4.4.2	Object Method Completion and Class Ambiguity	19
4.4.3	Object Method Completion in the Shell	20
4.4.4	Class and Keyword Inheritance	20
4.4.5	Structure Tag Completion	21
4.5	Routine Source	22
4.6	Resolving Routines	22
4.7	Code Templates	22
4.8	Abbreviations	22
4.9	Actions	24
4.9.1	Block Boundary Check	25
4.9.2	Padding Operators	25
4.9.3	Case Changes	26
4.10	Documentation Header	27
4.11	Motion Commands	27
4.12	Miscellaneous Options	28

<b>5</b>	<b>The IDLWAVE Shell</b>	<b>29</b>
5.1	Starting the Shell	29
5.2	Using the Shell	30
5.3	Commands Sent to the Shell	32
5.4	Debugging IDL Programs	32
5.4.1	A Tale of Two Modes	32
5.4.2	Debug Key Bindings	33
5.4.3	Breakpoints and Stepping	33
5.4.4	Compiling Programs	35
5.4.5	Walking the Calling Stack	35
5.4.6	Electric Debug Mode	36
5.5	Examining Variables	38
5.6	Custom Expression Examination	39
<b>6</b>	<b>Installation</b>	<b>41</b>
6.1	Installing IDLWAVE	41
6.2	Installing Online Help	41
<b>7</b>	<b>Acknowledgements</b>	<b>42</b>
<b>Appendix A</b>	<b>Sources of Routine Info</b>	<b>43</b>
A.1	Routine Definitions	43
A.2	Routine Information Sources	43
A.3	Catalogs	44
A.3.1	Library Catalogs	45
A.3.2	User Catalog	46
A.4	Load-Path Shadows	47
A.5	Documentation Scan	48
<b>Appendix B</b>	<b>HTML Help Browser Tips</b>	<b>49</b>
<b>Appendix C</b>	<b>Configuration Examples</b>	<b>50</b>
<b>Appendix D</b>	<b>Windows and MacOS</b>	<b>53</b>
<b>Appendix E</b>	<b>Troubleshooting</b>	<b>54</b>
<b>Index</b>		<b>57</b>

# 1 Introduction

IDLWAVE is a package which supports editing source files written in the Interactive Data Language (IDL<sup>1</sup>), and running IDL as an inferior shell<sup>2</sup>. It is a feature-rich replacement for the IDLDE development environment included with IDL, and uses the full power of Emacs to make editing and running IDL programs easier, quicker, and more structured.

IDLWAVE consists of two main parts: a major mode for editing IDL source files (`idlwave-mode`) and a mode for running the IDL program as an inferior shell (`idlwave-shell-mode`). Although one mode can be used without the other, both work together closely to form a complete development environment. Here is a brief summary of what IDLWAVE does:

- Smart code indentation and automatic-formatting.
- Three level syntax highlighting support.
- Context-sensitive display of calling sequences and keywords for more than 1000 native IDL routines, extendible to any additional number of local routines, and already available with many pre-scanned libraries.
- Fast, context-sensitive online HTML help, or source-header help for undocumented routines.
- Context sensitive completion of routine names, keywords, system variables, class names and much more.
- Easy insertion of code templates and abbreviations of common constructs.
- Automatic corrections to enforce a variety of customizable coding standards.
- Integrity checks and auto-termination of logical blocks.
- Routine name space conflict search with likelihood-of-use ranking.
- Support for ‘`imenu`’ (Emacs) and ‘`func-menu`’ (XEmacs).
- Documentation support.
- Running IDL as an inferior Shell with history search, command line editing and all the completion and routine info capabilities present in IDL source buffers.
- Full handling of debugging with breakpoints, with interactive setting of break conditions, and easy stepping through code.
- Compilation, execution and interactive single-keystroke debugging of programs directly from the source buffer.
- Quick, source-guided navigation of the calling stack, with variable inspection, etc.
- Examining variables and expressions with a mouse click.
- And much, much more...

IDLWAVE is the distant successor to the ‘`idl.el`’ and ‘`idl-shell.el`’ files written by Chris Chase. The modes and files had to be renamed because of a name space conflict with CORBA’s `idl-mode`, defined in Emacs in the file ‘`cc-mode.el`’.

---

<sup>1</sup> IDL is a registered trademark of Research Systems, Inc.

<sup>2</sup> IDLWAVE can also be used for editing source files for the related WAVE/CL language, but with only limited support.

In this manual, each section ends with a list of related user options. Don't be confused by the sheer number of options available — in most cases the default settings are just fine. The variables are listed here to make sure you know where to look if you want to change anything. For a full description of what a particular variable does and how to configure it, see the documentation string of that variable (available with `C-h v`). Some configuration examples are also given in the appendix.

## 2 IDLWAVE in a Nutshell

### Editing IDL Programs

<code>(TAB)</code>	Indent the current line relative to context.
<code>C-M-\</code>	Re-indent all lines in the current region.
<code>C-M-q</code>	Re-indent all lines in the current routine.
<code>C-u (TAB)</code>	Re-indent all lines in the current statement.
<code>M-(RET)</code>	Start a continuation line, splitting the current line at point.
<code>M-q</code>	Fill the current comment paragraph.
<code>C-c ?</code>	Display calling sequence and keywords for the procedure or function call at point.
<code>M-?</code>	Load context sensitive online help for nearby routine, keyword, etc.
<code>M-(TAB)</code>	Complete a procedure name, function name or keyword in the buffer.
<code>C-c C-i</code>	Update IDLWAVE's knowledge about functions and procedures.
<code>C-c C-v</code>	Visit the source code of a procedure/function.
<code>C-u C-c C-v</code>	Visit the source code of a procedure/function in this buffer.
<code>C-c C-h</code>	Insert a standard documentation header.
<code>C-c (RET)</code>	Insert a new timestamp and history item in the documentation header.

### Running the IDLWAVE Shell, Debugging Programs

<code>C-c C-s</code>	Start IDL as a subprocess and/or switch to the shell buffer.
<code>(Up), M-p</code>	Cycle back through IDL command history.
<code>(Down), M-n</code>	Cycle forward.
<code>(TAB)</code>	Complete a procedure name, function name or keyword in the shell buffer.
<code>C-c C-d C-c</code>	Save and compile the source file in the current buffer.
<code>C-c C-d C-x</code>	Go to next syntax error.
<code>C-c C-d C-v</code>	Switch to electric debug mode.
<code>C-c C-d C-b</code>	Set a breakpoint at the nearest viable source line.
<code>C-c C-d C-d</code>	Clear the nearest breakpoint.
<code>C-c C-d [</code>	Go to the previous breakpoint.
<code>C-c C-d ]</code>	Go to the next breakpoint.
<code>C-c C-d C-p</code>	Print the value of the expression near point in IDL.

### Commonly used Settings in '.emacs'

```
;; Change the indentation preferences
;; Start autoloading routine info after 2 idle seconds
(setq idlwave-init-rinfo-when-idle-after 2)
;; Pad operators with spaces
(setq idlwave-do-actions t
      idlwave-surround-by-blank t)
;; Syntax Highlighting
(add-hook 'idlwave-mode-hook 'turn-on-font-lock)
;; Automatically start the shell when needed
(setq idlwave-shell-automatic-start t)
;; Bind debugging commands with CONTROL and SHIFT modifiers
(setq idlwave-shell-debug-modifiers '(control shift))
```

## 3 Getting Started (Tutorial)

### 3.1 Lesson I: Development Cycle

The purpose of this tutorial is to guide you through a very basic development cycle using IDLWAVE. We will paste a simple program into a buffer and use the shell to compile, debug and run it. On the way we will use many of the important IDLWAVE commands. Note, however, that IDLWAVE has many more capabilities than covered here, which can be discovered by reading the entire manual, or hovering over the shoulder of your nearest IDLWAVE guru for a few days.

It is assumed that you have access to Emacs or XEmacs with the full IDLWAVE package including online help (see [Chapter 6 \[Installation\], page 41](#)). We also assume that you are familiar with Emacs and can read the nomenclature of key presses in Emacs (in particular, *C* stands for `CONTROL` and *M* for `META` (often the `ALT` key carries this functionality)).

Open a new source file by typing:

```
C-x C-f tutorial.pro RET
```

A buffer for this file will pop up, and it should be in IDLWAVE mode, indicated in the mode line just below the editing window. Also, the menu bar should contain 'IDLWAVE'.

Now cut-and-paste the following code, also available as 'tutorial.pro' in the IDLWAVE distribution.

```
function daynr,d,m,y
  ;; compute a sequence number for a date
  ;; works 1901-2099.
  if y lt 100 then y = y+1900
  if m le 2 then delta = 1 else delta = 0
  m1 = m + delta*12 + 1
  y1 = y * delta
  return, d + floor(m1*30.6)+floor(y1*365.25)+5
end

function weekday,day,month,year
  ;; compute weekday number for date
  nr = daynr(day,month,year)
  return, nr mod 7
end

pro plot_wday,day,month
  ;; Plot the weekday of a date in the first 10 years of this century.
  years = 2000,+indgen(10)
  wdays = intarr(10)
  for i=0,n_elements(wdays)-1 do begin
    wdays[i] = weekday(day,month,years[i])
  end
  plot,years,wdays,YS=2,YT="Wday (0=Sunday)"
end
```

The indentation probably looks funny, since it's different from the settings you use, so use the `(TAB)` key in each line to automatically line it up (or, more quickly, *select* the entire buffer with `C-x h`, and indent the whole region with `C-M-\`). Notice how different syntactical elements are highlighted in different colors, if you have set up support for font-lock.

Let's check out two particular editing features of IDLWAVE. Place the cursor after the `end` statement of the `for` loop and press `(SPC)`. IDLWAVE blinks back to the beginning of the block and changes the generic `end` to the specific `endfor` automatically (as long as the variable `idlwave-expand-generic-end` is turned on — see [Section 3.2 \[Lesson II – Customization\]](#), page 6). Now place the cursor in any line you would like to split and press `M-(RET)`. The line is split at the cursor position, with the continuation '\$' and indentation all taken care of. Use `C-/` to undo the last change.

The procedure `plot_wday` is supposed to plot the day of the week of a given date for the first 10 years of the 21st century. As in most code, there are a few bugs, which we are going to use IDLWAVE to help us fix.

First, let's launch the IDLWAVE shell. You do this with the command `C-c C-s`. The Emacs window will split or another window will popup to display IDL running in a shell interaction buffer. Type a few commands like `print, !PI` to convince yourself that you can work there just as well as in a terminal, or the IDLDE. Use the arrow keys to cycle through your command history. Are we having fun now?

Now go back to the source window and type `C-c C-d C-c` to compile the program. If you watch the shell buffer, you see that IDLWAVE types `.run "tutorial.pro"` for you. But the compilation fails because there is a comma in the line `years=...`. The line with the error is highlighted and the cursor positioned at the error, so remove the comma (you should only need to hit *Delete!*). Compile again, using the same keystrokes as before. Notice that the file is automatically saved for you. This time everything should work fine, and you should see the three routines compile.

Now we want to use the command to plot the day of the week on January 1st. We could type the full command ourselves, but why do that? Go back to the shell window, type `'plot_'` and hit `(TAB)`. After a bit of a delay (while IDLWAVE initializes its routine info database, if necessary), the window will split to show all procedures it knows starting with that string, and `plot_wday` should be one of them. Saving the buffer alerted IDLWAVE about this new routine. Click with the middle mouse button on `plot_wday` and it will be copied to the shell buffer, or if you prefer, add `'w'` to `'plot_'` to make it unambiguous (depending on what other routines starting with `'plot_'` you have installed on your system), hit `(TAB)` again, and the full routine name will be completed. Now provide the two arguments:

```
plot_wday, 1, 1
```

and press `(RET)`. This fails with an error message telling you the `YT` keyword to plot is ambiguous. What are the allowed keywords again? Go back to the source window and put the cursor into the `'plot'` line and press `C-c ?`. This shows the routine info window for the `plot` routine, which contains a list of keywords, along with the argument list. Oh, we wanted `YTITLE`. Fix that up. Recompile with `C-c C-d C-c`. Jump back into the shell with `C-c C-s`, press the `(UP)` arrow to recall the previous command and execute again.

This time we get a plot, but it is pretty ugly — the points are all connected with a line. Hmm, isn't there a way for `plot` to use symbols instead? What was that keyword? Position

the cursor on the plot line after a comma (where you'd normally type a keyword), and hit  $M-\overline{\text{Tab}}$ . A long list of plot's keywords appears. Aha, there it is, `PSYM`. Middle click to insert it. An '=' sign is included for you too. Now what were the values of `PSYM` supposed to be? With the cursor on or after the keyword, press  $M-?$  for online help (alternatively, you could have right clicked on the colored keyword itself in the completion list). A browser will pop up showing the HTML documentation for the `PYSM` keyword. OK, let's use `diamonds=4`. Fix this, recompile (you know the command by now:  $C-c C-d C-c$ ), go back to the shell (if it's vanished, you know what to do:  $C-c C-s$ ) and execute again. Now things look pretty good.

Let's try a different day — how about April fool's day?

```
plot_wday,1,4
```

Oops, this looks very wrong. All April Fool's days cannot be Fridays! We've got a bug in the program, perhaps in the `daynr` function. Let's put a breakpoint on the last line there. Position the cursor on the `'return, d+...'` line and press  $C-c C-d C-b$ . IDL sets a breakpoint (as you see in the shell window), and the break line is indicated. Back to the shell buffer, re-execute the previous command. IDL stops at the line with the breakpoint. Now hold down the SHIFT key and click with the middle mouse button on a few variables there: `'d'`, `'y'`, `'m'`, `'y1'`, etc. Maybe `d` isn't the correct type. CONTROL-SHIFT middle-click on it for help. Well, it's an integer, so that's not the problem. Aha, `'y1'` is zero, but it should be the year, depending on `delta`. Shift click `'delta'` to see that it's 0. Below, we see the offending line: `'y1=y*delta...'` the multiplication should have been a minus sign! Hit `q` to exit the debugging mode, and fix the line to read:

```
y1 = y - delta
```

Now remove all breakpoints:  $C-c C-d C-a$ . Recompile and rerun the command. Everything should now work fine. How about those leap years? Change the code to plot 100 years and see that every 28 years, the sequence of weekdays repeats.

## 3.2 Lesson II: Customization

Emacs is probably the most customizable piece of software ever written, and it would be a shame if you did not make use of this to adapt IDLWAVE to your own preferences. Customizing Emacs or IDLWAVE is accomplished by setting Lisp variables in the `'.emacs'` file in your home directory — but do not be dismayed; for the most part, you can just copy and work from the examples given here.

Let's first use a boolean variable. These are variables which you turn on or off, much like a checkbox. A value of `'t'` means on, a value of `'nil'` means off. Copy the following line into your `'.emacs'` file, exit and restart Emacs.

```
(setq idlwave-reserved-word-upcase t)
```

When this option is turned on, each reserved word you type into an IDL source buffer will be converted to upper case when you press  $\overline{\text{SPC}}$  or  $\overline{\text{RET}}$  right after the word. Try it out! `'if'` changes to `'IF'`, `'begin'` to `'BEGIN'`. If you don't like this behavior, remove the option again from your `'.emacs'` file and restart Emacs.

You likely have your own indentation preferences for IDL code. For example, some may prefer to indent the main block of an IDL program slightly from the margin and use only 3 spaces as indentation between `BEGIN` and `END`. Try the following lines in `'.emacs'`:

```
(setq idlwave-main-block-indent 1)
(setq idlwave-block-indent 3)
(setq idlwave-end-offset -3)
```

Restart Emacs, and re-indent the program we developed in the first part of this tutorial with `C-c h` and `C-M-\`. You may want to keep these lines in `.emacs`, with values adjusted to your likings. If you want to get more information about any of these variables, type, e.g., `C-h v idlwave-main-block-indent` (`RET`). To find which variables can be customized, look for items marked ‘User Option:’ throughout this manual.

If you cannot seem to master this Lisp customization in `.emacs`, there is another, more user-friendly way to customize all the IDLWAVE variables. You can access it through the IDLWAVE menu in one of the `.pro` buffers, menu item `Customize->Browse IDLWAVE Group`. Here you’ll be presented with all the various variables grouped into categories. You can navigate the hierarchy (e.g. `IDLWAVE Code Formatting->Idlwave Abbrev And Indent Action->Idlwave Expand Generic End` to turn on END expansion), read about the variables, change them, and ‘Save for Future Sessions’. Few of these variables need customization, but you can exercise considerable control over IDLWAVE’s functionality with them.

You may also find the key bindings used for the debugging commands too long and complicated. Often we have heard complaints along the lines of, “Do I really have to go through the finger gymnastics of `C-c C-d C-c` to run a simple command?” Due to Emacs rules and conventions, shorter bindings cannot be set by default, but you can easily enable them. First, there is a way to assign all debugging commands in a single sweep to another simpler combination. The only problem is that we have to use something which Emacs does not need for other important commands. One good option is to execute debugging commands by holding down `CONTROL` and `SHIFT` while pressing a single character: `C-S-b` for setting a breakpoint, `C-S-c` for compiling the current source file, `C-S-a` for deleting all breakpoints (try it, it’s easier). You can enable this with:

```
(setq idlwave-shell-debug-modifiers '(shift control))
```

If you have a special keyboard with, for example, a `SUPER` key, you could even shorten that:

```
(setq idlwave-shell-debug-modifiers '(super))
```

to get compilation on `S-c`. Often, a modifier key like `SUPER` or `HYPHER` is bound or can be bound to an otherwise unused key on your keyboard — consult your system documentation.

You can also assign specific commands to keys. This you must do in the `mode-hook`, a special function which is run when a new IDLWAVE buffer gets set up. The possibilities for key customization are endless. Here we set function keys `f4-f8` to common debugging commands.

```
;; First for the source buffer
(add-hook 'idlwave-mode-hook
  (lambda ()
    (local-set-key [f4] 'idlwave-shell-retall)
    (local-set-key [f5] 'idlwave-shell-break-here)
    (local-set-key [f6] 'idlwave-shell-clear-current-bp)
    (local-set-key [f7] 'idlwave-shell-cont)
    (local-set-key [f8] 'idlwave-shell-clear-all-bp)))
```

```
;; Then for the shell buffer
(add-hook 'idlwave-shell-mode-hook
  (lambda ()
    (local-set-key [f4] 'idlwave-shell-retall)
    (local-set-key [f5] 'idlwave-shell-break-here)
    (local-set-key [f6] 'idlwave-shell-clear-current-bp)
    (local-set-key [f7] 'idlwave-shell-cont)
    (local-set-key [f8] 'idlwave-shell-clear-all-bp)))
```

### 3.3 Lesson III: User and Library Catalogs

We have already used the routine `info` display in the first part of this tutorial. This was invoked using `C-c ?`, and displays information about the IDL routine near the cursor position. Wouldn't it be nice to have the same kind of information available for your own routines and for the huge amount of code in major libraries like JHUP or the IDL-Astro library? In many cases, you may already have this information. Files named `.idlwave_catalog` in library directories contain scanned information on the routines in that directory; many popular libraries ship with these "library catalogs" pre-scanned. Users can scan their own routines in one of two ways: either using the supplied tool to scan directories and build their own `.idlwave_catalog` files, or using the built-in method to create a single "user catalog", which we'll show here. See [Section A.3 \[Catalogs\]](#), page 44, for more information on choosing which method to use.

