

IDLWAVE User Manual

Major Emacs mode and shell for IDL and WAVE/CL files
Edition 4.7, December 2000

by **Carsten Dominik**

This is edition 4.7 of the *IDLWAVE User Manual* for IDLWAVE version 4.7, December 2000.

Copyright © 1999, 2000, 2001 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 1.1 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

1	Introduction	1
2	IDLWAVE in a Nutshell	2
3	Getting Started (Tutorial)	3
3.1	Lesson I: Development Cycle	3
3.2	Lesson II: Customization	5
3.3	Lesson III: Library Catalog	7
4	The IDLWAVE Major Mode	8
4.1	Code Formatting	8
4.1.1	Code Indentation	8
4.1.2	Comment Indentation	8
4.1.3	Continuation Lines and Filling	9
4.1.4	Syntax Highlighting	10
4.2	Routine Info	10
4.3	Online Help	12
4.4	Completion	14
4.5	Routine Source	16
4.6	Resolving Routines	16
4.7	Code Templates	16
4.8	Actions	17
4.8.1	Block Boundary Check	17
4.8.2	Padding Operators	18
4.8.3	Case Changes	18
4.9	Documentation Header	19
4.10	Motion Commands	19
4.11	Miscellaneous Options	20
5	The IDLWAVE Shell	21
5.1	Starting the Shell	21
5.2	Using the Shell	22
5.3	Debugging IDL Programs	23
5.3.1	Compiling Programs	24
5.3.2	Breakpoints and Stepping	24
5.3.3	Examining Variables	25
6	Installation	27
6.1	Installing IDLWAVE	27
6.2	Installing Online Help	27
6.3	Upgrading from the old ‘idl.e1’ file	27

7 Acknowledgement	28
Appendix A Sources of Routine Info	29
A.1 Routine Definitions	29
A.2 Routine Information Sources	29
A.3 Library Catalog	30
A.4 Load-Path Shadows	31
A.5 Documentation Scan	31
Appendix B Configuration Examples	32
Index	34

1 Introduction

IDLWAVE is a package to support editing command files for the Interactive Data Language (IDL), and for running IDL as an inferior shell. It also can be used for WAVE/CL command files, but the support for these is limited. Note that this package has nothing to do with the Interface Definition Language as part of the Common Object Request Broker Architecture (CORBA).

IDLWAVE is the 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'`. If you have been using the old files, check Section 6.3 [Upgrading from `idl.el`], page 27 for information on how to switch.

IDLWAVE consists of two parts: A major mode for editing command files (`idlwave-mode`) and a mode to allow running the IDL program as an inferior shell (`idlwave-shell-mode`). Both modes work closely together and form a complete development environment.

Here is a brief summary of what IDLWAVE does.

- Code indentation and formatting.
- Font-lock support on three levels.
- Display of calling sequence and keywords of more than 1000 IDL routines and any routines in your local IDL library.
- Name space conflict search.
- Fast context-sensitive online help.
- Context sensitive completion of routine names and keywords.
- Insertion of code templates.
- Actions to enforce coding standards during typing.
- Block structure check.
- Support for `'imenu'` (Emacs) and `'func-menu'` (XEmacs).
- Documentation support.
- Running IDL as inferior process.
- Shell with history search, command line editing and completion.
- Compilation, execution and debugging of programs directly from the source buffer.
- Examining expressions with a mouse click.

In this manual, each section contains a list of user options related to the subject. 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 things. For a full description of what a particular variable does and how to configure it, see the documentation string of that variable. 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>M-C-\</code>	Re-indent all lines in the current region.
<code>M-(RET)</code>	Start a continuation line. Or split the current line at point.
<code>M-q</code>	Fill the current comment paragraph.
<code>C-c ?</code>	Display calling sequence, keywords of the procedure/function call at point.
<code>M-?</code>	Context sensitive online help.
<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>	Find the source code of a procedure/function.
<code>C-c C-h</code>	Insert a standard documentation header.
<code>C-c C-m</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 interaction buffer.
<code>M-p</code>	Cycle back through IDL command history matching command line input.
<code>M-n</code>	Cycle forward.
<code>M-(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>	Goto next syntax error.
<code>C-c C-d C-b</code>	Set a breakpoint at the current source line.
<code>C-c C-d C-d</code>	Clear the current breakpoint.
<code>C-c C-d C-p</code>	Ask IDL to print the value of the expression near point.