To build a user catalog, select `Routine Info/Select Catalog Directories` from the IDLWAVE entry in the menu bar. If necessary, start the shell first with `C-c C-s` (see [Section 5.1 \[Starting the Shell\]](#), page 29). IDLWAVE will find out about the IDL `!PATH` variable and offer a list of directories on the path. Simply select them all (or whichever you want — directories with existing library catalogs will not be selected by default) and click on the `'Scan&Save'` button. Then go for a cup of coffee while IDLWAVE collects information for each and every IDL routine on your search path. All this information is written to the file `.idlwave/idlusercat.el` in your home directory and will from now on automatically load whenever you use IDLWAVE. You may find it necessary to rebuild the catalog on occasion as your local libraries change, or build a library catalog for those directories instead. Invoke routine `info` (`C-c ?`) or completion (`M-(TAB)`) on any routine or partial routine name you know to be located in the library. E.g., if you have scanned the IDL-Astro library:

```
a=readf(M-(TAB))
```

expands to `'readfits'`. Then try

```
a=readfits(C-c?)
```

and you get:

```
Usage:      Result = READFITS(filename, header, heap)
```

```
...
```

I hope you made it until here. Now you are set to work with IDLWAVE. On the way you will want to change other things, and to learn more about the possibilities not discussed in this short tutorial. Read the manual, look at the documentation strings of interesting variables (with `C-h v idlwave<-variable-name> (RET)`) and ask the remaining questions on the newsgroup `comp.lang.idl-pvwave`.

## 4 The IDLWAVE Major Mode

The IDLWAVE major mode supports editing IDL source files. In this chapter we describe the main features of the mode and how to customize them.

### 4.1 Code Formatting

The IDL language, with its early roots in FORTRAN, modern implementation in C, and liberal borrowing of features of many vector and other languages along its 25+ year history, has inherited an unusual mix of syntax elements. Left to his or her own devices, a novice IDL programmer will often conjure code which is very difficult to read and impossible to adapt. Much can be gleaned from studying available IDL code libraries for coding style pointers, but, due to the variety of IDL syntax elements, replicating this style can be challenging at best. Luckily, IDLWAVE understands the structure of IDL code very well, and takes care of almost all formatting issues for you. After configuring it to match your coding standards, you can rely on it to help keep your code neat and organized.

#### 4.1.1 Code Indentation

Like all Emacs programming modes, IDLWAVE performs code indentation. The `(TAB)` key indents the current line relative to context. `(LFD)` insert a newline and indents the new line. The indentation is governed by a number of variables. IDLWAVE indents blocks (between `PRO/FUNCTION/BEGIN` and `END`), and continuation lines.

To re-indent a larger portion of code (e.g. when working with foreign code written with different conventions), use `C-M-\ (indent-region)` after marking the relevant code. Useful marking commands are `C-x h` (the entire file) or `C-M-h` (the current subprogram). The command `C-M-q` reindents the entire current routine. See [Section 4.9 \[Actions\], page 24](#), for information how to impose additional formatting conventions on foreign code.

`idlwave-main-block-indent` (2) [User Option]

Extra indentation for the main block of code. That is the block between the `FUNCTION/PRO` statement and the `END` statement for that program unit.

`idlwave-block-indent` (3) [User Option]

Extra indentation applied to block lines. If you change this, you probably also want to change `idlwave-end-offset`.

`idlwave-end-offset` (-3) [User Option]

Extra indentation applied to block `END` lines. A value equal to negative `idlwave-block-indent` will make `END` lines line up with the block `BEGIN` lines.

#### 4.1.2 Continued Statement Indentation

Continuation lines (following a line ending with `$`) can receive a fixed indentation offset from the main level, but in several situations IDLWAVE can use a special form of indentation which aligns continued statements more naturally. Special indentation is calculated for continued routine definition statements and calls, enclosing parentheses (like function calls, structure/class definitions, explicit structures or lists, etc.), and continued assignments. An attempt is made to line up with the first non-whitespace character after the relevant opening punctuation mark (`,`, `(`, `{`, `[`, `=`). For lines without any non-comment characters on the line

with the opening punctuation, the continued line(s) are aligned just past the punctuation. An example:

```
function foo, a, b, $
      c, d
  bar = sin( a + b + $
           c + d)
end
```

The only drawback to this special continued statement indentation is that it consumes more space, e.g., for long function names or left hand sides of an assignment:

```
function thisfunctionnameisverylongsoitwillleavelittleroom, a, b, $
      c, d
```

You can instruct IDLWAVE when to avoid using this special continuation indentation by setting the variable `idlwave-max-extra-continuation-indent`, which specifies the maximum additional indentation beyond the basic indent to be tolerated, otherwise defaulting to a fixed-offset from the enclosing indent (the size of which offset is set in `idlwave-continuation-indent`). As a special case, continuations of routine calls without any arguments or keywords will *not* align the continued line, under the assumption that you continued because you needed the space.

Also, since the indentation level can be somewhat dynamic in continued statements with special continuation indentation, especially if `idlwave-max-extra-continuation-indent` is small, the key `C-u TAB` will re-indent all lines in the current statement. Note that `idlwave-indent-to-open-paren`, if non-`nil`, overrides the `idlwave-max-extra-continuation-indent` limit, for parentheses only, forcing them always to line up.

`idlwave-continuation-indent` (2) [User Option]

Extra indentation applied to normal continuation lines.

`idlwave-max-extra-continuation-indent` (20) [User Option]

The maximum additional indentation (over the basic continuation-indent) that will be permitted for special continues. To effectively disable special continuation indentation, set to 0. To enable it constantly, set to a large number (like 100). Note that the indentation in a long continued statement never decreases from line to line, outside of nested parentheses statements.

`idlwave-indent-to-open-paren` (t) [User Option]

Non-`nil` means indent continuation lines to innermost open parenthesis, regardless of whether the `idlwave-max-extra-continuation-indent` limit is satisfied.

### 4.1.3 Comment Indentation

In IDL, lines starting with a ‘;’ are called *comment lines*. Comment lines are indented as follows:

```
;;;      The indentation of lines starting with three semicolons remains unchanged.
;;       Lines starting with two semicolons are indented like the surrounding code.
;        Lines starting with a single semicolon are indented to a minimum column.
```

The indentation of comments starting in column 0 is never changed.

`idlwave-no-change-comment` [User Option]  
The indentation of a comment starting with this regexp will not be changed.

`idlwave-begin-line-comment` [User Option]  
A comment anchored at the beginning of line.

`idlwave-code-comment` [User Option]  
A comment that starts with this regexp is indented as if it is a part of IDL code.

#### 4.1.4 Continuation Lines and Filling

In IDL, a newline character terminates a statement unless preceded by a '\$'. If you would like to start a continuation line, use `M-RET`, which calls the command `idlwave-split-line`. It inserts the continuation character '\$', terminates the line and indents the new line. The command `M-RET` can also be invoked inside a string to split it at that point, in which case the '+' concatenation operator is used.

When filling comment paragraphs, IDLWAVE overloads the normal filling functions and uses a function which creates the hanging paragraphs customary in IDL routine headers. When `auto-fill-mode` is turned on (toggle with `C-c C-a`), comments will be auto-filled. If the first line of a paragraph contains a match for `idlwave-hang-indent-regexp` (a dash-space by default), subsequent lines are positioned to line up after it, as in the following example.

```

;=====
; x - an array containing
;   lots of interesting numbers.
;
; y - another variable where
;   a hanging paragraph is used
;   to describe it.
;=====

```

You can also refill a comment at any time paragraph with `M-q`. Comment delimiting lines as in the above example, consisting of one or more ';' followed by one or more of the characters '+=-\_\*', are kept in place, as is.

`idlwave-fill-comment-line-only (t)` [User Option]  
Non-nil means auto fill will only operate on comment lines.

`idlwave-auto-fill-split-string (t)` [User Option]  
Non-nil means auto fill will split strings with the IDL '+' operator.

`idlwave-split-line-string (t)` [User Option]  
Non-nil means `idlwave-split-line` will split strings with '+'.  
Non-nil means `idlwave-split-line` will split strings with '+'.

`idlwave-hanging-indent (t)` [User Option]  
Non-nil means comment paragraphs are indented under the hanging indent given by `idlwave-hang-indent-regexp` match in the first line of the paragraph.

`idlwave-hang-indent-regexp ("- ")` [User Option]  
Regular expression matching the position of the hanging indent in the first line of a comment paragraph.

`idlwave-use-last-hang-indent` (`nil`) [User Option]  
 Non-`nil` means use last match on line for `idlwave-indent-regexp`.

### 4.1.5 Syntax Highlighting

Highlighting of keywords, comments, strings etc. can be accomplished with `font-lock`. If you are using `global-font-lock-mode` (in Emacs), or have `font-lock` turned on in any other buffer in XEmacs, it should also automatically work in IDLWAVE buffers. If you'd prefer invoking `font-lock` individually by mode, you can enforce it in `idlwave-mode` with the following line in your `‘.emacs’`:

```
(add-hook 'idlwave-mode-hook 'turn-on-font-lock)
```

IDLWAVE supports 3 increasing levels of syntax highlighting. The variable `font-lock-maximum-decoration` determines which level is selected. Individual categories of special tokens can be selected for highlighting using the variable `idlwave-default-font-lock-items`.

`idlwave-default-font-lock-items` [User Option]  
 Items which should be fontified on the default fontification level 2.

### 4.1.6 Octals and Highlighting

A rare syntax highlighting problem results from an extremely unfortunate notation for octal numbers in IDL: `"123`. This unpaired quotation mark is very difficult to parse, given that it can be mixed on a single line with any number of strings. Emacs will incorrectly identify this as a string, and the highlighting of following lines of code can be distorted, since the string is never terminated.

One solution to this involves terminating the mistakenly identified string yourself by providing a closing quotation mark in a comment:

```
string("305B) + $ ;" <--- for font-lock
' is an Angstrom.'
```

A far better solution is to abandon this notation for octals altogether, and use the more sensible alternative IDL provides:

```
string('305'OB) + ' is an Angstrom.'
```

This simultaneously solves the `font-lock` problem and is more consistent with the notation for hexadecimal numbers, e.g. `'C5'XB`.

## 4.2 Routine Info

IDL comes bundled with more than one thousand procedures, functions and object methods, and large libraries typically contain hundreds or even thousands more (each with a few to tens of keywords and arguments). This large command set can make it difficult to remember the calling sequence and keywords for the routines you use, but IDLWAVE can help. It builds up routine information from a wide variety of sources; IDLWAVE in fact knows far more about the `‘.pro’` routines on your system than IDL itself! It maintains a list of all built-in routines, with calling sequences and keywords<sup>1</sup>. It also scans Emacs

<sup>1</sup> This list is created by scanning the IDL manuals and might contain (very few) errors. Please report any errors to the maintainer, so that they can be fixed.

buffers for routine definitions, queries the IDLWAVE-Shell for information about routines currently compiled there, and automatically locates library and user-created catalogs. This information is updated automatically, and so should usually be current. To force a global update and refresh the routine information, use `C-c C-i (idlwave-update-routine-info)`.

To display the information about a routine, press `C-c ?`, which calls the command `idlwave-routine-info`. When the current cursor position is on the name or in the argument list of a procedure or function, information will be displayed about the routine. For example, consider the indicated cursor positions in the following line:

```
plot,x,alog(x+5*sin(x) + 2),
  | | | | | | | |
  1 2 3 4 5 6 7 8
```

On positions 1,2 and 8, information about the ‘`plot`’ procedure will be shown. On positions 3,4, and 7, the ‘`alog`’ function will be described, while positions 5 and 6 will investigate the ‘`sin`’ function.

When you ask for routine information about an object method, and the method exists in several classes, IDLWAVE queries for the class of the object, unless the class is already known through a text property on the ‘`->`’ operator (see [Section 4.4.2 \[Object Method Completion and Class Ambiguity\]](#), page 19), or by having been explicitly included in the call (e.g. `a->myclass::Foo`).

The description displayed contains the calling sequence, the list of keywords and the source location of this routine. It looks like this:

```
Usage:      XMANAGER, NAME, ID
Keywords:   BACKGROUND CATCH CLEANUP EVENT_HANDLER GROUP_LEADER
           JUST_REG MODAL NO_BLOCK
Source:     SystemLib [LCSB] /soft1/idl53/lib/xmanager.pro
```

If a definition of this routine exists in several files accessible to IDLWAVE, several ‘`Source`’ lines will point to the different files. This may indicate that your routine is shadowing a system library routine, which may or may not be what you want (see [Section A.4 \[Load-Path Shadows\]](#), page 47). The information about the calling sequence and keywords is derived from the first source listed. Library routines are available only if you have scanned your local IDL directories or are using pre-scanned libraries (see [Section A.3 \[Catalogs\]](#), page 44). The source entry consists of a *source category*, a set of *flags* and the path to the *source file*. The following default categories exist:

<i>System</i>	A system routine of unknown origin. When the system library has been scanned as part of a catalog (see <a href="#">Section A.3 [Catalogs]</a> , page 44), this category will automatically split into the next two.
<i>Builtin</i>	A builtin system routine with no source code available.
<i>SystemLib</i>	A library system routine in the official lib directory ‘ <code>!DIR/lib</code> ’.
<i>Obsolete</i>	A library routine in the official lib directory ‘ <code>!DIR/lib/obsolete</code> ’.
<i>Library</i>	A routine in a file on IDL’s search path <code>!PATH</code> .
<i>Other</i>	Any other routine with a file not known to be on the search path.
<i>Unresolved</i>	An otherwise unknown routine the shell lists as unresolved (referenced, but not compiled).

Any routines discovered in library catalogs (see [Section A.3.1 \[Library Catalogs\]](#), page 45), will display the category assigned during creation, e.g. ‘`NasaLib`’. For routines

not discovered in this way, you can create additional categories based on the routine's filename using the variable `idlwave-special-lib-alist`.

The flags [LCSB] indicate the source of the information IDLWAVE has regarding the file: from a library catalog ([L---]), from a user catalog ([-C--]), from the IDL Shell ([-S-]) or from an Emacs buffer ([-B]). Combinations are possible (a compiled library routine visited in a buffer might read [L-SB]). If a file contains multiple definitions of the same routine, the file name will be prefixed with '(Nx)' where 'N' is the number of definitions.

Some of the text in the `*Help*` routine info buffer will be active (it is highlighted when the mouse moves over it). Typically, clicking with the right mouse button invokes online help lookup, and clicking with the middle mouse button inserts keywords or visits files:

<i>Usage</i>	If online help is installed, a click with the <i>right</i> mouse button on the <i>Usage:</i> line will access the help for the routine (see <a href="#">Section 4.3 [Online Help]</a> , page 14).
<i>Keyword</i>	Online help about keywords is also available with the <i>right</i> mouse button. Clicking on a keyword with the <i>middle</i> mouse button will insert this keyword in the buffer from where <code>idlwave-routine-info</code> was called. Holding down <code>(SHIFT)</code> while clicking also adds the initial <code>'/'</code> .
<i>Source</i>	Clicking with the <i>middle</i> mouse button on a <code>'Source'</code> line finds the source file of the routine and visits it in another window. Another click on the same line switches back to the buffer from which <code>C-c ?</code> was called. If you use the <i>right</i> mouse button, the source will not be visited by a buffer, but displayed in the online help window.
<i>Classes</i>	The <i>Classes</i> line is only included in the routine info window if the current class inherits from other classes. You can click with the <i>middle</i> mouse button to display routine info about the current method in other classes on the inheritance chain, if such a method exists there.

`idlwave-resize-routine-help-window (t)` [User Option]  
 Non-nil means resize the Routine-info `*Help*` window to fit the content.

`idlwave-special-lib-alist` [User Option]  
 Alist of regular expressions matching special library directories.

`idlwave-rinfo-max-source-lines (5)` [User Option]  
 Maximum number of source files displayed in the Routine Info window.

### 4.3 Online Help

For IDL system routines, extensive documentation is supplied with IDL. IDLWAVE can access the HTML version of this documentation very quickly and accurately, based on the local context. This can be *much* faster than using the IDL online help application, because IDLWAVE usually gets you to the right place in the documentation directly — e.g. a specific keyword of a routine — without any additional browsing and scrolling.

For this online help to work, an HTML version of the IDL documentation is required. Beginning with IDL 6.2, HTML documentation is distributed directly with IDL, along with an XML-based catalog of routine information. By default, IDLWAVE automatically attempts to convert this XML catalog into a format Emacs can more easily understand, and

caches this information in your `idlwave_config_directory` (`~/idlwave/`, by default). It also re-scans the XML catalog if it is newer than the current cached version. You can force rescan with the menu entry `IDLWAVE->Routine Info->Rescan XML Help Catalog`.

Before IDL 6.2, the HTML help was not distributed with IDL, and was not part of the standalone IDLWAVE distribution, but had to be downloaded separately. This is no longer necessary: all help and routine information is supplied with IDL versions 6.2 and later.

There are a variety of options for displaying the HTML help: see below. Help for routines without HTML documentation is also available, by using the routine documentation header and/or routine source.

In any IDL program (or, as with most IDLWAVE commands, in the IDL Shell), press `M-?` (`idlwave-context-help`), or click with `S-Mouse-3` to access context sensitive online help. The following locations are recognized context for help:

<i>Routine names</i>	The name of a routine (function, procedure, method).
<i>Keyword Parameters</i>	A keyword parameter of a routine.
<i>System Variables</i>	System variables like <code>!DPI</code> .
<i>System Variable Tags</i>	System variables tags like <code>!D.X_SIZE</code> .
<i>IDL Statements</i>	Statements like <code>PRO</code> , <code>REPEAT</code> , <code>COMPILE_OPT</code> , etc.
<i>IDL Controls</i>	Control structures like <code>FOR</code> , <code>SWITCH</code> , etc.
<i>Class names</i>	A class name in an <code>OBJ_NEW</code> call.
<i>Class Init Keywords</i>	Beyond the class name in an <code>OBJ_NEW</code> call.
<i>Executive Command</i>	An executive command like <code>.RUN</code> . Mostly useful in the shell.
<i>Structure Tags</i>	Structure tags like <code>state.xsize</code>
<i>Class Tags</i>	Class tags like <code>self.value</code> .
<i>Default</i>	The routine that would be selected for routine info display.

Note that the `OBJ_NEW` function is special in that the help displayed depends on the cursor position. If the cursor is on the `'OBJ_NEW'`, this function is described. If it is on the class name inside the quotes, the documentation for the class is pulled up. If the cursor is *after* the class name, anywhere in the argument list, the documentation for the corresponding `Init` method and its keywords is targeted.

Apart from an IDLWAVE buffer or shell, there are two more places from which online help can be accessed.

- Online help for routines and keywords can be accessed through the Routine Info display. Click with `Mouse-3` on an item to see the corresponding help (see [Section 4.2 \[Routine Info\]](#), page 12).
- When using completion and Emacs pops up a `'*Completions*` buffer with possible completions, clicking with `Mouse-3` on a completion item invokes help on that item (see [Section 4.4 \[Completion\]](#), page 18). Items for which help is available in the online system documentation (vs. just the program source itself) will be emphasized (e.g. colored blue).

In both cases, a blue face indicates that the item is documented in the IDL manual, but an attempt will be made to visit non-blue items directly in the originating source file.