Commonly used Settings in '.emacs'

```

;; Change the indentation preferences
(setq idlwave-main-block-indent 2          ; default 0
      idlwave-block-indent 2             ; default 4
      idlwave-end-offset -2)            ; default -4

;; Pad some 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))

;; Where are the online help files?
(setq idlwave-help-directory "~/idlwave")

```

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 with IDLWAVE. We will type a simple program into a buffer and use the shell to compile, debug and run this program. On the way we will use the most important commands in IDLWAVE. Note however that there is much more functionality available in IDLWAVE than we cover here, and it will pay off greatly if eventually you go further and read the whole manual.

I assume that you have access to Emacs or XEmacs with the full IDLWAVE package including online help (see Chapter 6 [Installation], page 27). I 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. You can see this by looking at the mode line, just below the editing window. Also, the menu bar should contain entries ‘IDLWAVE’ and ‘Debug’.

Now cut-and-paste the following program, 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` followed by `M-C-\`). 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. Now place the cursor in any line you would like to split into two 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 weekday of a given date for the first 10 years of the 21st century. I have put in a few bugs which we are going to fix now.

First, let's launch the IDLWAVE shell. You do this with the command `C-c C-s`. The Emacs window will split and display IDL running in a shell interaction buffer. Type a few commands like `print, !PI` to convince yourself that you can work there like in an xterminal, 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 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 weekdays for 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), 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 was enough to tell 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, hit `(TAB)`, 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 pops up the routine info window for the `plot` routine, which contains a list of keywords, and 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-(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). The online help window will pop up showing the 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 the command to recall it by now: `C-c C-s`) and execute again. Now things look pretty good.

Lets 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. Lets 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 line is highlighted in some way. 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! So fix the line to

```
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 available, and it would be a shame if you did not make use of this and adapt IDLWAVE to your own preferences. Customizing Emacs or IDLWAVE means that you have to set Lisp variables in the `'.emacs'` file in your home directory. This looks scary to many people because of all the parenthesis. However, you can just cut and paste the examples given here and work from there.

Lets 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 `(SPC)` or `(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.

Now I bet you have your own indentation preferences for IDL code. For example, I like to indent the main block of an IDL program a bit, different from the conventions used by RSI. Also, I'd like to use only 3 spaces as indentation between `BEGIN` and `END`. Try the following lines in `'.emacs'`

```
(setq idlwave-main-block-indent 2)
(setq idlwave-block-indent 3)
(setq idlwave-end-offset -3)
```

Restart Emacs, take the program we developed in the first part of this tutorial and re-indent it with `C-c h` and `M-C-\`. You probably 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:’ in the manual.

If you cannot wrap your head around this Lisp stuff, 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, option `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 Main Block Indent`), 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.

Many people I talk to find the key bindings used for the debugging commands too long and complicated. Do I always have to type `C-c C-d C-c` to get a single simple command? Due to Emacs rules and conventions I cannot make better bindings by default, but you can. First, there is a way to assign all debugging commands in a single sweep to other combinations. The only problem is that we have to use something which Emacs does not need for other important commands. A 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. You can have this with

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

If you have a special keyboard with for example a `HYPHER` key, you could use

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

instead to get compilation on `H-c`.

You can also assign specific commands to function keys. This you must do in the `mode-hook`, a special function which is run when a new buffer gets set up. Keybindings can only be done when the buffer exists. The possibilities for key customization are endless. Here we set function keys `f5-f8` to common debugging commands.

```
;; First for the source buffer
(add-hook 'idlwave-mode-hook
  (lambda ()
    (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 [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: Library Catalog

We have already used the routine info display in the first part of this tutorial. This was the key `C-c ?` which displays information about the IDL routine near the cursor position. Wouldn't it be nice to have the same available for your own library routines and for the huge amount of code in major extension libraries like JHUPL or the IDL-Astro library? To do this, you must give IDLWAVE a chance to study these routines first. We call this *Building the library catalog*.

From the IDLWAVE entry in the menu bar, select **Routine Info/Select Catalog Directories**. If necessary, start the shell first with `C-c C-s` (see Section 5.1 [Starting the Shell], page 21). 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) 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 `.idlcat` in your home directory and will from now on be automatically loaded whenever you use IDLWAVE. Try to use routine info (`C-c ?`) or completion (`M-<TAB>`) while 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 `comp.lang.idl-pvwave`.

4 The IDLWAVE Major Mode

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

4.1 Code Formatting

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.

To re-indent a larger portion of code (e.g. when working with foreign code written with different conventions), use `M-C-\` (`indent-region`) after marking the relevant code. Useful marking commands are `C-x h` (the entire file) or `M-C-h` (the current subprogram). See Section 4.8 [Actions], page 17, for information how to impose additional formatting conventions on foreign code.

idlwave-main-block-indent (0) 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 (5) User Option

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

idlwave-end-offset (-4) 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.

idlwave-continuation-indent (2) User Option

Extra indentation applied to continuation lines and inside unbalanced parenthesis.

4.1.2 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 indent 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.

4.1.3 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 a '\$' to indicate that the following line is a continuation of the current line, terminates the line with a newline and indents the new line. The command `M-RET` can also be used in the middle of a line to split the line at that point. When used inside a long string constant, the string is split with the '+' concatenation operator.