### 4.3.1 Help with HTML Documentation

Help using the HTML documentation is invoked with the built-in Emacs command `browse-url`, which displays the relevant help topic in a browser of your choosing. Beginning with

version 6.2, IDL comes with the help browser *IDL Assistant*, which it uses by default for displaying online help on all supported platforms. This browser offers topical searches, an index, and is also now the default and recommended IDLWAVE help browser. The variable `idlwave-help-use-assistant` controls whether this browser is used. Note that, due to limitations in the Assistant, invoking help within IDLWAVE and `? topic` within IDL will result in two running copies of Assistant.

Aside from the IDL Assistant, there are many possible browsers to choose among, with differing advantages and disadvantages. The variable `idlwave-help-browser-function` controls which browser help is sent to (as long as `idlwave-help-use-assistant` is not set). This function is used to set the variable `browse-url-browser-function` locally for IDLWAVE help only. Customize the latter variable to see what choices of browsers your system offers. Certain browsers like `w3` (bundled with many versions of Emacs) and `w3m` (<http://emacs-w3m.namazu.org/>) are run within Emacs, and use Emacs buffers to display the HTML help. This can be convenient, especially on small displays, and images can even be displayed in-line on newer Emacs versions. However, better formatting results are often achieved with external browsers, like Mozilla. IDLWAVE assumes any browser function containing "w3" is displayed in a local buffer. If you are using another Emacs-local browser for which this is not true, set the variable `idlwave-help-browser-is-local`.

With IDL 6.2 or later, it is important to ensure that the variable `idlwave-system-directory` is set (see [Section A.3 \[Catalogs\], page 44](#)). One easy way to ensure this is to run the IDL Shell (`C-c C-s`). It will be queried for this directory, and the results will be cached to file for subsequent use.

See [Appendix B \[HTML Help Browser Tips\], page 49](#), for more information on selecting and configuring a browser for use with IDL's HTML help system.

`idlwave-html-system-help-location` `'help/online_help'` [User Option]  
Relative directory of the system-supplied HTML help directory, considered with respect to `idlwave-system-directory`. Relevant for IDL 6.2 and greater. Should not change.

`idlwave-html-help-location` `'/usr/local/etc/'` [User Option]  
The directory where the `'idl_html_help'` HTML directory live. Obsolete and ignored for IDL 6.2 and greater (`idlwave-html-system-help-location` is used instead).

`idlwave-help-use-assistant` `t` [User Option]  
If set, use the IDL Assistant if possible for online HTML help, otherwise use the browser function specified in `idlwave-help-browser-function`.

`idlwave-help-browser-function` [User Option]  
The browser function to use to display IDLWAVE HTML help. Should be one of the functions available for setting `browse-url-browser-function`, which see.

`idlwave-help-browser-is-local` [User Option]  
Is the browser selected in `idlwave-help-browser-function` run in a local Emacs buffer or window? Defaults to `t` if the function contains "-w3".

`idlwave-help-link-face` [User Option]  
The face for links to IDLWAVE online help.

### 4.3.2 Help with Source

For routines which are not documented in an HTML manual (for example personal or library routines), the source code itself is used as help text. If the requested information can be found in a (more or less) standard DocLib file header, IDLWAVE shows the header (scrolling down to a keyword, if appropriate). Otherwise the routine definition statement (`pro/function`) is shown. The doclib header sections which are searched for include 'NAME' and 'KEYWORDS'. Localization support can be added by customizing the `idlwave-help-doclib-name` and `idlwave-help-doclib-keyword` variables.

Help is also available for class structure tags (`self.TAG`), and generic structure tags, if structure tag completion is enabled (see [Section 4.4.5 \[Structure Tag Completion\]](#), page 21). This is implemented by visiting the tag within the class or structure definition source itself. Help is not available on built-in system class tags.

The help window is normally displayed in the same frame, but can be popped-up in a separate frame. The following commands can be used to navigate inside the help system for source files:

<code>(SPACE)</code>	Scroll forward one page.
<code>(RET)</code>	Scroll forward one line.
<code>(DEL)</code>	Scroll back one page.
<code>h</code>	Jump to DocLib Header of the routine whose source is displayed as help.
<code>H</code>	Jump to the first DocLib Header in the file.
<code>.</code> (Dot)	Jump back and forth between the routine definition (the <code>pro/function</code> statement) and the description of the help item in the DocLib header.
<code>F</code>	Fontify the buffer like source code. See the variable <code>idlwave-help-fontify-source-code</code> .
<code>q</code>	Kill the help window.

`idlwave-help-use-dedicated-frame (nil)` [User Option]  
 Non-`nil` means use a separate frame for Online Help if possible.

`idlwave-help-frame-parameters` [User Option]  
 The frame parameters for the special Online Help frame.

`idlwave-max-popup-menu-items (20)` [User Option]  
 Maximum number of items per pane in pop-up menus.

`idlwave-extra-help-function` [User Option]  
 Function to call for help if the normal help fails.

`idlwave-help-fontify-source-code (nil)` [User Option]  
 Non-`nil` means fontify source code displayed as help.

`idlwave-help-source-try-header (t)` [User Option]  
 Non-`nil` means try to find help in routine header when displaying source file.

`idlwave-help-doclib-name ("name")` [User Option]  
 The case-insensitive heading word in doclib headers to locate the *name* section. Can be a regexp, e.g. `"\\(name\\|nom\\)"`.

`idlwave-help-doclib-keyword` ("KEYWORD") [User Option]  
 The case-insensitive heading word in doclib headers to locate the *keywords* section.  
 Can be a regexp.

## 4.4 Completion

IDLWAVE offers completion for class names, routine names, keywords, system variables, system variable tags, class structure tags, regular structure tags and file names. As in many programming modes, completion is bound to  $M-\overline{\text{TAB}}$  (or simply  $\overline{\text{TAB}}$  in the IDLWAVE Shell — see Section 5.2 [Using the Shell], page 30). Completion uses exactly the same internal information as routine info, so when necessary (rarely) it can be updated with  $C-c C-i$  (`idlwave-update-routine-info`).

The completion function is context sensitive and figures out what to complete based on the location of the point. Here are example lines and what  $M-\overline{\text{TAB}}$  would try to complete when the cursor is on the position marked with a ‘\_’:

<code>plo_</code>	Procedure
<code>x = a_</code>	Function
<code>plot,xra_</code>	Keyword of <code>plot</code> procedure
<code>plot,x,y,/x_</code>	Keyword of <code>plot</code> procedure
<code>plot,min(_</code>	Keyword of <code>min</code> function
<code>obj -&gt; a_</code>	Object method (procedure)
<code>a[2,3] = obj -&gt; a_</code>	Object method (function)
<code>x = obj_new('IDL_</code>	Class name
<code>x = obj_new('MyCl',a_</code>	Keyword to <code>Init</code> method in class <code>MyCl</code>
<code>pro A_</code>	Class name
<code>pro _</code>	Fill in <code>Class::</code> of first method in this file
<code>!v_</code>	System variable
<code>!version.t_</code>	Structure tag of system variable
<code>self.g_</code>	Class structure tag in methods
<code>state.w_</code>	Structure tag, if tag completion enabled
<code>name = 'a_</code>	File name (default inside quotes)

The only place where completion is ambiguous is procedure/function *keywords* versus *functions*. After ‘`plot,x,_`’, IDLWAVE will always assume a keyword to ‘`plot`’. However, a function is also a possible completion here. You can force completion of a function name at such a location by using a prefix arg:  $C-u M-\overline{\text{TAB}}$ .

Giving two prefix arguments ( $C-u C-u M-\overline{\text{TAB}}$ ) prompts for a regular expression to search among the commands to be completed. As an example, completing a blank line in this way will allow you to search for a procedure matching a regexp.

If the list of completions is too long to fit in the ‘\*Completions\*’ window, the window can be scrolled by pressing  $M-\overline{\text{TAB}}$  repeatedly. Online help (if installed) for each possible completion is available by clicking with *Mouse-3* on the item. Items for which system online help (from the IDL manual) is available will be emphasized (e.g. colored blue). For other items, the corresponding source code or DocLib header will be used as the help text.

Completion is not a blocking operation — you are free to continue editing, enter commands, or simply ignore the ‘\*Completions\*’ buffer during a completion operation. If,

however, the most recent command was a completion, `C-g` will remove the buffer and restore the window configuration. You can also remove the buffer at any time with no negative consequences.

`idlwave-keyword-completion-adds-equal` (t) [User Option]  
 Non-`nil` means completion automatically adds '=' after completed keywords.

`idlwave-function-completion-adds-paren` (t) [User Option]  
 Non-`nil` means completion automatically adds '(' after completed function. A value of '2' means also add the closing parenthesis and position the cursor between the two.

`idlwave-completion-restore-window-configuration` (t) [User Option]  
 Non-`nil` means restore window configuration after successful completion.

`idlwave-highlight-help-links-in-completion` (t) [User Option]  
 Non-`nil` means highlight completions for which system help is available.

#### 4.4.1 Case of Completed Words

IDL is a case-insensitive language, so casing is a matter of style only. IDLWAVE helps maintain a consistent casing style for completed items. The case of the completed words is determined by what is already in the buffer. As an exception, when the partial word being completed is all lower case, the completion will be lower case as well. If at least one character is upper case, the string will be completed in upper case or mixed case, depending on the value of the variable `idlwave-completion-case`. The default is to use upper case for procedures, functions and keywords, and mixed case for object class names and methods, similar to the conventions in the IDL manuals. For instance, to enable mixed-case completion for routines in addition to classes and methods, you need an entry such as (`routine . preserve`) in that variable. To enable total control over the case of completed items, independent of buffer context, set `idlwave-completion-force-default-case` to `non-nil`.

`idlwave-completion-case` [User Option]  
 Association list setting the case (UPPER/lower/Capitalized/MixedCase...) of completed words.

`idlwave-completion-force-default-case` (nil) [User Option]  
 Non-`nil` means completion will always honor the settings in `idlwave-completion-case`. When `nil` (the default), entirely lower case strings will always be completed to lower case, no matter what the settings in `idlwave-completion-case`.

`idlwave-complete-empty-string-as-lower-case` (nil) [User Option]  
 Non-`nil` means the empty string is considered lower case for completion.

#### 4.4.2 Object Method Completion and Class Ambiguity

An object method is not uniquely determined without the object's class. Since the class is almost always omitted in the calling source (as required to obtain the true benefits of object-based programming), IDLWAVE considers all available methods in all classes as possible method name completions. The combined list of keywords of the current method in *all* known classes which contain that method will be considered for keyword completion.

In the `*Completions*` buffer, the matching classes will be shown next to each item (see option `idlwave-completion-show-classes`). As a special case, the class of an object called `'self'` is always taken to be the class of the current routine, when in an IDLWAVE buffer. All inherits classes are considered as well.

You can also call `idlwave-complete` with a prefix arg: `C-u M-TAB`. IDLWAVE will then prompt you for the class in order to narrow down the number of possible completions. The variable `idlwave-query-class` can be configured to make such prompting the default for all methods (not recommended), or selectively for very common methods for which the number of completing keywords would be too large (e.g. `Init`, `SetProperty`, `GetProperty`).

After you have specified the class for a particular statement (e.g. when completing the method), IDLWAVE can remember it for the rest of the editing session. Subsequent completions in the same statement (e.g. keywords) can then reuse this class information. This works by placing a text property on the method invocation operator `'->'`, after which the operator will be shown in a different face (bold by default). The variable `idlwave-store-inquired-class` can be used to turn it off or on.

`idlwave-completion-show-classes` (1) [User Option]

Non-nil means show up to that many classes in `*Completions*` buffer when completing object methods and keywords.

`idlwave-completion-fontify-classes` (t) [User Option]

Non-nil means fontify the classes in completions buffer.

`idlwave-query-class` (nil) [User Option]

Association list governing query for object classes during completion.

`idlwave-store-inquired-class` (t) [User Option]

Non-nil means store class of a method call as text property on `'->'`.

`idlwave-class-arrow-face` [User Option]

Face to highlight object operator arrows `'->'` which carry a saved class text property.

### 4.4.3 Object Method Completion in the Shell

In the IDLWAVE Shell (see [Chapter 5 \[The IDLWAVE Shell\]](#), page 29), objects on which methods are being invoked have a special property: they must exist as variables, and so their class can be determined (for instance, using the `obj_class()` function). In the Shell, when attempting completion, routine info, or online help within a method routine, a query is sent to determine the class of the object. If this query is successful, the class found will be used to select appropriate completions, routine info, or help. If unsuccessful, information from all known classes will be used (as in the buffer).

### 4.4.4 Class and Keyword Inheritance

Class inheritance affects which methods are called in IDL. An object of a class which inherits methods from one or more superclasses can override that method by defining its own method of the same name, extend the method by calling the method(s) of its superclass(es) in its version, or inherit the method directly by making no modifications. IDLWAVE examines class definitions during completion and routine information display, and records all inheritance information it finds. This information is displayed if appropriate with the

calling sequence for methods (see [Section 4.2 \[Routine Info\]](#), page 12), as long as variable `idlwave-support-inheritance` is non-`nil`.

In many class methods, *keyword inheritance* (`_EXTRA` and `_REF_EXTRA`) is used hand-in-hand with class inheritance and method overriding. E.g., in a `SetProperty` method, this technique allows a single call `obj->SetProperty` to set properties up the entire class inheritance chain. This is often referred to as *chaining*, and is characterized by chained method calls like `self->MySuperClass::SetProperty, _EXTRA=e`.

IDLWAVE can accommodate this special synergy between class and keyword inheritance: if `_EXTRA` or `_REF_EXTRA` is detected among a method's keyword parameters, all keywords of superclass versions of the method being considered can be included in completion. There is of course no guarantee that this type of keyword chaining actually occurs, but for some methods it's a very convenient assumption. The variable `idlwave-keyword-class-inheritance` can be used to configure which methods have keyword inheritance treated in this simple, class-driven way. By default, only `Init` and `(Get|Set)Property` are. The completion buffer will label keywords based on their originating class.

`idlwave-support-inheritance` (t) [User Option]  
 Non-`nil` means consider inheritance during completion, online help etc.

`idlwave-keyword-class-inheritance` [User Option]  
 A list of regular expressions to match methods for which simple class-driven keyword inheritance will be used for Completion.

#### 4.4.5 Structure Tag Completion

In many programs, especially those involving widgets, large structures (e.g. the `'state'` structure) are used to communicate among routines. It is very convenient to be able to complete structure tags, in the same way as for instance variables (tags) of the `'self'` object (see [Section 4.4.2 \[Object Method Completion and Class Ambiguity\]](#), page 19). Add-in code for structure tag completion is available in the form of a loadable completion module: `'idlw-complete-structtag.el'`. Tag completion in structures is highly ambiguous (much more so than `'self'` completion), so `idlw-complete-structtag` makes an unusual and very specific assumption: the exact same variable name is used to refer to the structure in all parts of the program. This is entirely unenforced by the IDL language, but is a typical convention. If you consistently refer to the same structure with the same variable name (e.g. `'state'`), structure tags which are read from its definition in the same file can be used for completion.

Structure tag completion is not enabled by default. To enable it, simply add the following to your `'emacs'`:

```
(add-hook 'idlwave-load-hook
          (lambda () (require 'idlw-complete-structtag)))
```

Once enabled, you'll also be able to access online help on the structure tags, using the usual methods (see [Section 4.3 \[Online Help\]](#), page 14). In addition, structure variables in the shell will be queried for tag names, similar to the way object variables in the shell are queried for method names. So, e.g.:

```
IDL> st.[Tab]
```

will complete with all structure fields of the structure `st`.

## 4.5 Routine Source

In addition to clicking on a *Source:* line in the routine info window, there is another way to quickly visit the source file of a routine. The command `C-c C-v` (`idlwave-find-module`) asks for a module name, offering the same default as `idlwave-routine-info` would have used, taken from nearby buffer contents. In the minibuffer, specify a complete routine name (including any class part). IDLWAVE will display the source file in another window, positioned at the routine in question. You can also limit this to a routine in the current buffer only, with completion, and a context-sensitive default, by using a single prefix (`C-u C-c C-v`) or the convenience binding `C-c C-t`.

Since getting the source of a routine into a buffer is so easy with IDLWAVE, too many buffers visiting different IDL source files are sometimes created. The special command `C-c C-k` (`idlwave-kill-autoloaded-buffers`) can be used to easily remove these buffers.

## 4.6 Resolving Routines

The key sequence `C-c =` calls the command `idlwave-resolve` and sends the line `'RESOLVE_ROUTINE, 'routine_name''` to IDL in order to resolve (compile) it. The default routine to be resolved is taken from context, but you get a chance to edit it. Usually this is not necessary, since IDL automatically discovers routines on its path.

`idlwave-resolve` is one way to get a library module within reach of IDLWAVE's routine info collecting functions. A better way is to keep routine information available in catalogs (see [Section A.3 \[Catalogs\], page 44](#)). Routine info on modules will then be available without the need to compile the modules first, and even without a running shell.

See [Appendix A \[Sources of Routine Info\], page 43](#), for more information on the ways IDLWAVE collects data about routines, and how to update this information.

## 4.7 Code Templates

IDLWAVE can insert IDL code templates into the buffer. For a few templates, this is done with direct key bindings:

<code>C-c C-c</code>	CASE statement template
<code>C-c C-f</code>	FOR loop template
<code>C-c C-r</code>	REPEAT loop template
<code>C-c C-w</code>	WHILE loop template

All code templates are also available as abbreviations (see [Section 4.8 \[Abbreviations\], page 22](#)).

## 4.8 Abbreviations

Special abbreviations exist to enable rapid entry of commonly used commands. Emacs abbreviations are expanded by typing text into the buffer and pressing `(SPC)` or `(RET)`. The special abbreviations used to insert code templates all start with a `'\'` (the backslash), or, optionally, any other character set in `idlwave-abbrev-start-char`. IDLWAVE ensures that abbreviations are only expanded where they should be (i.e., not in a string or comment), and permits the point to be moved after an abbreviation expansion — very useful for positioning the mark inside of parentheses, etc.

Special abbreviations are pre-defined for code templates and other useful items. To visit the full list of abbreviations, use *M-x idlwave-list-abbrevs*.