When filling comment paragraphs, IDLWAVE overloads the normal filling functions and uses a function which creates hanging paragraphs as they are customary in the 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 is matched by `idlwave-hang-indent-regexp`, subsequent lines are indented to after the position of this match, as in the following example.

```

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

You also refill a comment paragraph with `M-q`.

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 '+'.

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`.

4.1.4 Syntax Highlighting

Highlighting of keywords, comments, strings etc. can be accomplished with `font-lock`. If you are using `global-font-lock-mode` (on Emacs), or have `font-lock` turned on in any other buffer in XEmacs, it should also automatically work in IDLWAVE buffers. If not, you can enforce it with the following line in your `.emacs`

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

IDLWAVE supports 3 levels of syntax highlighting. The variable `font-lock-maximum-decoration` determines which level is selected.

idlwave-default-font-lock-items

User Option

Items which should be fontified on the default fontification level 2.

4.2 Routine Info

IDL defines more than one thousand procedures, functions and object methods. This large command set makes it difficult to remember the calling sequence and keywords of a command. IDLWAVE contains a list of all builtin routines with calling sequences and keywords¹. It also scans Emacs buffers and library files for routine definitions and queries the IDLWAVE-Shell for the properties of modules currently compiled under the shell. This information is updated automatically. If you think the information is not up-to-date, use `C-c C-i` (`idlwave-update-routine-info`) to enforce a global update.

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 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 select 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.

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 [CSB] /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 library routine, which may or may not be what you want (see Section A.4 [Load-Path Shadows], page 31). The information about the calling sequence and the keywords is derived from the first source listed. Library routines can only be supported if

¹ This list was created by scanning the IDL manual and might contain (very few) errors. Please report any detected errors to the maintainer, so that they can be fixed.

you have scanned the local IDL library (see Section A.3 [Library Catalog], page 30). The source entry consists of a *source category*, a set of *flags* and the path to the *source file*. The following categories exist:

<i>System</i>	A system routine, but we do not know if it is <i>Builtin</i> or <i>SystemLib</i> . When the system library has been scanned (see Section A.3 [Library Catalog], page 30), this category will automatically split into the next two.
<i>Builtin</i>	A builtin routine with no source code available.
<i>SystemLib</i>	A library routine in the official lib directory ‘!DIR/lib’.
<i>Obsolete</i>	A library routine in the official lib directory ‘!DIR/lib/obsolete’.
<i>Library</i>	A file on IDL’s search path !PATH.
<i>Other</i>	Any other file not known to be on the search path.
<i>Unresolved</i>	The shell lists this routine as unresolved.

You can define additional categories based on the file name and path with the variable `idlwave-special-lib-alist`.

The flags [CSB] indicate if the file is known to IDLWAVE from the library catalog ([C--], see Section A.3 [Library Catalog], page 30), from the Shell ([-S-]) or from an Emacs buffer ([--B]). Combinations are possible. 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*’ buffer will be active (it highlights when you move the mouse over it). Clicking on these items will have the following effects:

<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 Section 4.3 [Online Help], page 12).
<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 ‘/’.
<i>Source</i>	Clicking with the <i>middle</i> mouse button on a ‘Source’ 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.

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, RSI provides extensive documentation. IDLWAVE can access an ASCII version of this documentation very quickly and accurately. This is *much* faster than using the IDL online help application, also because usually IDLWAVE gets you to the right place in the docs directly, without additional browsing and scrolling. For this online help to work, you need an ASCII version of the IDL documentation which is not part of the standard IDLWAVE distribution. The required files can be downloaded from the maintainers webpage (<http://idlwave.org/>). As the text is extracted from PDF files, the formatting of the help text is good for normal text, but graphics and multiline equations will not be well represented. See also Section A.5 [Documentation Scan], page 31.

For routines which are not documented in the IDL manual (for example your own routines), the source code 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. Otherwise the routine definition statement (`pro/function`) is shown.

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

<i>Routine name</i>	The name of a routine (function, procedure, method).
<i>Keyword Parameter</i>	Keyword parameter of a routine.
<i>System Variable</i>	System variable like <code>!DPI</code> .
<i>IDL Statement</i>	Like <code>PRO</code> , <code>REPEAT</code> , or <code>COMPILE_OPT</code> .
<i>Class name</i>	Class name in <code>OBJ_NEW</code> call.
<i>Executive Command</i>	Executive command like <code>.RUN</code> . Mostly useful in the shell.
<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 source buffers, 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 10).
- When using completion and Emacs pops up a window with possible completions, clicking with *Mouse-3* on a completion item invokes help on that item (see Section 4.4 [Completion], page 14).

In both cases, a blue face indicates that the item is documented in the IDL manual.

The help window is normally displayed in a separate frame. The following commands can be used to navigate inside the help system.

<code>(SPACE)</code>	Scroll forward one page.
<code>(RET)</code>	Scroll forward one line.
<code>(DEL)</code>	Scroll back one page.