Template abbreviations:

<code>\pr</code>	PROCEDURE template
<code>\fu</code>	FUNCTION template
<code>\c</code>	CASE statement template
<code>\f</code>	FOR loop template
<code>\r</code>	REPEAT loop template
<code>\w</code>	WHILE loop template
<code>\i</code>	IF statement template
<code>\elif</code>	IF-ELSE statement template

String abbreviations:

<code>\ap</code>	<code>arg_present()</code>
<code>\b</code>	<code>begin</code>
<code>\cb</code>	<code>byte()</code>
<code>\cc</code>	<code>complex()</code>
<code>\cd</code>	<code>double()</code>
<code>\cf</code>	<code>float()</code>
<code>\cl</code>	<code>long()</code>
<code>\co</code>	<code>common</code>
<code>\cs</code>	<code>string()</code>
<code>\cx</code>	<code>fix()</code>
<code>\e</code>	<code>else</code>
<code>\ec</code>	<code>endcase</code>
<code>\ee</code>	<code>endelse</code>
<code>\ef</code>	<code>endfor</code>
<code>\ei</code>	<code>endif else if</code>
<code>\el</code>	<code>endif else</code>
<code>\en</code>	<code>endif</code>
<code>\er</code>	<code>endrep</code>
<code>\es</code>	<code>endswitch</code>
<code>\ew</code>	<code>endwhile</code>
<code>\g</code>	<code>goto,</code>
<code>\h</code>	<code>help,</code>
<code>\ik</code>	<code>if keyword_set() then</code>
<code>\iap</code>	<code>if arg_present() then</code>
<code>\ine</code>	<code>if n_elements() eq 0 then</code>
<code>\inn</code>	<code>if n_elements() ne 0 then</code>
<code>\k</code>	<code>keyword_set()</code>
<code>\n</code>	<code>n_elements()</code>
<code>\np</code>	<code>n_params()</code>
<code>\oi</code>	<code>on_ioerror,</code>
<code>\or</code>	<code>openr,</code>
<code>\ou</code>	<code>openu,</code>
<code>\ow</code>	<code>openw,</code>
<code>\p</code>	<code>print,</code>

```

\pt      plot,
\pv      ptr_valid()
\re      read,
\rf      readf,
\rt      return
\ru      readu,
\s       size()
\sc      strcompress()
\sl      strlowercase()
\sm      strmid()
\sn      strlen()
\sp      strpos()
\sr      strtrim()
\st      strput()
\su      struppercase()
\t       then
\u       until
\wc      widget_control,
\wi      widget_info()
\wu      writeu,

```

You can easily add your own abbreviations or override existing abbrevs with `define-abbrev` in your mode hook, using the convenience function `idlwave-define-abbrev`:

```

(add-hook 'idlwave-mode-hook
  (lambda ()
    (idlwave-define-abbrev "wb" "widget_base()"
      (idlwave-keyword-abbrev 1))
    (idlwave-define-abbrev "ine" "IF N_Elements() EQ 0 THEN"
      (idlwave-keyword-abbrev 11))))

```

Notice how the abbreviation (here *wb*) and its expansion (*widget\_base()*) are given as arguments, and the single argument to `idlwave-keyword-abbrev` (here *1*) specifies how far back to move the point upon expansion (in this example, to put it between the parentheses).

The abbreviations are expanded in upper or lower case, depending upon the variables `idlwave-abbrev-change-case` and, for reserved word templates, `idlwave-reserved-word-upcase` (see [Section 4.9.3 \[Case Changes\]](#), page 26).

`idlwave-abbrev-start-char` ("`\`") [User Option]

A single character string used to start abbreviations in abbrev mode. Beware of common characters which might naturally occur in sequence with abbreviation strings.

`idlwave-abbrev-move` (t) [User Option]

Non-nil means the abbrev hook can move point, e.g. to end up between the parentheses of a function call.

## 4.9 Actions

*Actions* are special formatting commands which are executed automatically while you write code in order to check the structure of the program or to enforce coding standards. Most

actions which have been implemented in IDLWAVE are turned off by default, assuming that the average user wants her code the way she writes it. But if you are a lazy typist and want your code to adhere to certain standards, actions can be helpful.

Actions can be applied in three ways:

- Some actions are applied directly while typing. For example, pressing ‘=’ can run a check to make sure that this operator is surrounded by spaces and insert these spaces if necessary. Pressing `(SPC)` after a reserved word can call a command to change the word to upper case.
- When a line is re-indented with `(TAB)`, actions can be applied to the entire line. To enable this, the variable `idlwave-do-actions` must be non-`nil`.
- Actions can also be applied to a larger piece of code, e.g. to convert foreign code to your own style. To do this, mark the relevant part of the code and execute `M-x expand-region-abbrevs`. Useful marking commands are `C-x h` (the entire file) or `C-M-h` (the current subprogram). See [Section 4.1.1 \[Code Indentation\], page 9](#), for information how to adjust the indentation of the code.

`idlwave-do-actions` (`nil`) [User Option]

Non-`nil` means performs actions when indenting. Individual action settings are described below and set separately.

### 4.9.1 Block Boundary Check

Whenever you type an `END` statement, IDLWAVE finds the corresponding start of the block and the cursor blinks back to that location for a second. If you have typed a specific `END`, like `ENDIF` or `ENDCASE`, you get a warning if that terminator does not match the type of block it terminates.

Set the variable `idlwave-expand-generic-end` in order to have all generic `END` statements automatically expanded to the appropriate type. You can also type `C-c ]` to close the current block by inserting the appropriate `END` statement.

`idlwave-show-block` (`t`) [User Option]

Non-`nil` means point blinks to block beginning for `idlwave-show-begin`.

`idlwave-expand-generic-end` (`t`) [User Option]

Non-`nil` means expand generic `END` to `ENDIF/ENDELSE/ENDWHILE` etc.

`idlwave-reindent-end` (`t`) [User Option]

Non-`nil` means re-indent line after `END` was typed.

### 4.9.2 Padding Operators

Some operators can be automatically surrounded by spaces. This can happen when the operator is typed, or later when the line is indented. IDLWAVE can pad the operators ‘<’, ‘>’, ‘,’, ‘=’, and ‘->’, as well as the modified assignment operators (‘`AND=`’, ‘`OR=`’, etc.). This feature is turned off by default. If you want to turn it on, customize the variables `idlwave-surround-by-blank` and `idlwave-do-actions` and turn both on. You can also define similar actions for other operators by using the function `idlwave-action-and-binding` in the mode hook. For example, to enforce space padding of the ‘+’ and ‘\*’ operators (outside of strings and comments, of course), try this in ‘.emacs’

```
(add-hook 'idlwave-mode-hook
  (lambda ()
    (setq idlwave-surround-by-blank t) ; Turn this type of actions on
    (idlwave-action-and-binding "*" '(idlwave-surround 1 1))
    (idlwave-action-and-binding "+" '(idlwave-surround 1 1))))
```

Note that the modified assignment operators which begin with a word ('AND=', 'OR=', 'NOT=', etc.) require a leading space to be recognized (e.g. `vAND=4` would be interpreted as a variable `vAND`). Also note that, since e.g., `>` and `>=` are both valid operators, it is impossible to surround both by blanks while they are being typed. Similarly with `&` and `&&`. For these, a compromise is made: the padding is placed on the left, and if the longer operator is keyed in, on the right as well (otherwise you must insert spaces to pad right yourself, or press simply press `Tab` to repad everything if `idlwave-do-actions` is on).

**idlwave-surround-by-blank** (nil) [User Option]  
 Non-nil means enable `idlwave-surround`. If non-nil, '=', '<', '>', '&', ',', '->', and the modified assignment operators ('AND=', 'OR=', etc.) are surrounded with spaces by `idlwave-surround`.

**idlwave-pad-keyword** (t) [User Option]  
 Non-nil means space-pad the '=' in keyword assignments.

### 4.9.3 Case Changes

Actions can be used to change the case of reserved words or expanded abbreviations by customizing the variables `idlwave-abbrev-change-case` and `idlwave-reserved-word-upcase`. If you want to change the case of additional words automatically, put something like the following into your '.emacs' file:

```
(add-hook 'idlwave-mode-hook
  (lambda ()
    ;; Capitalize system vars
    (idlwave-action-and-binding idlwave-sysvar '(capitalize-word 1) t)
    ;; Capitalize procedure name
    (idlwave-action-and-binding "\\<\\(pro\\|function\\)\\>[ \\t]*\\<"
      '(capitalize-word 1) t)
    ;; Capitalize common block name
    (idlwave-action-and-binding "\\<common\\>[ \\t]+\\<"
      '(capitalize-word 1) t)))
```

For more information, see the documentation string for the function `idlwave-action-and-binding`. For information on controlling the case of routines, keywords, classes, and methods as they are completed, see [Section 4.4 \[Completion\], page 18](#).

**idlwave-abbrev-change-case** (nil) [User Option]  
 Non-nil means all abbrevs will be forced to either upper or lower case. Valid values are nil, t, and down.

**idlwave-reserved-word-upcase** (nil) [User Option]  
 Non-nil means reserved words will be made upper case via abbrev expansion.

## 4.10 Documentation Header

The command `C-c C-h` inserts a standard routine header into the buffer, with the usual fields for documentation (a different header can be specified with `idlwave-file-header`). One of the keywords is ‘MODIFICATION HISTORY’ under which the changes to a routine can be recorded. The command `C-c C-m` jumps to the ‘MODIFICATION HISTORY’ of the current routine or file and inserts the user name with a timestamp.

`idlwave-file-header` [User Option]

The doc-header template or a path to a file containing it.

`idlwave-header-to-beginning-of-file` (nil) [User Option]

Non-nil means the documentation header will always be at start of file.

`idlwave-timestamp-hook` [User Option]

The hook function used to update the timestamp of a function.

`idlwave-doc-modifications-keyword` [User Option]

The modifications keyword to use with the log documentation commands.

`idlwave-doclib-start` [User Option]

Regexp matching the start of a document library header.

`idlwave-doclib-end` [User Option]

Regexp matching the start of a document library header.

## 4.11 Motion Commands

IDLWAVE supports both ‘Imenu’ and ‘Func-menu’, two packages which make it easy to jump to the definitions of functions and procedures in the current file with a pop-up selection. To bind ‘Imenu’ to a mouse-press, use in your ‘.emacs’:

```
(define-key global-map [S-down-mouse-3] 'imenu)
```

In addition, ‘Speedbar’ support allows convenient navigation of a source tree of IDL routine files, quickly stepping to routine definitions. See `Tools->Display Speedbar`.

Several commands allow you to move quickly through the structure of an IDL program:

<code>C-M-a</code>	Beginning of subprogram
<code>C-M-e</code>	End of subprogram
<code>C-c {</code>	Beginning of block (stay inside the block)
<code>C-c }</code>	End of block (stay inside the block)
<code>C-M-n</code>	Forward block (on same level)
<code>C-M-p</code>	Backward block (on same level)
<code>C-M-d</code>	Down block (enters a block)
<code>C-M-u</code>	Backward up block (leaves a block)
<code>C-c C-n</code>	Next Statement

## 4.12 Miscellaneous Options

<code>idlwave-help-application</code>	[User Option]
The external application providing reference help for programming.	
<code>idlwave-startup-message (t)</code>	[User Option]
Non-nil means display a startup message when <code>idlwave-mode</code> ' is first called.	
<code>idlwave-mode-hook</code>	[User Option]
Normal hook. Executed when a buffer is put into <code>idlwave-mode</code> .	
<code>idlwave-load-hook</code>	[User Option]
Normal hook. Executed when <code>'idlwave.el'</code> is loaded.	

## 5 The IDLWAVE Shell

The IDLWAVE shell is an Emacs major mode which permits running the IDL program as an inferior process of Emacs, and works closely with the IDLWAVE major mode in buffers. It can be used to work with IDL interactively, to compile and run IDL programs in Emacs buffers and to debug these programs. The IDLWAVE shell is built on ‘comint’, an Emacs packages which handles the communication with the IDL program. Unfortunately, IDL for Windows does not have command-prompt versions and thus do not allow the interaction with Emacs — so the IDLWAVE shell currently only works under Unix and MacOSX.

### 5.1 Starting the Shell

The IDLWAVE shell can be started with the command *M-x idlwave-shell*. In *idlwave-mode* the function is bound to *C-c C-s*. It creates a buffer ‘\*idl\*’ which is used to interact with the shell. If the shell is already running, *C-c C-s* will simply switch to the shell buffer. The command *C-c C-l* (*idlwave-shell-recenter-shell-window*) displays the shell window without selecting it. The shell can also be started automatically when another command tries to send a command to it. To enable auto start, set the variable *idlwave-shell-automatic-start* to *t*.

In order to create a separate frame for the IDLWAVE shell buffer, call *idlwave-shell* with a prefix argument: *C-u C-c C-s* or *C-u C-c C-l*. If you always want a dedicated frame for the shell window, configure the variable *idlwave-shell-use-dedicated-frame*.

To launch a quick IDLWAVE shell directly from a shell prompt without an IDLWAVE buffer (e.g., as a replacement for running inside an xterm), define a system alias with the following content:

```
emacs -geometry 80x32 -eval "(idlwave-shell 'quick)"
```

Replace the ‘-geometry 80x32’ option with ‘-nw’ if you prefer the Emacs process to run directly inside the terminal window.

To use IDLWAVE with ENVI or other custom packages which change the ‘IDL>’ prompt, you must change the *idlwave-shell-prompt-pattern*, which defaults to “`^ ?IDL>`”. Normally, you can just replace the ‘IDL’ in this expression with the prompt you see. A suitable pattern which matches the prompt for both ENVI and IDL simultaneously is “`^ ?\\(ENVI\\|IDL\\)>`”.

*idlwave-shell-explicit-file-name* (‘idl’) [User Option]  
This is the command to run IDL.

*idlwave-shell-command-line-options* [User Option]  
A list of command line options for calling the IDL program.

*idlwave-shell-prompt-pattern* [User Option]  
Regex to match IDL prompt at beginning of a line.

*idlwave-shell-process-name* [User Option]  
Name to be associated with the IDL process.

*idlwave-shell-automatic-start* (nil) [User Option]  
Non-nil means attempt to invoke *idlwave-shell* if not already running.

<code>idlwave-shell-initial-commands</code>	[User Option]
Initial commands, separated by newlines, to send to IDL.	
<code>idlwave-shell-save-command-history</code> (t)	[User Option]
Non-nil means preserve command history between sessions.	
<code>idlwave-shell-command-history-file</code>	[User Option]
( <code>~/idlwave/.idlwhist</code> )	
The file in which the command history of the idlwave shell is saved. Unless it's an absolute path, it goes in <code>idlwave-config-directory</code> .	
<code>idlwave-shell-use-dedicated-frame</code> (nil)	[User Option]
Non-nil means IDLWAVE should use a special frame to display the shell buffer.	
<code>idlwave-shell-use-dedicated-window</code> (nil)	[User Option]
Non-nil means use a dedicated window for the shell, taking care not it replace it with other buffers.	
<code>idlwave-shell-frame-parameters</code>	[User Option]
The frame parameters for a dedicated idlwave-shell frame.	
<code>idlwave-shell-raise-frame</code> (t)	[User Option]
Non-nil means 'idlwave-shell' raises the frame showing the shell window.	
<code>idlwave-shell-temp-pro-prefix</code>	[User Option]
The prefix for temporary IDL files used when compiling regions.	
<code>idlwave-shell-mode-hook</code>	[User Option]
Hook for customizing <code>idlwave-shell-mode</code> .	

## 5.2 Using the Shell

The IDLWAVE shell works in the same fashion as other shell modes in Emacs. It provides command history, command line editing and job control. The `UP` and `DOWN` arrows cycle through the input history just like in an X terminal<sup>1</sup>. The history is preserved between emacs and IDL sessions. Here is a list of commonly used commands:

<code>UP</code> , <code>M-p</code>	Cycle backwards in input history
<code>DOWN</code> ,	Cycle forwards in input history
<code>M-n</code>	
<code>M-r</code>	Previous input matching a regexp
<code>M-s</code>	Next input matching a regexp
<code>return</code>	Send input or copy line to current prompt
<code>C-c C-a</code>	Beginning of line; skip prompt
<code>C-c C-u</code>	Kill input to beginning of line
<code>C-c C-w</code>	Kill word before cursor
<code>C-c C-c</code>	Send <code>^C</code>
<code>C-c C-z</code>	Send <code>^Z</code>

<sup>1</sup> This is different from normal Emacs/Comint behavior, but more like an xterm. If you prefer the default comint functionality, check the variable `idlwave-shell-arrows-do-history`.

`C-c C-\` Send `^\`  
`C-c C-o` Delete last batch of process output  
`C-c C-r` Show last batch of process output  
`C-c C-l` List input history

In addition to these standard ‘comint’ commands, `idlwave-shell-mode` provides many of the same commands which simplify writing IDL code available in IDLWAVE buffers. This includes abbreviations, online help, and completion. See [Section 4.2 \[Routine Info\]](#), page 12 and [Section 4.3 \[Online Help\]](#), page 14 and [Section 4.4 \[Completion\]](#), page 18 for more information on these commands.

`(TAB)` Completion of file names (between quotes and after executive commands ‘.run’ and ‘.compile’), routine names, class names, keywords, system variables, system variable tags etc. (`idlwave-shell-complete`).

`M-(TAB)` Same as `(TAB)`

`C-c ?` Routine Info display (`idlwave-routine-info`)

`M-?` IDL online help on routine (`idlwave-routine-info-from-idlhelp`)

`C-c C-i` Update routine info from buffers and shell (`idlwave-update-routine-info`)

`C-c C-v` Find the source file of a routine (`idlwave-find-module`)

`C-c C-t` Find the source file of a routine in the currently visited file (`idlwave-find-module-this-file`).

`C-c =` Compile a library routine (`idlwave-resolve`)

`idlwave-shell-arrows-do-history (t)` [User Option]  
 Non-nil means `(UP)` and `(DOWN)` arrows move through command history like xterm.

`idlwave-shell-comint-settings` [User Option]  
 Alist of special settings for the comint variables in the IDLWAVE Shell.

`idlwave-shell-file-name-chars` [User Option]  
 The characters allowed in file names, as a string. Used for file name completion.

`idlwave-shell-graphics-window-size` [User Option]  
 Size of IDL graphics windows popped up by special IDLWAVE command.

IDLWAVE works in line input mode: You compose a full command line, using all the power Emacs gives you to do this. When you press `(RET)`, the whole line is sent to IDL. Sometimes it is necessary to send single characters (without a newline), for example when an IDL program is waiting for single character input with the `GET_KBRD` function. You can send a single character to IDL with the command `C-c C-x` (`idlwave-shell-send-char`). When you press `C-c C-y` (`idlwave-shell-char-mode-loop`), IDLWAVE runs a blocking loop which accepts characters and immediately sends them to IDL. The loop can be exited with `C-g`. It terminates also automatically when the current IDL command is finished. Check the documentation of the two variables described below for a way to make IDL programs trigger automatic switches of the input mode.

`idlwave-shell-use-input-mode-magic (nil)` [User Option]  
 Non-nil means IDLWAVE should check for input mode spells in output.

`idlwave-shell-input-mode-spells` [User Option]  
 The three regular expressions which match the magic spells for input modes.

### 5.3 Commands Sent to the Shell

The IDLWAVE buffers and shell interact very closely. In addition to the normal commands you enter at the IDL> prompt, many other special commands are sent to the shell, sometimes as a direct result of invoking a key command, menu item, or toolbar button, but also automatically, as part of the normal flow of information updates between the buffer and shell.

The commands sent include `breakpoint`, `.step` and other debug commands (see [Section 5.4 \[Debugging IDL Programs\]](#), page 32), `.run` and other compilation statements (see [Section 5.4.4 \[Compiling Programs\]](#), page 35), examination commands like `print` and `help` (see [Section 5.5 \[Examining Variables\]](#), page 38), and other special purpose commands designed to keep information on the running shell current.

By default, much of this background shell input and output is hidden from the user, but this is configurable. The custom variable `idlwave-abbrev-show-commands` allows you to configure which commands sent to the shell are shown there. For a related customization for separating the output of *examine* commands, see [Section 5.5 \[Examining Variables\]](#), page 38.

`idlwave-shell-show-commands ('(run misc breakpoint))` [User Option]

A list of command types to echo in the shell when sent. Possible values are `run` for `.run`, `.compile` and other run commands, `misc` for lesser used commands like `window`, `retail`, `close`, etc., `breakpoint` for breakpoint setting and clearing commands, and `debug` for other debug, stepping, and continue commands. In addition, if the variable is set to the single symbol `'everything`, all the copious shell input is displayed (which is probably only useful for debugging purposes). N.B. For hidden commands which produce output by side-effect, that output remains hidden (e.g., stepping through a `print` command). As a special case, any error message in the output will be displayed (e.g., stepping to an error).

### 5.4 Debugging IDL Programs

Programs can be compiled, run, and debugged directly from the source buffer in Emacs, walking through arbitrarily deeply nested code, printing expressions and skipping up and down the calling stack along the way. IDLWAVE makes compiling and debugging IDL programs far less cumbersome by providing a full-featured, key/menu/toolbar-driven interface to commands like `breakpoint`, `.step`, `.run`, etc. It can even perform complex debug operations not natively supported by IDL (like continuing to the line at the cursor).

The IDLWAVE shell installs key bindings both in the shell buffer and in all IDL code buffers of the current Emacs session, so debug commands work in both places (in the shell, commands operate on the last file compiled). On Emacs versions which support it, a debugging toolbar is also installed. The toolbar display can be toggled with `C-c C-d C-t` (`idlwave-shell-toggle-toolbar`).

`idlwave-shell-use-toolbar (t)` [User Option]

Non-`nil` means use the debugging toolbar in all IDL related buffers.

### 5.4.1 A Tale of Two Modes

The many debugging, compiling, and examination commands provided in IDLWAVE are available simultaneously through two different interfaces: the original, multi-key command interface, and the new Electric Debug Mode. The functionality they offer is similar, but the way you interact with them is quite different. The main difference is that, in Electric Debug Mode, the source buffers are made read-only, and single key-strokes are used to step through, examine expressions, set and remove breakpoints, etc. The same variables, prefix arguments, and settings apply to both versions, and both can be used interchangeably. By default, when breakpoints are hit, Electric Debug Mode is enabled. The traditional interface is described first. See [Section 5.4.6 \[Electric Debug Mode\], page 36](#), for more on that mode. Note that electric debug mode can be prevented from activating automatically by customizing the variable `idlwave-shell-automatic-electric-debug`.

### 5.4.2 Debug Key Bindings

The standard debugging key bindings are always available by default on the prefix key `C-c C-d`, so, for example, setting a breakpoint is done with `C-c C-d C-b`, and compiling a source file with `C-c C-d C-c`. You can also easily configure IDLWAVE to use one or more modifier keys not in use by other commands, in lieu of the prefix `C-c C-d` (though these bindings will typically also be available — see `idlwave-shell-activate-prefix-keybindings`). For example, if you include in `.emacs`:

```
(setq idlwave-shell-debug-modifiers '(control shift))
```

a breakpoint can then be set by pressing `b` while holding down `shift` and `control` keys, i.e. `C-S-b`. Compiling a source file will be on `C-S-c`, deleting a breakpoint `C-S-d`, etc. In the remainder of this chapter we will assume that the `C-c C-d` bindings are active, but each of these bindings will have an equivalent shortcut if modifiers are given in the `idlwave-shell-debug-modifiers` variable (see [Section 3.2 \[Lesson II – Customization\], page 6](#)). A much simpler and faster form of debugging for running code is also available by default — see [Section 5.4.6 \[Electric Debug Mode\], page 36](#).

`idlwave-shell-prefix-key` (`C-c C-d`) [User Option]

The prefix key for the debugging map `idlwave-shell-mode-prefix-map`.

`idlwave-shell-activate-prefix-keybindings` (`t`) [User Option]

Non-`nil` means debug commands will be bound to the prefix key, like `C-c C-d C-b`.

`idlwave-shell-debug-modifiers` (`nil`) [User Option]

List of modifier keys to use for additional, alternative binding of debugging commands in the shell and source buffers. Can be one or more of `control`, `meta`, `super`, `hyper`, `alt`, and `shift`.

### 5.4.3 Breakpoints and Stepping

IDLWAVE helps you set breakpoints and step through code. Setting a breakpoint in the current line of the source buffer is accomplished with `C-c C-d C-b` (`idlwave-shell-break-here`). With a prefix arg of 1 (i.e. `C-1 C-c C-d C-b`), the breakpoint gets a `/ONCE` keyword, meaning that it will be deleted after first use. With a numeric prefix greater than one (e.g. `C-4 C-c C-d C-b`), the breakpoint will only be active the `n`th time it is hit. With a single non-numeric prefix (i.e. `C-u C-c C-d C-b`), prompt for a condition — an IDL expression

to be evaluated and trigger the breakpoint only if true. To clear the breakpoint in the current line, use `C-c C-d C-d` (`idlwave-clear-current-bp`). When executed from the shell window, the breakpoint where IDL is currently stopped will be deleted. To clear all breakpoints, use `C-c C-d C-a` (`idlwave-clear-all-bp`). Breakpoints can also be disabled and re-enabled: `C-c C-d C-\` (`idlwave-shell-toggle-enable-current-bp`).

Breakpoint lines are highlighted or indicated with an icon in the source code (different icons for conditional, after, and other break types). Disabled breakpoints are *grayed out* by default. Note that IDL places breakpoints as close as possible on or after the line you specify. IDLWAVE queries the shell for the actual breakpoint location which was set, so the exact line you specify may not be marked. You can re-sync the breakpoint list and update the display at any time (e.g., if you add or remove some on the command line) using `C-c C-d C-l`.

In recent IDLWAVE versions, the breakpoint line is highlighted when the mouse is moved over it, and a tooltip pops up describing the break details. `Mouse-3` on the breakpoint line pops up a menu of breakpoint actions, including clearing, disabling, and adding or changing break conditions or “after” break count.

Once the program has stopped somewhere, you can step through it. The most important stepping commands are `C-c C-d C-s` to execute one line of IDL code (“step into”); `C-c C-d C-n` to step a single line, treating procedure and function calls as a single step (“step over”); `C-c C-d C-h` to continue execution to the line at the cursor and `C-c C-d C-r` to continue execution. See [Section 5.3 \[Commands Sent to the Shell\], page 32](#), for information on displaying or hiding the breakpoint and stepping commands the shell receives. Here is a summary of the breakpoint and stepping commands:

<code>C-c C-d C-b</code>	Set breakpoint ( <code>idlwave-shell-break-here</code> )
<code>C-c C-d C-i</code>	Set breakpoint in module named here ( <code>idlwave-shell-break-in</code> )
<code>C-c C-d C-d</code>	Clear current breakpoint ( <code>idlwave-shell-clear-current-bp</code> )
<code>C-c C-d C-a</code>	Clear all breakpoints ( <code>idlwave-shell-clear-all-bp</code> )
<code>C-c C-d [</code>	Go to the previous breakpoint ( <code>idlwave-shell-goto-previous-bp</code> )
<code>C-c C-d ]</code>	Go to the next breakpoint ( <code>idlwave-shell-goto-next-bp</code> )
<code>C-c C-d C-\</code>	Disable/Enable current breakpoint ( <code>idlwave-shell-toggle-enable-current-bp</code> )
<code>C-c C-d C-j</code>	Set a breakpoint at the beginning of the enclosing routine.
<code>C-c C-d C-s</code>	Step, into function calls ( <code>idlwave-shell-step</code> )
<code>C-c C-d C-n</code>	Step, over function calls ( <code>idlwave-shell-stepover</code> )
<code>C-c C-d C-k</code>	Skip one statement ( <code>idlwave-shell-skip</code> )
<code>C-c C-d C-u</code>	Continue to end of block ( <code>idlwave-shell-up</code> )
<code>C-c C-d C-m</code>	Continue to end of function ( <code>idlwave-shell-return</code> )
<code>C-c C-d C-o</code>	Continue past end of function ( <code>idlwave-shell-out</code> )
<code>C-c C-d C-h</code>	Continue to line at cursor position ( <code>idlwave-shell-to-here</code> )
<code>C-c C-d C-r</code>	Continue execution to next breakpoint, if any ( <code>idlwave-shell-cont</code> )
<code>C-c C-d C-up</code>	Show higher level in calling stack ( <code>idlwave-shell-stack-up</code> )
<code>C-c C-d C-down</code>	Show lower level in calling stack ( <code>idlwave-shell-stack-down</code> )

All of these commands have equivalents in Electric Debug Mode, which provides faster single-key access (see [Section 5.4.6 \[Electric Debug Mode\]](#), page 36).

The line where IDL is currently stopped, at breakpoints, halts, and errors, etc., is marked with a color overlay or arrow, depending on the setting in `idlwave-shell-mark-stop-line`. If an overlay face is used to mark the stop line (as it is by default), when stepping through code, the face color is temporarily changed to gray, until IDL completes the next command and moves to the new line.

`idlwave-shell-mark-breakpoints` (t) [User Option]

Non-`nil` means mark breakpoints in the source file buffers. The value indicates the preferred method. Valid values are `nil`, `t`, `face`, and `glyph`.

`idlwave-shell-breakpoint-face` [User Option]

The face for breakpoint lines in the source code if `idlwave-shell-mark-breakpoints` has the value `face`.

`idlwave-shell-breakpoint-popup-menu` (t) [User Option]

Whether to pop-up a menu and present a tooltip description on breakpoint lines.

`idlwave-shell-mark-stop-line` (t) [User Option]

Non-`nil` means mark the source code line where IDL is currently stopped. The value specifies the preferred method. Valid values are `nil`, `t`, `arrow`, and `face`.

`idlwave-shell-overlay-arrow` (">") [User Option]

The overlay arrow to display at source lines where execution halts, if configured in `idlwave-shell-mark-stop-line`.

`idlwave-shell-stop-line-face` [User Option]

The face which highlights the source line where IDL is stopped, if configured in `idlwave-shell-mark-stop-line`.

#### 5.4.4 Compiling Programs

In order to compile the current buffer under the IDLWAVE shell, press `C-c C-d C-c` (`idlwave-save-and-run`). This first saves the current buffer and then sends the command `‘.run path/to/file’` to the shell. You can also execute `C-c C-d C-c` from the shell buffer, in which case the most recently compiled buffer will be saved and re-compiled.

When developing or debugging a program, it is often necessary to execute the same command line many times. A convenient way to do this is `C-c C-d C-y` (`idlwave-shell-execute-default-command-line`). This command first resets IDL from a state of interrupted execution by closing all files and returning to the main interpreter level. Then a default command line is sent to the shell. To edit the default command line, call `idlwave-shell-execute-default-command-line` with a prefix argument: `C-u C-c C-d C-y`. If no default command line has been set (or you give two prefix arguments), the last command on the `comint` input history is sent.

#### 5.4.5 Walking the Calling Stack

While debugging a program, it can be very useful to check the context in which the current routine was called, for instance to help understand the value of the arguments passed. To do

so conveniently you need to examine the calling stack. If execution is stopped somewhere deep in a program, you can use the commands `C-c C-d C-UP` (`idlwave-shell-stack-up`) and `C-c C-d C-DOWN` (`idlwave-shell-stack-down`), or the corresponding toolbar buttons, to move up or down through the calling stack. The mode line of the shell window will indicate the position within the stack with a label like `[-3:MYPRO]`. The line of IDL code at that stack position will be highlighted. If you continue execution, IDLWAVE will automatically return to the current level. See [Section 5.5 \[Examining Variables\]](#), page 38, for information how to examine the value of variables and expressions on higher calling stack levels.

### 5.4.6 Electric Debug Mode

Even with a convenient debug key prefix enabled, repetitive stepping, variable examination (see [Section 5.5 \[Examining Variables\]](#), page 38), and other debugging activities can be awkward and slow using commands which require multiple keystrokes. Luckily, there's a better way, inspired by the lisp e-debug mode, and available through the *Electric Debug Mode*. By default, as soon as a breakpoint is hit, this minor mode is enabled. The buffer showing the line where execution has halted is switched to Electric Debug Mode. This mode is visible as `*Debugging*` in the mode line, and a different face (violet by default, if color is available) for the line stopped at point. The buffer is made read-only and single-character bindings for the most commonly used debugging commands are enabled. These character commands (a list of which is available with `C-?`) are:

<code>a</code>	Clear all breakpoints ( <code>idlwave-shell-clear-all-bp</code> )
<code>b</code>	Set breakpoint, <code>C-u b</code> for a conditional break, <code>C-n b</code> for nth hit ( <code>idlwave-shell-break-here</code> )
<code>d</code>	Clear current breakpoint ( <code>idlwave-shell-clear-current-bp</code> )
<code>e</code>	Prompt for expression to print ( <code>idlwave-shell-clear-current-bp</code> ).
<code>h</code>	Continue to the line at cursor position ( <code>idlwave-shell-to-here</code> )
<code>i</code>	Set breakpoint in module named here ( <code>idlwave-shell-break-in</code> )
<code>[</code>	Go to the previous breakpoint in the file ( <code>idlwave-shell-goto-previous-bp</code> )
<code>]</code>	Go to the next breakpoint in the file ( <code>idlwave-shell-goto-next-bp</code> )
<code>\</code>	Disable/Enable current breakpoint ( <code>idlwave-shell-toggle-enable-current-bp</code> )
<code>j</code>	Set breakpoint at beginning of enclosing routine ( <code>idlwave-shell-break-this-module</code> )
<code>k</code>	Skip one statement ( <code>idlwave-shell-skip</code> )
<code>m</code>	Continue to end of function ( <code>idlwave-shell-return</code> )
<code>n</code>	Step, over function calls ( <code>idlwave-shell-stepover</code> )
<code>o</code>	Continue past end of function ( <code>idlwave-shell-out</code> )
<code>p</code>	Print expression near point or in region with <code>C-u p</code> ( <code>idlwave-shell-print</code> )
<code>q</code>	End the debugging session and return to the Shell's main level ( <code>idlwave-shell-retall</code> )
<code>r</code>	Continue execution to next breakpoint, if any ( <code>idlwave-shell-cont</code> )
<code>s</code> or <code><u>SPACE</u></code>	Step, into function calls ( <code>idlwave-shell-step</code> )
<code>t</code>	Print a calling-level traceback in the shell

<code>u</code>	Continue to end of block ( <code>idlwave-shell-up</code> )
<code>v</code>	Turn Electric Debug Mode off ( <code>idlwave-shell-electric-debug-mode</code> )
<code>x</code>	Examine expression near point (or in region with <code>C-u x</code> ) with shortcut of examine type.
<code>z</code>	Reset IDL ( <code>idlwave-shell-reset</code> )
<code>+ or =</code>	Show higher level in calling stack ( <code>idlwave-shell-stack-up</code> )
<code>- or _</code>	Show lower level in calling stack ( <code>idlwave-shell-stack-down</code> )
<code>?</code>	Help on expression near point or in region with <code>C-u ?</code> ( <code>idlwave-shell-help-expression</code> )
<code>C-?</code>	Show help on the commands available.

Most single-character electric debug bindings use the final keystroke of the equivalent multiple key commands (which are of course also still available), but some differ (e.g. `e,t,q,x`). Some have additional convenience bindings (like `<SPACE>` for stepping). All prefix and other argument options described in this section for the commands invoked by electric debug bindings are still valid. For example, `C-u b` sets a conditional breakpoint, just as it did with `C-u C-c C-d C-b`.

You can toggle the electric debug mode at any time in a buffer using `C-c C-d C-v` (`v` to turn it off while in the mode), or from the Debug menu. Normally the mode will be enabled and disabled at the appropriate times, but occasionally you might want to edit a file while still debugging it, or switch to the mode for conveniently setting lots of breakpoints.

To quickly abandon a debugging session and return to normal editing at the Shell's main level, use `q` (`idlwave-shell-retail`). This disables electric debug mode in all IDLWAVE buffers<sup>2</sup>. Help is available for the command shortcuts with `C-?`. If you find this mode gets in your way, you can keep it from automatically activating by setting the variable `idlwave-shell-automatic-electric-debug` to `nil`, or `'breakpoint`. If you'd like the convenient electric debug shortcuts available also when run-time errors are encountered, set to `t`.

`idlwave-shell-automatic-electric-debug ('breakpoint)` [User Option]

Whether to enter electric debug mode automatically when a breakpoint or run-time error is encountered, and then disable it in all buffers when the `$MAIN$` level is reached (either through normal program execution, or retail). In addition to `nil` for never, and `t` for both breakpoints and errors, this can be `'breakpoint` (the default) to enable it only at breakpoint halts.

`idlwave-shell-electric-stop-color (Violet)` [User Option]

Default color of the stopped line overlay when in electric debug mode.

`idlwave-shell-electric-stop-line-face` [User Option]

The face to use for the stopped line. Defaults to a face similar to the modeline, with color `idlwave-shell-electric-stop-color`.

`idlwave-shell-electric-zap-to-file (t)` [User Option]

If set, when entering electric debug mode, select the window displaying the file where point is stopped. This takes point away from the shell window, but is useful for immediate stepping, etc.

<sup>2</sup> Note that this binding is not symmetric: `C-c C-d C-q` is bound to `idlwave-shell-quit`, which quits your IDL session.

## 5.5 Examining Variables

Do you find yourself repeatedly typing, e.g. `print,n_elements(x)`, and similar statements to remind yourself of the type/size/structure/value/etc. of variables and expressions in your code or at the command line? IDLWAVE has a suite of special commands to automate these types of variable or expression examinations. They work by sending statements to the shell formatted to include the indicated expression, and can be accessed in several ways.

These *examine* commands can be used in the shell or buffer at any time (as long as the shell is running), and are very useful when execution is stopped in a buffer due to a triggered breakpoint or error, or while composing a long command in the IDLWAVE shell. In the latter case, the command is sent to the shell and its output is visible, but point remains unmoved in the command being composed — you can inspect the constituents of a command you’re building without interrupting the process of building it! You can even print arbitrary expressions from older input or output further up in the shell window — any expression, variable, number, or function you see can be examined.

If the variable `idlwave-shell-separate-examine-output` is non-nil (the default), all examine output will be sent to a special ‘\*Examine\*’ buffer, rather than the shell. The output of prior examine commands is saved in this buffer. In this buffer `␣` clears the contents, and `␣` hides the buffer.

The two most basic examine commands are bound to `C-c C-d C-p`, to print the expression at point, and `C-c C-d ?`, to invoke help on this expression<sup>3</sup>. The expression at point is either an array expression or a function call, or the contents of a pair of parentheses. The chosen expression is highlighted, and simultaneously the resulting output is highlighted in the shell or separate output buffer. Calling the above commands with a prefix argument will use the current region as expression instead of using the one at point. `which` can be useful for examining complicated, multi-line expressions. Two prefix arguments (`C-u C-u C-c C-d C-p`) will prompt for an expression to print directly. By default, when invoking `print`, only an initial portion of long arrays will be printed, up to `idlwave-shell-max-print-length`.

For added speed and convenience, there are mouse bindings which allow you to click on expressions and examine their values. Use `S-Mouse-2` to print an expression and `C-M-Mouse-2` to invoke help (i.e. you need to hold down `⌘` and `⌘` while clicking with the middle mouse button). If you simply click, the nearest expression will be selected in the same manner as described above. You can also *drag* the mouse in order to highlight exactly the specific expression or sub-expression you want to examine. For custom expression examination, and the powerful customizable pop-up examine selection, See [Section 5.6 \[Custom Expression Examination\]](#), page 39.