<code>n, p</code>	Browse to the next or previous topic (in physical sequence).
<code>b, f</code>	Move back and forward through the help topic history.
<code>c</code>	Clear the history.
<code>Mouse-2</code>	Follow a link. Active links are displayed in a different font. Items under <i>See Also</i> are active, and classes have links to their methods and back.
<code>o</code>	Open a topic. The topic can be selected with completion.
<code>*</code>	Load the whole help file into Emacs, for global text searches.
<code>q</code>	Kill the help window.

When the help text is a source file, the following commands are also available.

<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> .

idlwave-help-directory User Option

The directory where `idlw-help.txt` and `idlw-help.el` are stored.

idlwave-help-use-dedicated-frame (`t`) 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-link-face User Option

The face for links to IDLWAVE online help.

idlwave-help-activate-links-agressively (`t`) User Option

Non-`nil` means, make all possible links in help window active.

4.4 Completion

IDLWAVE offers completion for class names, routine names and keywords. As in many programming modes, completion is bound to $M-\overline{\text{TAB}}$. Completion uses the same internal information as routine info, so when necessary 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 at 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 -> a*</code>	Object method (procedure)
<code>a(2,3) = obj -> 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

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 displayed in a different font. For other items, the corresponding source code or DocLib header is available as help text.

The case of the completed words is determined by what is already in the buffer. 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. 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. These defaults can be changed with the variable `idlwave-completion-case`.

idlwave-completion-case User Option
 Association list setting the case (UPPER/lower/Capitalized/...) 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), lower case strings will be completed to lower case.

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

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 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.

Object Method Completion and Class Ambiguity

An object method is not uniquely determined without the object’s class. Since the class part is usually omitted in the source code, IDLWAVE considers all available methods in all classes as possible completions of an object method name. For keywords, the combined keywords of the current method in all available classes will be considered. In the ‘*Completions*’ buffer, the classes allowed for each completion will be shown next to the item (see option `idlwave-completion-show-classes`). As a special case, the class of an object called ‘self’ object is always the class of the current routine. All classes it inherits from are considered as well where appropriate.

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 this behavior the default (not recommended). 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. Remembering the class works by placing a text property in the object operator ‘->’. This is not enabled by default - the variable `idlwave-store-inquired-class` can be used to turn it on.

idlwave-support-inheritance (t) User Option
 Non-`nil` means, treat inheritance with completion, online help etc.

idlwave-completion-show-classes (1) User Option
 Non-`nil` means, show 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 (nil) 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 class text property.

4.5 Routine Source

Apart from clicking on a *Source:* line in the routine info window, there is also another way to find 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. In the minibuffer, specify a complete routine name (including the class part). IDLWAVE will display the source file in another window.

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 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.

`idlwave-resolve` is one way to get a library module within reach of IDLWAVE's routine info collecting functions. A better way is to scan (parts of) the library (see Section A.3 [Library Catalog], page 30). Routine info on library 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 29, for in-depth information where 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

Otherwise, special abbreviations are used. 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). Here are a few examples of predefined abbreviations. For a full list, use `M-x idlwave-list-abbrevs`.

<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
<code>\b</code>	BEGIN

The templates are expanded in upper or lower case, depending upon the variables `idlwave-abbrev-change-case` and `idlwave-reserved-word-upcase`.

idlwave-abbrev-start-char ("`\`") User Option
 A single character string used to start abbreviations in abbrev mode.

idlwave-abbrev-move (`t`) User Option
 Non-`nil` means the abbrev hook can move point, e.g. to end up between the parenthesis of a function call.

4.8 Actions

Actions are special 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, they can be helpful.

Action 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`.
- Action can also be applied to a larger piece of code, e.g. in order 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 `M-C-h` (the current subprogram). See Section 4.1.1 [Code Indentation], page 8, for information how to adjust the indentation of the code.

idlwave-do-actions (`nil`) User Option
 Non-`nil` means performs actions when indenting.

4.8.1 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 kind of `END` 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 a specific 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.

4.8.2 Padding Operators

Some operators can be automatically surrounded by spaces. This can happen when the operator is typed, or also later when the line is indented. IDLWAVE contains this setting for the operators '&', '<', '>', ',', '=', and '->',¹ but the feature is turned off by default. If you want to turn it on, customize the variables `idlwave-surround-by-blank` and `idlwave-do-actions`. 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, 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))))
```

idlwave-surround-by-blank (nil)

User Option

Non-nil means, enable `idlwave-surround`. If non-nil, '=', '<', '>', '&', ',', '->' are surrounded with spaces by `idlwave-surround`.

idlwave-pad-keyword (t)

User Option

Non-nil means pad '=' for keywords like assignments.

4.8.3 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`.

idlwave-abbrev-change-case (nil)

User Option

Non-nil means all abbrevs will be forced to either upper or lower case. Legal values are nil, t, and down.

¹ Operators longer than one character can only be padded during line indentation.

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

4.9 Documentation Header

The command *C-c C-h* inserts a standard routine header into the buffer, with the usual fields for documentation. 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-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.

4.10 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.

Several commands allow to move quickly through the structure of an IDL program. These are

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

4.11 Miscellaneous Options

idlwave-help-application

User Option

The external application providing reference help for programming.

idlwave-startup-message (τ)

User Option

Non-`nil` means display a startup message when `idlwave-mode`' is first called.

idlwave-mode-hook

User Option

Normal hook. Executed when a buffer is put into `idlwave-mode`.

idlwave-load-hook

User Option

Normal hook. Executed when `'idlwave.el'` is loaded.

5 The IDLWAVE Shell

The IDLWAVE shell is an Emacs major mode which allows to run the IDL program as an inferior process of Emacs. 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 uses ‘comint’, an Emacs packages which handles the communication with the IDL program. Unfortunately IDL for Windows and MacOS does not allow the interaction with Emacs¹, so the IDLWAVE shell only works under GNU and Unix.

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 simple switch to the shell buffer. The command *C-c C-l* (*idlwave-shell-recenter-shell-window*) displays the shell window without selecting it.

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*.

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*.

idlwave-shell-explicit-file-name	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
Regexp 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	User Option
Non- <i>nil</i> means attempt to invoke <i>idlwave-shell</i> if not already running.	
idlwave-shell-initial-commands	User Option
Initial commands, separated by newlines, to send to IDL.	
idlwave-shell-use-dedicated-frame (<i>nil</i>)	User Option
Non- <i>nil</i> means, IDLWAVE should use a special frame to display shell buffer.	

¹ Please inform the maintainer if you come up with a way to make the IDLWAVE shell work on these systems.

idlwave-shell-frame-parameters	User Option
The frame parameters for a dedicated <code>idlwave-shell</code> frame.	
idlwave-shell-temp-pro-prefix	User Option
The prefix for temporary IDL files used when compiling regions.	
idlwave-shell-mode-hook	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 terminal². Here is a list of commonly used commands.

<code>(UP)</code>	Cycle backwards in input history
<code>(DOWN)</code>	Cycle forwards in input history
<code>M-p</code>	Cycle backwards in input history <i>matching input</i>
<code>M-n</code>	Cycle forwards in input history <i>matching input</i>
<code>M-r</code>	Previous input matching a regexp
<code>M-s</code>	Next input that matches 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>
<code>C-c C-\</code>	Send <code>^\</code>
<code>C-c C-o</code>	Delete last batch of process output
<code>C-c C-r</code>	Show last batch of process output
<code>C-c C-l</code>	List input history

In addition to these standard ‘comint’ commands, `idlwave-shell-mode` provides many of the commands which simplify writing IDL code, including abbreviations, online help, and completion. See Section 4.2 [Routine Info], page 10 and Section 4.3 [Online Help], page 12 and Section 4.4 [Completion], page 14 for more information on these commands.

<code>(TAB)</code>	Completion of file names, routine names and keywords (<code>idlwave-shell-complete</code>)
<code>M-(TAB)</code>	Same as <code>(TAB)</code>
<code>C-c ?</code>	Routine Info display (<code>idlwave-routine-info</code>)
<code>M-?</code>	IDL online help on routine (<code>idlwave-routine-info-from-idlhelp</code>)
<code>C-c C-i</code>	Update routine info from buffers and shell (<code>idlwave-update-routine-info</code>)
<code>C-c C-v</code>	Find the source file of a routine (<code>idlwave-find-module</code>)
<code>C-c =</code>	Compile a library routine (<code>idlwave-resolve</code>)

² 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`.