The same variable inspection commands work both in the IDL Shell and IDLWAVE buffers, and even for variables at higher levels of the calling stack. For instance, if you’re stopped at a breakpoint in a routine, you can examine the values of variables and expressions inside its calling routine, and so on, all the way up through the calling stack. Simply step up the stack, and print variables as you see them (see [Section 5.4.5 \[Walking the Calling Stack\]](#), page 35, for information on stepping back through the calling stack). The following restrictions apply for all levels except the current:

- Array expressions must use the ‘[ ]’ index delimiters. Identifiers with a ‘( )’ will be interpreted as function calls.

<sup>3</sup> Available as `p` and `?` in Electric Debug Mode (see [Section 5.4.6 \[Electric Debug Mode\]](#), page 36)

- N.B.: printing values of expressions on higher levels of the calling stack uses the *unsupported* IDL routine `ROUTINE_NAMES`, which may or may not be available in future versions of IDL. Caveat Examinor.

`idlwave-shell-expression-face` [User Option]  
The face for `idlwave-shell-expression-overlay`. Allows you to choose the font, color and other properties for the expression printed by IDL.

`idlwave-shell-output-face` [User Option]  
The face for `idlwave-shell-output-overlay`. Allows to choose the font, color and other properties for the most recent output of IDL when examining an expression."

`idlwave-shell-separate-examine-output` (t) [User Option]  
If non-nil, re-direct the output of examine commands to a special `*Examine*` buffer, instead of in the shell itself.

`idlwave-shell-max-print-length` (200) [User Option]  
The maximum number of leading array entries to print, when examining array expressions.

## 5.6 Custom Expression Examination

The variety of possible variable and expression examination commands is endless (just look, for instance, at the keyword list to `widget_info()`). Rather than attempt to include them all, IDLWAVE provides two easy methods to customize your own commands, with a special mouse examine command, and two macros for generating your own examine key and mouse bindings.

The most powerful and flexible mouse examine command of all is available on `C-S-Mouse-2`. Just as for all the other mouse examine commands, it permits click or drag expression selection, but instead of sending hard-coded commands to the shell, it pops-up a customizable selection list of examine functions to choose among, configured with the `idlwave-shell-examine-alist` variable<sup>4</sup>. This variable is a list of key-value pairs (an *alist* in Emacs parlance), where the key gives a name to be shown for the examine command, and the value is the command strings itself, in which the text `___` (three underscores) will be replaced by the selected expression before being sent to the shell. An example might be key `Structure Help` with value `help,___,/STRUCTURE`. In that case, you'd be prompted with *Structure Help*, which might send something like `help,var,/STRUCTURE` to the shell for output. `idlwave-shell-examine-alist` comes configured by default with a large list of examine commands, but you can easily customize it to add your own.

In addition to configuring the functions available to the pop-up mouse command, you can easily create your own customized bindings to inspect expressions using the two convenience macros `idlwave-shell-examine` and `idlwave-shell-mouse-examine`. These create keyboard or mouse-based custom inspections of variables, sharing all the same properties of the built-in examine commands. Both functions take a single string argument sharing the syntax of the `idlwave-shell-examine-alist` values, e.g.:

<sup>4</sup> In Electric Debug Mode (see [Section 5.4.6 \[Electric Debug Mode\]](#), page 36), the key `x` provides a single-character shortcut interface to the same examine functions for the expression at point or marked by the region.

```
(add-hook 'idlwave-shell-mode-hook
  (lambda ()
    (idlwave-shell-define-key-both [s-down-mouse-2]
      (idlwave-shell-mouse-examine
        "print, size(___,/DIMENSIONS)"))
    (idlwave-shell-define-key-both [f9] (idlwave-shell-examine
      "print, size(___,/DIMENSIONS)"))
    (idlwave-shell-define-key-both [f10] (idlwave-shell-examine
      "print,size(___,/TNAME)"))
    (idlwave-shell-define-key-both [f11] (idlwave-shell-examine
      "help,___,/STRUCTURE")))))
```

Now pressing `(f9)`, or middle-mouse dragging with the `(SUPER)` key depressed, will print the dimensions of the nearby or highlighted expression. Pressing `(f10)` will give the type string, and `(f11)` will show the contents of a nearby structure. As you can see, the possibilities are only marginally finite.

`idlwave-shell-examine-alist` [User Option]

An alist of examine commands in which the keys name the command and are displayed in the selection pop-up, and the values are custom IDL examine command strings to send, after all instances of `___` (three underscores) are replaced by the indicated expression.

## 6 Installation

### 6.1 Installing IDLWAVE

IDLWAVE is part of Emacs 21.1 and later. It is also an XEmacs package and can be installed from [the XEmacs ftp site](#) with the normal package management system on XEmacs 21. You can also download IDLWAVE and install it yourself from [the maintainers webpage](#). Follow the instructions in the INSTALL file.

### 6.2 Installing Online Help

Starting with IDL v6.2, all necessary online help files and routine information are distributed directly with IDL. Nothing additional is required.

For version of IDL prior to 6.2 (and IDLWAVE prior to version 6.0), if you want to use the online help display, an additional set of files (HTML versions of the IDL documentation) must be installed. These files can also be downloaded from [the maintainers webpage](#). You need to place the files somewhere on your system and tell IDLWAVE where they are with:

```
; e.g. /usr/local/etc/  
(setq idlwave-html-help-location "/path/to/help/dir/")
```

The default location is `‘/usr/local/etc’`, and if you install the directory there, you do not need to set this variable. Note that the help package only changes with new versions of the IDL documentation, and need not be updated unless your version of IDL changes. Since the help system is distributed with IDL starting at version 6.2, no new help packages will be created for these versions.

## 7 Acknowledgements

The main contributors to the IDLWAVE package have been:

- **Chris Chase**, the original author. Chris wrote ‘idl.e1’ and ‘idl-shell.e1’ and maintained them for several years.
- **Carsten Dominik** was in charge of the package from version 3.0, during which time he overhauled almost everything, modernized IDLWAVE with many new features, and developed the manual.
- **J.D. Smith**, the current maintainer, as of version 4.10, helped shape object method completion and most new features introduced in versions 4.x, and introduced many new features for IDLWAVE versions 5.x and 6.x.

The following people have also contributed to the development of IDLWAVE with patches, ideas, bug reports and suggestions.

- Ulrik Dickow <dickow\_\_at\_\_nbi.dk>
- Eric E. Dors <edors\_\_at\_\_lanl.gov>
- Stein Vidar H. Haugan <s.v.h.haugan\_\_at\_\_astro.uio.no>
- David Huenemoerder <dph\_\_at\_\_space.mit.edu>
- Kevin Ivory <Kevin.Ivory\_\_at\_\_linmpi.mpg.de>
- Dick Jackson <dick\_\_at\_\_d-jackson.com>
- Xuyong Liu <liu\_\_at\_\_stsci.edu>
- Simon Marshall <Simon.Marshall\_\_at\_\_esrin.esa.it>
- Craig Markwardt <craigm\_\_at\_\_cow.physics.wisc.edu>
- Laurent Mugnier <mugnier\_\_at\_\_onera.fr>
- Lubos Pochman <lubos\_\_at\_\_rsinc.com>
- Bob Portmann <portmann\_\_at\_\_al.noaa.gov>
- Patrick M. Ryan <pat\_\_at\_\_jaameri.gsfc.nasa.gov>
- Marty Ryba <ryba\_\_at\_\_ll.mit.edu>
- Phil Williams <williams\_\_at\_\_irc.chmcc.org>
- Phil Sterne <sterne\_\_at\_\_dublin.llnl.gov>
- Paul Sorenson <aardvark62\_\_at\_\_msn.com>

Doug Dirks was instrumental in providing the crucial IDL XML catalog to support HTML help with IDL v6.2 and later, and Ali Bahrami provided scripts and documentation to interface with the IDL Assistant.

Thanks to everyone!

## Appendix A Sources of Routine Info

In [Section 4.2 \[Routine Info\]](#), [page 12](#) and [Section 4.4 \[Completion\]](#), [page 18](#) we showed how IDLWAVE displays the calling sequence and keywords of routines, and completes routine names and keywords. For these features to work, IDLWAVE must know about the accessible routines.

### A.1 Routine Definitions

Routines which can be used in an IDL program can be defined in several places:

1. *Builtin routines* are defined inside IDL itself. The source code of such routines is not available, but instead are learned about through the IDL documentation.
2. Routines which are *part of the current program*, are defined in a file explicitly compiled by the user. This file may or may not be located on the IDL search path.
3. *Library routines* are defined in files located on IDL's search path. When a library routine is called for the first time, IDL will find the source file and compile it dynamically. A special sub-category of library routines are the *system routines* distributed with IDL, and usually available in the 'lib' subdirectory of the IDL distribution.
4. External routines written in other languages (like Fortran or C) can be called with `CALL_EXTERNAL`, linked into IDL via `LINKIMAGE`, or included as dynamically loaded modules (DLMS). Currently IDLWAVE cannot provide routine info and completion for such external routines, except by querying the Shell for calling information (DLMS only).

### A.2 Routine Information Sources

To maintain the most comprehensive information about all IDL routines on a system, IDLWAVE collects data from many sources:

1. It has a *builtin list* with information about the routines IDL ships with. IDLWAVE 6.0 is distributed with a list of 1966 routines, reflecting IDL version 6.2. As of IDL v6.2, the routine info is distributed directly with IDL in the form of an XML catalog which IDLWAVE scans. Formerly, this list was created by scanning the IDL manuals to produce the file 'idlw-rinfo.el'.
2. IDLWAVE *scans* all its *buffers* in the current Emacs session for routine definitions. This is done automatically when routine information or completion is first requested by the user. Each new buffer and each buffer saved after making changes is also scanned. The command `C-c C-i` (`idlwave-update-routine-info`) can be used at any time to rescan all buffers.
3. If you have an IDLWAVE-Shell running in the Emacs session, IDLWAVE will *query the shell* for compiled routines and their arguments. This happens automatically when routine information or completion is first requested by the user. Each time an Emacs buffer is compiled with `C-c C-d C-c`, the routine info for that file is queried. Though rarely necessary, the command `C-c C-i` (`idlwave-update-routine-info`) can be used to explicitly update the shell routine data.
4. Many popular libraries are distributed with routine information already scanned into *library catalogs* (see [Section A.3.1 \[Library Catalogs\]](#), [page 45](#)). These per-directory

catalog files can also be built by the user with the supplied ‘`idlwave_catalog`’ tool. They are automatically discovered by IDLWAVE.

- IDLWAVE can scan selected directories of source files and store the result in a single *user catalog* file which will be automatically loaded just like ‘`idlw-rinfo.el`’. See [Section A.3.2 \[User Catalog\], page 46](#), for information on how to scan files in this way.

Loading all the routine and catalog information can be a time consuming process, especially over slow networks. Depending on the system and network configuration it could take up to 30 seconds (though locally on fast systems is usually only a few seconds). In order to minimize the wait time upon your first completion or routine info command in a session, IDLWAVE uses Emacs idle time to do the initialization in six steps, yielding to user input in between. If this gets into your way, set the variable `idlwave-init-rinfo-when-idle-after` to 0 (zero). The more routines documented in library and user catalogs, the slower the loading will be, so reducing this number can help alleviate any long load times.

`idlwave-init-rinfo-when-idle-after` (10) [User Option]

Seconds of idle time before routine info is automatically initialized.

`idlwave-scan-all-buffers-for-routine-info` (t) [User Option]

Non-`nil` means scan all buffers for IDL programs when updating info.

`idlwave-query-shell-for-routine-info` (t) [User Option]

Non-`nil` means query the shell for info about compiled routines.

`idlwave-auto-routine-info-updates` [User Option]

Controls under what circumstances routine info is updated automatically.

### A.3 Catalogs

*Catalogs* are files containing scanned information on individual routines, including arguments and keywords, calling sequence, file path, class and procedure vs. function type, etc. They represent a way of extending the internal built-in information available for IDL system routines (see [Section 4.2 \[Routine Info\], page 12](#)) to other source collections.

Starting with version 5.0, there are two types of catalogs available with IDLWAVE. The traditional *user catalog* and the newer *library catalogs*. Although they can be used interchangeably, the library catalogs are more flexible, and preferred. There are few occasions when a user catalog might be preferred — read below. Both types of catalogs can coexist without causing problems.

To facilitate the catalog systems, IDLWAVE stores information it gathers from the shell about the IDL search paths, and can write this information out automatically, or on-demand (menu `Debug->Save Path Info`). On systems with no shell from which to discover the path information (e.g. Windows), a library path must be specified in `idlwave-library-path` to allow library catalogs to be located, and to setup directories for user catalog scan (see [Section A.3.2 \[User Catalog\], page 46](#) for more on this variable). Note that, before the shell is running, IDLWAVE can only know about the IDL search path by consulting the file pointed to by `idlwave-path-file` (‘`~/idlwave/idlpath.el`’, by default). If `idlwave-auto-write-path` is enabled (which is the default), the paths are written out whenever the IDLWAVE shell is started.

- idlwave-auto-write-path (t)** [User Option]  
Write out information on the !PATH and !DIR paths from IDL automatically when they change and when the Shell is closed. These paths are needed to locate library catalogs.
- idlwave-library-path** [User Option]  
IDL library path for Windows and MacOS. Under Unix/MacOSX, will be obtained from the Shell when run.
- idlwave-system-directory** [User Option]  
The IDL system directory for Windows and MacOS. Also needed for locating HTML help and the IDL Assistant for IDL v6.2 and later. Under Unix/MacOSX, will be obtained from the Shell and recorded, if run.
- idlwave-config-directory (~/.idlwave)** [User Option]  
Default path where IDLWAVE saves configuration information, a user catalog (if any), and a cached scan of the XML catalog (IDL v6.2 and later).

### A.3.1 Library Catalogs

Library catalogs consist of files named `‘.idlwave_catalog’` stored in directories containing `.pro` routine files. They are discovered on the IDL search path and loaded automatically when routine information is read. Each catalog file documents the routines found in that directory — one catalog per directory. Every catalog has a library name associated with it (e.g. *AstroLib*). This name will be shown briefly when the catalog is found, and in the routine info of routines it documents.

Many popular libraries of routines are shipped with IDLWAVE catalog files by default, and so will be automatically discovered. Library catalogs are scanned externally to Emacs using a tool provided with IDLWAVE. Each catalog can be re-scanned independently of any other. Catalogs can easily be made available system-wide with a common source repository, providing uniform routine information, and lifting the burden of scanning from the user (who may not even know they’re using a scanned catalog). Since all catalogs are independent, they can be re-scanned automatically to gather updates, e.g. in a ‘cron’ job. Scanning is much faster than with the built-in user catalog method. One minor disadvantage: the entire IDL search path is scanned for catalog files every time IDLWAVE starts up, which might be slow if accessing IDL routines over a slow network.

A Perl tool to create library catalogs is distributed with IDLWAVE: `idlwave_catalog`. It can be called quite simply:

```
idlwave_catalog MyLib
```

This will scan all directories recursively beneath the current and populate them with `‘.idlwave_catalog’` files, tagging the routines found there with the name library “MyLib”. The full usage information:

```
Usage: idlwave_catalog [-l] [-v] [-d] [-s] [-f] [-h] libname
       libname - Unique name of the catalog (4 or more alphanumeric
                characters).
       -l - Scan local directory only, otherwise recursively
            catalog all directories at or beneath this one.
       -v - Print verbose information.
```

```

-d - Instead of scanning, delete all .idlwave_catalog files
    here or below.
-s - Be silent.
-f - Force overwriting any catalogs found with a different
    library name.
-h - Print this usage.

```

To re-load the library catalogs on the IDL path, force a system routine info update using a single prefix to `idlwave-update-routine-info`: `C-u C-c C-i`.

`idlwave-use-library-catalogs` (t) [User Option]  
 Whether to search for and load library catalogs. Disable if load performance is a problem and/or the catalogs are not needed.

### A.3.2 User Catalog

The user catalog is the old routine catalog system. It is produced within Emacs, and stored in a single file in the user's home directory ('`.idlwave/idlusercat.el`' by default). Although library catalogs are more flexible, there may be reasons to prefer a user catalog instead, including:

- The scan is internal to Emacs, so you don't need a working Perl installation, as you do for library catalogs.
- Can be used to scan directories for which the user has no write privileges.
- Easy widget-based path selection.

However, no routine info is available in the user catalog by default; the user must actively complete a scan. In addition, this type of catalog is all or nothing: if a single routine changes, the entire catalog must be rescanned to update it. Creating the user catalog is also much slower than scanning library catalogs.

You can scan any of the directories on the currently known path. Under Windows and MacOS (not OSX), you need to specify the IDL search path in the variable `idlwave-library-path`, and the location of the IDL directory (the value of the `!DIR` system variable) in the variable `idlwave-system-directory`, like this<sup>1</sup>:

```

(setq idlwave-library-path
  ('(+c:/RSI/IDL56/lib/" "+c:/user/me/idllibs"))
(setq idlwave-system-directory "c:/RSI/IDL56/")

```

Under GNU/Linux and UNIX, these values will be automatically gathered from the IDL-WAVE shell, if run.

The command `M-x idlwave-create-user-catalog-file` (or the menu item '`IDLWAVE->Routine Info->Select Catalog Directories`') can then be used to create a user catalog. It brings up a widget in which you can select some or all directories on the search path. Directories which already contain a library catalog are marked with '`[LIB]`', and need not be scanned (although there is no harm if you do so, other than the additional memory used for the duplication).

After selecting directories, click on the '`[Scan & Save]`' button in the widget to scan all files in the selected directories and write out the resulting routine information. In

<sup>1</sup> The initial '+' leads to recursive expansion of the path, just like in IDL

order to update the library information using the directory selection, call the command `idlwave-update-routine-info` with a double prefix argument: `C-u C-u C-c C-i`. This will rescan files in the previously selected directories, write an updated version of the user catalog file and rebuild IDLWAVE's internal lists. If you give three prefix arguments `C-u C-u C-u C-c C-i`, updating will be done with a background job<sup>2</sup>. You can continue to work, and the library catalog will be re-read when it is ready. If you find you need to update the user catalog often, you should consider building a library catalog for your routines instead (see [Section A.3.1 \[Library Catalogs\]](#), page 45).

`idlwave-special-lib-alist` [User Option]  
 Alist of regular expressions matching special library directories for labeling in routine-info display.

## A.4 Load-Path Shadows

IDLWAVE can compile a list of routines which are (re-)defined in more than one file. Since one definition will hide (shadow) the others depending on which file is compiled first, such multiple definitions are called "load-path shadows". IDLWAVE has several routines to scan for load path shadows. The output is placed into the special buffer `*Shadows*`. The format of the output is identical to the source section of the routine info buffer (see [Section 4.2 \[Routine Info\]](#), page 12). The different definitions of a routine are ordered by *likelihood of use*. So the first entry will be most likely the one you'll get if an unsuspecting command uses that routine. Before listing shadows, you should make sure that routine info is up-to-date by pressing `C-c C-i`. Here are the different routines (also available in the Menu `'IDLWAVE->Routine Info'`):

*M-x idlwave-list-buffer-load-path-shadows*

This command checks the names of all routines defined in the current buffer for shadowing conflicts with other routines accessible to IDLWAVE. The command also has a key binding: `C-c C-b`

*M-x idlwave-list-shell-load-path-shadows.*

Checks all routines compiled under the shell for shadowing. This is very useful when you have written a complete application. Just compile the application, use `RESOLVE_ALL` to compile any routines used by your code, update the routine info inside IDLWAVE with `C-c C-i` and then check for shadowing.