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

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 Debugging IDL Programs

Programs can be compiled, run, and debugged directly from the source buffer in Emacs. The IDLWAVE shell installs key bindings both in the shell buffer and in all IDL code buffers of the current Emacs session. On Emacs versions which support this, it also installs a debugging toolbar. The display of the toolbar can be toggled with `C-c C-d C-t` (`idlwave-shell-toggle-toolbar`).

The debugging key bindings are 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`, compiling a source file with `C-c C-d C-c`. If you find this too much work you can choose a combination of modifier keys which is not used by other commands. For example, if you write in `.emacs`

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

a breakpoint can 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 single-keypress shortcut with the modifiers given in the `idlwave-shell-debug-modifiers` variable.

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 binding debugging commands in the shell and in source buffers.

idlwave-shell-use-toolbar (t) User Option
 Non-nil means, use the debugging toolbar in all IDL related buffers.

5.3.1 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 send 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 send 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*.

idlwave-shell-mark-stop-line (t) User Option
 Non-nil means, mark the source code line where IDL is currently stopped. The value decides about the preferred method. Legal values are `nil`, `t`, `arrow`, and `face`.

idlwave-shell-overlay-arrow (">") User Option
 The overlay arrow to display at source lines where execution halts.

idlwave-shell-stop-line-face User Option
 The face which highlights the source line where IDL is stopped.

5.3.2 Breakpoints and Stepping

You can set breakpoints and step through a program with IDLWAVE. Setting a breakpoint in the current line of the source buffer is done with *C-c C-d C-b* (*idlwave-shell-break-here*). With a prefix arg of 1, the breakpoint gets a `/ONCE` keyword, meaning that it will be deleted after first use. With a numeric prefix greater than one, the breakpoint will only be active the *n*th time it is hit. To clear the breakpoint in the current line, use *C-c C-d C-d* (*idlwave-clear-current-bp*). To clear all breakpoints, use *C-c C-d C-a* (*idlwave-clear-all-bp*). Breakpoint lines are highlighted in the source code.

Once the program has stopped somewhere, you can step through it. Here is a summary of the breakpoint and stepping commands:

<i>C-c C-d C-b</i>	Set breakpoint (<i>idlwave-shell-break-here</i>)
<i>C-c C-d C-i</i>	Set breakpoint in function named here (<i>idlwave-shell-break-in</i>)

<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 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 (<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>)

idlwave-shell-mark-breakpoints (t) User Option

Non-nil means, mark breakpoints in the source file buffers. The value indicates the preferred method. Legal 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`.

5.3.3 Examining Variables

When execution is stopped you can examine the values of variables. The command `C-c C-d C-p` prints the expression at point, while `C-c C-d ?` shows help on this expression. The expression at point is an array expression or a function call, or the contents of a pair of parenthesis. The selected expression becomes highlighted in the source code for a short time. Calling the above commands with a prefix argument will prompt for an expression instead of using the one at point.

It is very convenient to click with the mouse on expressions to retrieve their value. Expression printing is also bound to `S-Mouse-2` and expression help to `C-S-Mouse-2`. I.e. you need to hold down `(SHIFT)` and `(CONTROL)` while clicking with the mouse.

Printing of expressions also works on higher levels of the calling stack. This means that you can examine the values of variables and expressions inside the routine which called the current routine etc. 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 through the calling stack. The mode line of the shell window will indicate the routine and the calling stack level which define the context for printing expressions. 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.
- 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.

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-print-expression-function (nil)

User Option

A function to handle special display of evaluated expressions.

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 (<ftp://ftp.xemacs.org/pub/xemacs/packages/>) with the normal package management system on XEmacs 21. These pre-installed versions should work out-of-the-box. However, the files needed for online help are not distributed with XEmacs/Emacs and have to be installed separately¹ (see Section 6.2 [Installing Online Help], page 27).

You can also download IDLWAVE and install it yourself from the maintainers webpage (<http://idlwave.org/>). Follow the instructions in the INSTALL file.