*M-x idlwave-list-all-load-path-shadows*

This command checks all routines accessible to IDLWAVE for conflicts.

For these commands to work fully you need to scan the entire load path in either a user or library catalog. Also, IDLWAVE should be able to distinguish between the system library files (normally installed in `'/usr/local/rsi/idl/lib'`) and any site specific or user specific files. Therefore, such local files should not be installed inside the `'lib'` directory of the IDL directory. This is also advisable for many other reasons.

Users of Windows and MacOS (not X) also must set the variable `idlwave-system-directory` to the value of the `!DIR` system variable in IDL. IDLWAVE appends `'lib'` to the value of this variable and assumes that all files found on that path are system routines.

---

<sup>2</sup> Unix systems only, I think.

Another way to find out if a specific routine has multiple definitions on the load path is routine info display (see [Section 4.2 \[Routine Info\]](#), page 12).

## A.5 Documentation Scan

**Starting with version 6.2, IDL is distributed directly with HTML online help, and an XML-based catalog of routine information.** This makes scanning the manuals with the tool ‘get\_html\_rinfo’, and the ‘idlw-rinfo.el’ file it produced, as described here, entirely unnecessary. The information is left here for users wishing to produce a catalog of older IDL versions’ help.

IDLWAVE derives its knowledge about system routines from the IDL manuals. The file ‘idlw-rinfo.el’ contains the routine information for the IDL system routines, and links to relevant sections of the HTML documentation. The Online Help feature of IDLWAVE requires HTML versions of the IDL manuals to be available; the HTML documentation is not distributed with IDLWAVE by default, but must be downloaded separately from [the maintainers webpage](#).

The HTML files and related images can be produced from the ‘idl.chm’ HTMLHelp file distributed with IDL using the free Microsoft HTML Help Workshop. If you are lucky, the maintainer of IDLWAVE will always have access to the newest version of IDL and provide updates. The IDLWAVE distribution also contains the Perl program ‘get\_html\_rinfo’ which constructs the ‘idlw-rinfo.el’ file by scanning the HTML documents produced from the IDL documentation. Instructions on how to use ‘get\_html\_rinfo’ are in the program itself.

## Appendix B HTML Help Browser Tips

There are a wide variety of possible browsers to use for displaying the online HTML help available with IDLWAVE (starting with version 5.0). Since IDL v6.2, a single cross-platform HTML help browser, the *IDL Assistant* is distributed with IDL. If this help browser is available, it is the preferred choice, and the default. The variable `idlwave-help-use-assistant`, enabled by default, controls whether this help browser is used. If you use the IDL Assistant, the tips here are not relevant.

Since IDLWAVE runs on a many different system types, a single browser configuration is not possible, but choices abound. On many systems, the default browser configured in `browse-url-browser-function`, and hence inherited by default by `idlwave-help-browser-function`, is Netscape. Unfortunately, the HTML manuals decompiled from the original source contain formatting structures which Netscape 4.x does not handle well, though they are still readable. A much better choice is Mozilla, or one of the Mozilla-derived browsers such as **Galeon** (GNU/Linux), **Camino** (MacOSX), or **Firebird** (all platforms). Newer versions of Emacs provide a browser-function choice `browse-url-gnome-moz` which uses the Gnome-configured browser.

Note that the HTML files decompiled from the help sources contain specific references to the ‘Symbol’ font, which by default is not permitted in normal encodings (it’s invalid, technically). Though it only impacts a few symbols, you can trick Mozilla-based browsers into recognizing ‘Symbol’ by following the directions [here](#). With this fix in place, HTML help pages look almost identical to their PDF equivalents (yet can be bookmarked, browsed as history, searched, etc.).

Individual platform recommendations:

- Unix/MacOSX: The **w3m** browser and its associated `emacs-w3m` emacs mode provide in-buffer browsing with image display, and excellent speed and formatting. Both the Emacs mode and the browser itself must be downloaded separately. To use this browser, include

```
(setq idlwave-help-browser-function 'w3m-browse-url)
```

in your `.emacs`. Setting a few other nice `w3m` options cuts down on screen clutter:

```
(setq w3m-use-tab nil
      w3m-use-header-line nil
      w3m-use-toolbar nil)
```

If you use a dedicated frame for help, you might want to add the following, to get consistent behavior with the `q` key:

```
;; Close my help window when w3m closes.
(defadvice w3m-close-window (after idlwave-close activate)
  (if (boundp 'idlwave-help-frame)
      (idlwave-help-quit)))
```

Note that you can open the file in an external browser from within `w3m` using `M`.

## Appendix C Configuration Examples

**Question:** You have all these complicated configuration options in your package, but which ones do *you* as the maintainer actually set in your own configuration?

**Answer:** Not many, beyond custom key bindings. I set most defaults the way that seems best. However, the default settings do not turn on features which:

- are not self-evident (i.e. too magic) when used by an unsuspecting user.
- are too intrusive.
- will not work properly on all Emacs installations.
- break with widely used standards.
- use function or other non-standard keys.
- are purely personal customizations, like additional key bindings, and library names.

To see what I mean, here is the *entire* configuration the old maintainer had in his `.emacs`:

```
(setq idlwave-shell-debug-modifiers '(control shift)
      idlwave-store-inquired-class t
      idlwave-shell-automatic-start t
      idlwave-main-block-indent 2
      idlwave-init-rinfo-when-idle-after 2
      idlwave-help-dir "~/lib/emacs/idlwave"
      idlwave-special-lib-alist '(("/idl-astro/" . "AstroLib")
                                   ("/jhuapl/" . "JHUAPL-Lib")
                                   ("/dominik/lib/idl/" . "MyLib")))
```

However, if you are an Emacs power-user and want IDLWAVE to work completely differently, you can change almost every aspect of it. Here is an example of a much more extensive configuration of IDLWAVE. The user is King!

```
;;; Settings for IDLWAVE mode

(setq idlwave-block-indent 3) ; Indentation settings
(setq idlwave-main-block-indent 3)
(setq idlwave-end-offset -3)
(setq idlwave-continuation-indent 1)
(setq idlwave-begin-line-comment "^;[^\;]") ; Leave ";" but not ";;"
                                           ; anchored at start of line.

(setq idlwave-surround-by-blank t) ; Turn on padding ops =,<,>
(setq idlwave-pad-keyword nil) ; Remove spaces for keyword '= '
(setq idlwave-expand-generic-end t) ; convert END to ENDIF etc...
(setq idlwave-reserved-word-upcase t) ; Make reserved words upper case
                                       ; (with abbrevs only)

(setq idlwave-abbrev-change-case nil) ; Don't force case of expansions
(setq idlwave-hang-indent-regexp "- ") ; Change from "- " for auto-fill
(setq idlwave-show-block nil) ; Turn off blinking to begin
(setq idlwave-abbrev-move t) ; Allow abbrevs to move point
(setq idlwave-query-class '((method-default . nil) ; No query for method
                             (keyword-default . nil); or keyword completion
```

```

("INIT" . t)                ; except for these
("CLEANUP" . t)
("SETPROPERTY" .t)
("GETPROPERTY" .t)))

;; Using w3m for help (must install w3m and emacs-w3m)
(autoload 'w3m-browse-url "w3m" "Interface for w3m on Emacs." t)
(setq idlwave-help-browser-function 'w3m-browse-url
      w3m-use-tab nil ; no tabs, location line, or toolbar
      w3m-use-header-line nil
      w3m-use-toolbar nil)

;; Close my help window or frame when w3m closes with 'q'
(defadvice w3m-close-window (after idlwave-close activate)
  (if (boundp 'idlwave-help-frame)
      (idlwave-help-quit)))

;; Some setting can only be done from a mode hook. Here is an example:
(add-hook 'idlwave-mode-hook
  (lambda ()
    (setq case-fold-search nil)          ; Make searches case sensitive
    ;; Run other functions here
    (font-lock-mode 1)                  ; Turn on font-lock mode
    (idlwave-auto-fill-mode 0)          ; Turn off auto filling
    (setq idlwave-help-browser-function 'browse-url-w3)

    ;; Pad with 1 space (if -n is used then make the
    ;; padding a minimum of n spaces.) The defaults use -1
    ;; instead of 1.
    (idlwave-action-and-binding "=" '(idlwave-expand-equal 1 1))
    (idlwave-action-and-binding "<" '(idlwave-surround 1 1))
    (idlwave-action-and-binding ">" '(idlwave-surround 1 1 '(?-)))
    (idlwave-action-and-binding "&" '(idlwave-surround 1 1))

    ;; Only pad after comma and with exactly 1 space
    (idlwave-action-and-binding "," '(idlwave-surround nil 1))
    (idlwave-action-and-binding "&" '(idlwave-surround 1 1))

    ;; Pad only after '->', remove any space before the arrow
    (idlwave-action-and-binding "->" '(idlwave-surround 0 -1 nil 2))

    ;; Set some personal bindings
    ;; (In this case, makes ',' have the normal self-insert behavior.)
    (local-set-key "," 'self-insert-command)
    (local-set-key [f5] 'idlwave-shell-break-here)
    (local-set-key [f6] 'idlwave-shell-clear-current-bp)

```

```

;; Create a newline, indenting the original and new line.
;; A similar function that does _not_ reindent the original
;; line is on "\C-j" (The default for emacs programming modes).
(local-set-key "\n" 'idlwave-newline)
;; (local-set-key "\C-j" 'idlwave-newline) ; My preference.

;; Some personal abbreviations
(define-abbrev idlwave-mode-abbrev-table
  (concat idlwave-abbrev-start-char "wb") "widget_base()"
  (idlwave-keyword-abbrev 1))
(define-abbrev idlwave-mode-abbrev-table
  (concat idlwave-abbrev-start-char "on") "obj_new()"
  (idlwave-keyword-abbrev 1))
))

;;; Settings for IDLWAVE SHELL mode

(setq idlwave-shell-overlay-arrow "=>") ; default is ">"
(setq idlwave-shell-use-dedicated-frame t) ; Make a dedicated frame
(setq idlwave-shell-prompt-pattern "^WAVE> ") ; default is "^IDL> "
(setq idlwave-shell-explicit-file-name "wave")
(setq idlwave-shell-process-name "wave")
(setq idlwave-shell-use-toolbar nil) ; No toolbar

;; Most shell interaction settings can be done from the shell-mode-hook.
(add-hook 'idlwave-shell-mode-hook
  (lambda ()
    ;; Set up some custom key and mouse examine commands
    (idlwave-shell-define-key-both [s-down-mouse-2]
      (idlwave-shell-mouse-examine
        "print, size(___,/DIMENSIONS)"))
    (idlwave-shell-define-key-both [f9] (idlwave-shell-examine
      "print, size(___,/DIMENSIONS)"))
    (idlwave-shell-define-key-both [f10] (idlwave-shell-examine
      "print,size(___,/TNAME)"))
    (idlwave-shell-define-key-both [f11] (idlwave-shell-examine
      "help,___,/STRUCTURE")))))

```

## Appendix D Windows and MacOS

IDLWAVE was developed on a UNIX system. However, thanks to the portability of Emacs, much of IDLWAVE does also work under different operating systems like Windows (with NTEmacs or NTXEmacs) or MacOS.

The only real problem is that there is no command-line version of IDL for Windows or MacOS(<=9) with which IDLWAVE can interact. As a result, the IDLWAVE Shell does not work and you have to rely on IDLDE to run and debug your programs. However, editing IDL source files with Emacs/IDLWAVE works with all bells and whistles, including routine info, completion and fast online help. Only a small amount of additional information must be specified in your `.emacs` file: the path names which, on a UNIX system, are automatically gathered by talking to the IDL program.

Here is an example of the additional configuration needed for a Windows system. I am assuming that IDLWAVE has been installed in `'C:\Program Files\IDLWAVE'` and that IDL is installed in `'C:\RSI\IDL62'`.

```
;; location of the lisp files (only needed if IDLWAVE is not part of
;; your default X/Emacs installation)
(setq load-path (cons "c:/program files/IDLWAVE" load-path))

;; The location of the IDL library directories, both standard, and your own.
;; note that the initial "+" expands the path recursively
(setq idlwave-library-path
  '(+"c:/RSI/IDL55/lib/" "+c:/path/to/my/idllibs" ))

;; location of the IDL system directory (try "print,!DIR")
(setq idlwave-system-directory "c:/RSI/IDL62/")
```

Furthermore, Windows sometimes tries to outsmart you — make sure you check the following things:

- When you download the IDLWAVE distribution, make sure you save the file under the names `'idlwave.tar.gz'`.
- M-TAB switches among running programs — use Esc-TAB instead.
- Other issues as yet unnamed...

Windows users who'd like to make use of IDLWAVE's context-aware HTML help can skip the browser and use the HTMLHelp functionality directly. See [Section 4.3.1 \[Help with HTML Documentation\]](#), page 15.

## Appendix E Troubleshooting

Although IDLWAVE usually installs and works without difficulty, a few common problems and their solutions are documented below.

1. **Whenever an IDL error occurs or a breakpoint is hit, I get errors or strange behavior when I try to type anything into some of my IDLWAVE buffers.**

This is a *feature*, not an error. You're in *Electric Debug Mode* (see [Section 5.4.6 \[Electric Debug Mode\], page 36](#)). You should see `*Debugging*` in the mode-line. The buffer is read-only and all debugging and examination commands are available as single keystrokes; `C-?` lists these shortcuts. Use `q` to quit the mode, and customize the variable `idlwave-shell-automatic-electric-debug` if you prefer not to enter electric debug on breakpoints... but you really should try it before you disable it! You can also customize this variable to enter debug mode when errors are encountered.

2. **I get errors like 'Searching for program: no such file or directory, idl' when attempting to start the IDL shell.**

IDLWAVE needs to know where IDL is in order to run it as a process. By default, it attempts to invoke it simply as `'idl'`, which presumes such an executable is on your search path. You need to ensure `'idl'` is on your `'$PATH'`, or specify the full pathname to the `idl` program with the variable `idlwave-shell-explicit-file-name`. Note that you may need to set your shell search path in two places when running Emacs as an Aqua application with MacOSX; see the next topic.

3. **IDLWAVE is disregarding my 'IDL\_PATH' which I set under MacOSX**

If you run Emacs directly as an Aqua application, rather than from the console shell, the environment is set not from your usual shell configuration files (e.g. `'.cshrc'`), but from the file `'~/MacOSX/environment.plist'`. Either include your path settings there, or start Emacs and IDLWAVE from the shell.

4. **I get errors like 'Symbol's function is void: overlayp'**

You don't have the `'fsf-compat'` package installed, which IDLWAVE needs to run under XEmacs. Install it, or find an XEmacs distribution which includes it by default.

5. **I'm getting errors like 'Symbol's value as variable is void: cl-builtin-gethash' on completion or routine info.**

This error arises if you upgraded Emacs from 20.x to 21.x without re-installing IDLWAVE. Old Emacs and new Emacs are not byte-compatible in compiled lisp files. Presumably, you kept the original `.elc` files in place, and this is the source of the error. If you recompile (or just `"make; make install"`) from source, it should resolve this problem. Another option is to recompile the `'idlw*.el'` files by hand using `M-x byte-compile-file`. Why not take the opportunity to grab the latest IDLWAVE version at [the maintainers webpage](#).

6. **M-(TAB) doesn't complete words, it switches windows on my desktop.**

Your system is trapping `M-(TAB)` and using it for its own nefarious purposes: Emacs never sees the keystrokes. On many Unix systems, you can reconfigure your window manager to use another key sequence for switching among windows. Another option is to use the equivalent sequence `(ESC)-(TAB)`.

7. **When stopping at breakpoints or errors, IDLWAVE does not seem to highlight the relevant line in the source.**

IDLWAVE scans for error and halt messages and highlights the stop location in the correct file. However, if you've changed the system variable '`!ERROR_STATE.MSG_PREFIX`', it is unable to parse these message correctly. Don't do that.

8. **IDLWAVE doesn't work correctly when using ENVI.**

Though IDLWAVE was not written with ENVI in mind, it works just fine with it, as long as you update the prompt it's looking for ('IDL>' by default). You can do this with the variable `idlwave-shell-prompt-pattern` (see [Section 5.1 \[Starting the Shell\]](#), page 29), e.g., in your `.emacs`:

```
(setq idlwave-shell-prompt-pattern "^\\r? ?\\(ENVI\\|IDL\\)> ")
```

9. **Attempts to set breakpoints fail: no breakpoint is indicated in the IDLWAVE buffer.**

IDL changed its breakpoint reporting format starting with IDLv5.5. The first version of IDLWAVE to support the new format is IDLWAVE v4.10. If you have an older version and are using IDL >v5.5, you need to upgrade, and/or make sure your recent version of IDLWAVE is being found on the Emacs load-path (see the next entry). You can list the version being used with `C-h v idlwave-mode-version` [\[RET\]](#).

10. **I installed a new version of IDLWAVE, but the old version is still being used or IDLWAVE works, but when I tried to install the optional modules 'idlw-roprompt.el' or 'idlw-complete-structtag', I get errors like 'Cannot open load file'.**

The problem is that your Emacs is not finding the version of IDLWAVE you installed. Many Emacsen come with an older bundled copy of IDLWAVE (e.g. v4.7 for Emacs 21.x), which is likely what's being used instead. You need to make sure your Emacs *load-path* contains the directory where IDLWAVE is installed ('`/usr/local/share/emacs/site-lisp`', by default), *before* Emacs' default search directories. You can accomplish this by putting the following in your `.emacs`:

```
(setq load-path (cons "/usr/local/share/emacs/site-lisp" load-path))
```

You can check on your load-path value using `C-h v load-path` [\[RET\]](#), and `C-h m` in an IDLWAVE buffer should show you the version Emacs is using.

11. **IDLWAVE is screwing up the formatting of my '.idl' files.**

Actually, this isn't IDLWAVE at all, but '`idl-mode`', an unrelated programming mode for CORBA's Interface Definition Language (you should see '`(IDL)`', not '`(IDLWAVE)`' in the mode-line). One solution: don't name your file '`.idl`', but rather '`.pro`'. Another solution: make sure '`.idl`' files load IDLWAVE instead of '`idl-mode`' by adding the following to your `.emacs`:

```
(setcdr (rassoc 'idl-mode auto-mode-alist) 'idlwave-mode)
```

12. **The routine info for my local routines is out of date!**

IDLWAVE collects routine info from various locations (see [Section A.2 \[Routine Information Sources\]](#), page 43). Routines in files visited in a buffer or compiled in the shell should be up to date. For other routines, the information is only as current as the most recent scan. If you have a rapidly changing set of routines, and you'd like the latest routine information to be available for it, one powerful technique is to make use of the library catalog tool, '`idlwave_catalog`'. Simply add a line to your '`cron`' file ('`crontab -e`' will let you edit this on some systems), like this

```
45 3 * * 1-5 (cd /path/to/myidl-lib; /path/to/idlwave_catalog MyLib)
```

where ‘MyLib’ is the name of your library. This will rescan all ‘.pro’ files at or below ‘/path/to/myidl-lib’ every week night at 3:45am. You can even scan site-wide libraries with this method, and the most recent information will be available to all users. Since the scanning is very fast, there is very little impact.

**13. All the Greek-font characters in the HTML help are displayed as Latin characters!**

Unfortunately, the HTMLHelp files RSI provides attempt to switch to ‘Symbol’ font to display Greek characters, which is not really an permitted method for doing this in HTML. There is a "workaround" for some browsers: See [Appendix B \[HTML Help Browser Tips\]](#), page 49.

**14. In the shell, my long commands are truncated at 256 characters!**

This actually happens when running IDL in an XTerm as well. There are a couple of work arounds: `define_key,/control,'~d'` (e.g. in your ‘\$IDL\_STARTUP’ file) will disable the ‘EOF’ character and give you a 512 character limit. You won’t be able to use `(C-d)` to quit the shell, however. Another possibility is `!EDIT_INPUT=0`, which gives you an *infinite* limit (OK, a memory-bounded limit), but disables the processing of background widget events (those with `/NO_BLOCK` passed to `XManager`).

**15. When I invoke IDL HTML help on a routine, the page which is loaded is one page off, e.g. for CONVERT\_COORD, I get CONTOUR.**

You have a mismatch between your help index and the HTML help package you downloaded. You need to ensure you download a “downgrade kit” if you are using anything older than the latest HTML help package. A new help package appears with each IDL release (assuming the documentation is updated). See [the maintainers webpage](#) for more. Note that, starting with IDL 6.2, the HTML help and its catalog are distributed with IDL, and so should never be inconsistent.

**16. I get errors such as ‘void-variable browse-url-browser-function’ or similar when attempting to load IDLWAVE under XEmacs.**

You don’t have the ‘browse-url’ (or other required) XEmacs package. Unlike GNU Emacs, XEmacs distributes many packages separately from the main program. IDLWAVE is actually among these, but is not always the most up to date. When installing IDLWAVE as an XEmacs package, it should prompt you for required additional packages. When installing it from source, it won’t and you’ll get this error. The easiest solution is to install all the packages when you install XEmacs (the so-called ‘sumo’ bundle). The minimum set of XEmacs packages required by IDLWAVE is ‘`fsf-compat, xemacs-base, mail-lib`’.

# Index

- !**
- !DIR, IDL variable ..... 13, 46, 47
  - !PATH, IDL variable ..... 13, 43
- \***
- '\*Debugging\*' ..... 36
- 
- > ..... 20
- .**
- '`.emacs`' ..... 50
  - '`.idlwave_catalog`' ..... 45
- A**
- Abbreviations ..... 22
  - Acknowledgements ..... 42
  - Actions ..... 24
  - Actions, applied to foreign code ..... 25
  - Active text, in routine info ..... 14
  - Application, testing for shadowing ..... 47
  - Authors, of IDLWAVE ..... 42
  - `auto-fill-mode` ..... 11
- B**
- Block boundary check ..... 25
  - Block, closing ..... 25
  - Breakpoints ..... 33
  - Browser Tips ..... 49
  - Buffer, testing for shadowing ..... 47
  - Buffers, killing ..... 22
  - Buffers, scanning for routine info ..... 12, 43
  - Builtin list of routines ..... 43
- C**
- C-c ? ..... 13
  - C-c C-d ..... 33
  - C-c C-d C-b ..... 33
  - C-c C-d C-c ..... 35
  - C-c C-d C-p ..... 38
  - C-c C-h ..... 27
  - C-c C-i ..... 12, 18
  - C-c C-m ..... 27
  - C-c C-s ..... 29
  - C-c C-v ..... 22
  - C-M-\ ..... 9
  - `CALL_EXTERNAL`, IDL routine ..... 43
  - Calling sequences ..... 13
  - Calling stack, walking ..... 35
  - Cancelling completion ..... 18
  - Case changes ..... 26
  - Case of completed words ..... 19
  - Catalogs ..... 44
  - Categories, of routines ..... 13
  - `cc-mode.el` ..... 1
  - Changelog, in doc header ..... 27
  - Character input mode (Shell) ..... 31
  - Class ambiguity ..... 19
  - Class name completion ..... 18
  - Class query, forcing ..... 20
  - Class tags, in online help ..... 17
  - Closing a block ..... 25
  - Code formatting ..... 9
  - Code indentation ..... 9
  - Code structure, moving through ..... 27
  - Code templates ..... 22
  - Coding standards, enforcing ..... 24
  - Comint ..... 30
  - Comint, Emacs package ..... 29
  - Commands in shell, showing ..... 32
  - Comment indentation ..... 10
  - Compiling library modules ..... 22
  - Compiling programs ..... 35
  - Completion ..... 18
  - Completion, ambiguity ..... 18
  - Completion, cancelling ..... 18
  - Completion, forcing function name ..... 18
  - Completion, in the shell ..... 31
  - Completion, Online Help ..... 18
  - Completion, scrolling ..... 18
  - Completion, structure tag ..... 21
  - Configuration examples ..... 50
  - Context, for online help ..... 15
  - Continuation lines ..... 11
  - Continued statement indentation ..... 9
  - Contributors, to IDLWAVE ..... 42
  - Copyright, of IDLWAVE ..... 2
  - CORBA (Common Object Request Broker Architecture) ..... 1
  - Custom expression examination ..... 39
- D**
- Debugging ..... 32
  - Debugging Interface ..... 33
  - Dedicated frame, for shell buffer ..... 29
  - Default command line, executing ..... 35
  - Default routine, for info and help ..... 13
  - Default settings, of options ..... 50
  - DocLib header ..... 27
  - DocLib header, as online help ..... 17
  - Documentation header ..... 27
  - Downcase, enforcing for reserved words ..... 26

Duplicate routines . . . . . 14, 47

## E

Electric Debug Mode . . . . . 33, 36  
 Emacs, distributed with IDLWAVE . . . . . 41  
 Email address, of Maintainer . . . . . 42  
 END type checking . . . . . 25  
 END, automatic insertion . . . . . 25  
 END, expanding . . . . . 25  
 ENVI . . . . . 29  
 Examining expressions . . . . . 38  
 Example configuration . . . . . 50  
 Executing a default command line . . . . . 35  
 Execution, controlled . . . . . 33  
 Expressions, custom examination . . . . . 39  
 Expressions, printing & help . . . . . 38  
 External routines . . . . . 43

## F

Feature overview . . . . . 1  
 Filling . . . . . 11  
 Flags, in routine info . . . . . 14  
 Font lock . . . . . 12  
 Forcing class query . . . . . 20  
 Foreign code, adapting . . . . . 9, 25  
 Formatting, of code . . . . . 9  
 Frame, for shell buffer . . . . . 29  
 FTP site . . . . . 41  
 ‘Func-menu’, XEmacs package . . . . . 27  
 Function definitions, jumping to . . . . . 27  
 Function name completion . . . . . 18

## G

‘get\_html\_rinfo’ . . . . . 48  
 Getting Started . . . . . 4

## H

Hanging paragraphs . . . . . 10, 11  
 Header, for file documentation . . . . . 27  
 Help using HTML manuals . . . . . 15  
 Help using routine source . . . . . 17  
 HELP, on expressions . . . . . 38  
 Highlighting of syntax . . . . . 12  
 Highlighting of syntax, Octals . . . . . 12  
 Homepage for IDLWAVE . . . . . 41  
 Hooks . . . . . 28, 30  
 HTML Help . . . . . 15

## I

IDL Assistant . . . . . 15  
 IDL library routine info . . . . . 46  
 IDL manual, HTML version . . . . . 15  
 IDL variable !DIR . . . . . 13, 46, 47

IDL variable !PATH . . . . . 13, 43  
 IDL, as Emacs subprocess . . . . . 29  
 ‘idl-shell.el’ . . . . . 1  
 ‘idl.el’ . . . . . 1  
 IDL> Prompt . . . . . 29  
 ‘idlw-help.el’ . . . . . 14  
 ‘idlw-help.txt’ . . . . . 14  
 ‘idlw-rinfo.el’ . . . . . 48  
 IDLWAVE in a Nutshell . . . . . 3  
 IDLWAVE major mode . . . . . 9  
 IDLWAVE shell . . . . . 29  
 IDLWAVE, homepage . . . . . 41  
 idlwave-abbrev-change-case . . . . . 26  
 idlwave-abbrev-move . . . . . 24  
 idlwave-abbrev-start-char . . . . . 24  
 idlwave-auto-fill-split-string . . . . . 11  
 idlwave-auto-routine-info-updates . . . . . 44  
 idlwave-auto-write-path . . . . . 45  
 idlwave-begin-line-comment . . . . . 11  
 idlwave-block-indent . . . . . 9  
 idlwave-class-arrow-face . . . . . 20  
 idlwave-code-comment . . . . . 11  
 idlwave-complete-empty-string-as-lower-case  
 . . . . . 19  
 idlwave-completion-case . . . . . 19  
 idlwave-completion-fontify-classes . . . . . 20  
 idlwave-completion-force-default-case . . . . . 19  
 idlwave-completion-restore-window-  
 configuration . . . . . 19  
 idlwave-completion-show-classes . . . . . 20  
 idlwave-config-directory . . . . . 45  
 idlwave-continuation-indent . . . . . 10  
 idlwave-default-font-lock-items . . . . . 12  
 idlwave-do-actions . . . . . 25  
 idlwave-doc-modifications-keyword . . . . . 27  
 idlwave-doelib-end . . . . . 27  
 idlwave-doelib-start . . . . . 27  
 idlwave-end-offset . . . . . 9  
 idlwave-expand-generic-end . . . . . 25  
 idlwave-extra-help-function . . . . . 17  
 idlwave-file-header . . . . . 27  
 idlwave-fill-comment-line-only . . . . . 11  
 idlwave-function-completion-adds-paren . . . . . 19  
 idlwave-hang-indent-regexp . . . . . 11  
 idlwave-hanging-indent . . . . . 11  
 idlwave-header-to-beginning-of-file . . . . . 27  
 idlwave-help-application . . . . . 28  
 idlwave-help-browser-function . . . . . 16  
 idlwave-help-browser-is-local . . . . . 16  
 idlwave-help-doelib-keyword . . . . . 18  
 idlwave-help-doelib-name . . . . . 17  
 idlwave-help-fontify-source-code . . . . . 17  
 idlwave-help-frame-parameters . . . . . 17  
 idlwave-help-link-face . . . . . 16  
 idlwave-help-source-try-header . . . . . 17  
 idlwave-help-use-assistant . . . . . 16  
 idlwave-help-use-dedicated-frame . . . . . 17

- idlwave-highlight-help-links-in-completion ..... 19
  - idlwave-html-help-location ..... 16
  - idlwave-html-system-help-location ..... 16
  - idlwave-indent-to-open-paren ..... 10
  - idlwave-init-rinfo-when-idle-after ..... 44
  - idlwave-keyword-class-inheritance ..... 21
  - idlwave-keyword-completion-adds-equal .... 19
  - idlwave-library-path ..... 45
  - idlwave-load-hook ..... 28
  - idlwave-main-block-indent ..... 9
  - idlwave-max-extra-continuation-indent .... 10
  - idlwave-max-popup-menu-items ..... 17
  - idlwave-mode-hook ..... 28
  - idlwave-no-change-comment ..... 11
  - idlwave-pad-keyword ..... 26
  - idlwave-query-class ..... 20
  - idlwave-query-shell-for-routine-info ..... 44
  - idlwave-reindent-end ..... 25
  - idlwave-reserved-word-upcase ..... 26
  - idlwave-resize-routine-help-window ..... 14
  - idlwave-rinfo-max-source-lines ..... 14
  - idlwave-scan-all-buffers-for-routine-info ..... 44
  - idlwave-shell-activate-prefix-keybindings ..... 33
  - idlwave-shell-arrows-do-history ..... 31
  - idlwave-shell-automatic-electric-debug ... 37
  - idlwave-shell-automatic-start ..... 29
  - idlwave-shell-breakpoint-face ..... 35
  - idlwave-shell-breakpoint-popup-menu ..... 35
  - idlwave-shell-comint-settings ..... 31
  - idlwave-shell-command-history-file ..... 30
  - idlwave-shell-command-line-options ..... 29
  - idlwave-shell-debug-modifiers ..... 33
  - idlwave-shell-electric-stop-color ..... 37
  - idlwave-shell-electric-stop-line-face .... 37
  - idlwave-shell-electric-zap-to-file ..... 37
  - idlwave-shell-examine-alist ..... 40
  - idlwave-shell-explicit-file-name ..... 29
  - idlwave-shell-expression-face ..... 39
  - idlwave-shell-file-name-chars ..... 31
  - idlwave-shell-frame-parameters ..... 30
  - idlwave-shell-graphics-window-size ..... 31
  - idlwave-shell-initial-commands ..... 30
  - idlwave-shell-input-mode-spells ..... 31
  - idlwave-shell-mark-breakpoints ..... 35
  - idlwave-shell-mark-stop-line ..... 35
  - idlwave-shell-max-print-length ..... 39
  - idlwave-shell-mode-hook ..... 30
  - idlwave-shell-output-face ..... 39
  - idlwave-shell-overlay-arrow ..... 35
  - idlwave-shell-prefix-key ..... 33
  - idlwave-shell-process-name ..... 29
  - idlwave-shell-prompt-pattern ..... 29
  - idlwave-shell-raise-frame ..... 30
  - idlwave-shell-save-command-history ..... 30
  - idlwave-shell-separate-examine-output .... 39
  - idlwave-shell-show-commands ..... 32
  - idlwave-shell-stop-line-face ..... 35
  - idlwave-shell-temp-pro-prefix ..... 30
  - idlwave-shell-use-dedicated-frame ..... 30
  - idlwave-shell-use-dedicated-window ..... 30
  - idlwave-shell-use-input-mode-magic ..... 31
  - idlwave-shell-use-toolbar ..... 32
  - idlwave-show-block ..... 25
  - idlwave-special-lib-alist ..... 14, 47
  - idlwave-split-line-string ..... 11
  - idlwave-startup-message ..... 28
  - idlwave-store-inquired-class ..... 20
  - idlwave-support-inheritance ..... 21
  - idlwave-surround-by-blank ..... 26
  - idlwave-system-directory ..... 45
  - idlwave-timestamp-hook ..... 27
  - idlwave-use-last-hang-indent ..... 12
  - idlwave-use-library-catalogs ..... 46
  - idlwave\_catalog ..... 45
  - 'Imenu', Emacs package ..... 27
  - Indentation ..... 9
  - Indentation, continued statement ..... 9
  - Indentation, of foreign code ..... 9
  - Inheritance, class ..... 20
  - Inheritance, keyword ..... 20
  - Input mode ..... 31
  - Inserting keywords, from routine info ..... 14
  - Installation ..... 41
  - Installing online help ..... 14, 41
  - Interactive Data Language ..... 1
  - Interface Definition Language ..... 1
  - Interview, with the maintainer ..... 50
  - Introduction ..... 1
- ## K
- Key bindings ..... 33
  - Keybindings for debugging ..... 32
  - Keyword completion ..... 18
  - Keyword inheritance ..... 20
  - Keywords of a routine ..... 13
  - Killing autoloaded buffers ..... 22
- ## L
- Library catalogs ..... 45
  - Line input mode (Shell) ..... 31
  - Line splitting ..... 11
  - LINKIMAGE, IDL routine ..... 43
  - Load-path shadows ..... 13, 47
- ## M
- M-? ..... 15
  - M-q ..... 11
  - M-(RET) ..... 11
  - M-(TAB) ..... 18
  - MacOS ..... 29, 46, 47, 53

MacOSX ..... 53  
 Magic spells, for input mode ..... 31  
 Maintainer, of IDLWAVE ..... 42  
 Major mode, `idlwave-mode` ..... 9  
 Major mode, `idlwave-shell-mode` ..... 29  
 Method completion ..... 18  
 Method Completion in Shell ..... 20  
 Mixed case completion ..... 19  
 Modification timestamp ..... 27  
 Module source file ..... 22  
 Motion commands ..... 27  
 Mouse binding to print expressions ..... 38  
 Multiply defined routines ..... 14, 47

## N

Nutshell, IDLWAVE in a ..... 3

## O

OBJ\_NEW, special online help ..... 15  
 Object method completion ..... 18  
 Object methods ..... 19  
 Online Help ..... 14  
 Online Help from the routine info buffer ..... 14  
 Online Help in ‘\*Completions\*’ buffer ..... 18  
 Online Help, in the shell ..... 31  
 Online Help, Installation ..... 14, 41  
 Operators, padding with spaces ..... 25

## P

Padding operators with spaces ..... 25  
 Paragraphs, filling ..... 10  
 Paragraphs, hanging ..... 10  
 Perl program, to create ‘`idlw-rinfo.el`’ ..... 48  
 PRINT expressions ..... 38  
 Printing expressions ..... 38  
 Printing expressions, on calling stack ..... 38  
 Procedure definitions, jumping to ..... 27  
 Procedure name completion ..... 18  
 Program structure, moving through ..... 27  
 Programs, compiling ..... 35

## Q

Quick-Start ..... 4

## R

RESOLVE\_ROUTINE ..... 22  
 Restrictions for expression printing ..... 38  
 Routine definitions ..... 43  
 Routine definitions, multiple ..... 14, 47  
 Routine info ..... 12  
 Routine info sources ..... 43  
 Routine info, in the shell ..... 31  
 Routine source file ..... 22

Routine source information ..... 13  
 ROUTINE\_NAMES, IDL procedure ..... 39  
 Routines, resolving ..... 22

## S

Saving object class on `->` ..... 20  
 Scanning buffers for routine info ..... 12, 43  
 Scanning the documentation ..... 48  
 Scrolling the ‘\*Completions\*’ window ..... 18  
`self` object, default class ..... 19  
 Shadows, load-path ..... 13, 47  
 Shell, basic commands ..... 30  
 Shell, querying for routine info ..... 12, 43  
 Shell, starting ..... 29  
 Showing commands in shell ..... 32  
 Source code, as online help ..... 17  
 Source file, access from routine info ..... 14  
 Source file, of a routine ..... 22  
 Sources of routine information ..... 43  
 Space, around operators ..... 25  
 Speed, of online help ..... 14  
 ‘Speedbar’, Emacs package ..... 27  
 Spells, magic ..... 31  
 Splitting, of lines ..... 11  
 Starting the shell ..... 29  
 Stepping ..... 33  
 String splitting ..... 11  
 Structure tag completion ..... 21  
 Structure tags, in online help ..... 17  
 Subprocess of Emacs, IDL ..... 29  
 Summary of important commands ..... 3  
 Syntax highlighting ..... 12  
 Syntax highlighting, Octals ..... 12

## T

Templates ..... 22  
 Thanks ..... 42  
 Timestamp, in doc header ..... 27  
 Toolbar ..... 32  
 Troubleshooting ..... 54  
 Tutorial ..... 4

## U

Uppcase, enforcing for reserved words ..... 26  
 Updating routine info ..... 12, 43  
 URL, homepage for IDLWAVE ..... 41  
 User catalog ..... 46

## W

Windows ..... 29, 46, 47, 53

## X

XEmacs package IDLWAVE ..... 41  
 XML Help Catalog ..... 14