6.2 Installing Online Help

If you want to use the online help display, two additional files (an ASCII version of the IDL documentation and a topics/code file) must be installed. These files can also be downloaded from the maintainers webpage (<http://idlwave.org/>). You need to place the files somewhere on your system and tell IDLWAVE where they are with

```
(setq idlwave-help-directory "/path/to/help/files/")
```

6.3 Upgrading from the old ‘idl.el’ file

If you have been using the old ‘idl.el’ and ‘idl-shell.el’ files and would like to use IDLWAVE, you need to update your customization in ‘.emacs’.

1. Change all variable and function prefixes from ‘idl-’ to ‘idlwave-’.
2. Remove the now invalid `autoload` and `auto-mode-alist` forms pointing to the ‘idl.el’ and ‘idl-shell.el’ files. Install the new `autoload` forms.
3. If you have been using the hook function recommended in earlier versions to get a separate frame for the IDL shell, remove that command from your `idlwave-shell-mode-hook`. Instead, set the variable `idlwave-shell-use-dedicated-frame` with

```
(setq idlwave-shell-use-dedicated-frame t)
```

4. The key sequence `M-TAB` no longer inserts a TAB character. Like in many other Emacs modes, `M-TAB` now does completion. Inserting a TAB has therefore been moved to `C-TAB`. On a character based terminal you can also use `C-c SPC`.

¹ Due to copyright reasons, the ASCII version of the IDL manual cannot be distributed under the GPL.

7 Acknowledgement

The main contributors to the IDLWAVE package have been:

- **Chris Chase** (<mailto:chase@att.com>), the original author. Chris wrote ‘idl.el’ and ‘idl-shell.el’ and maintained them for several years.
- **Carsten Dominik** (<mailto:dominik@astro.uva.nl>), who have been in charge of the package since version 3.0, and also wrote this manual.
- **John-David Smith** (<mailto:jdsmith@astro.cornell.edu>), current maintainer, who is also responsible for this manual’s maintenance. John-David has also shaped Object method completion and most new features in version 4.0 with his ideas, bug reports, and patient explanations of IDL internals.

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

- Ulrik Dickow <dickow@nbi.dk>
- Eric E. Dors <edors@lanl.gov>
- Stein Vidar H. Haugan <s.v.h.haugan@astro.uio.no>
- David Huenemoerder <dph@space.mit.edu>
- Kevin Ivory <Kevin.Ivory@linmpi.mpg.de>
- Xuyong Liu <liu@stsci.edu>
- Simon Marshall <Simon.Marshall@esrin.esa.it>
- Craig Markwardt <craigm@cow.physics.wisc.edu>
- Laurent Mugnier <mugnier@onera.fr>
- Lubos Pochman <lubos@rsinc.com>
- Patrick M. Ryan <pat@jaameri.gsfc.nasa.gov>
- Marty Ryba <ryba@ll.mit.edu>
- Phil Williams <williams@irc.chmcc.org>
- Phil Sterne <sterne@dublin.llnl.gov>

Thanks to everyone!

Appendix A Sources of Routine Info

In Section 4.2 [Routine Info], page 10 and Section 4.4 [Completion], page 14 it was shown how IDLWAVE displays the calling sequence and keywords of routines, and how it 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 accessible to the user.
2. Routines *part of the current program* are defined in a file which is explicitly compiled by the user. This file may or may not be located on the IDL search path.
3. *Library routines* are defined in special files which are located somewhere 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.
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 external routines.

A.2 Routine Information Sources

In order to know about as many routines as possible, IDLWAVE will do the following to collect information:

1. It has a *builtin list* with the properties of the builtin IDL routines. IDLWAVE 4.7 is distributed with a list of 1287 routines and 5724 keywords, reflecting IDL version 5.4. This list has been created by scanning the IDL manuals and is stored in the file `'idlw-rinfo.el'`. See Section A.5 [Documentation Scan], page 31, for information how to regenerate this file for new versions of IDL.
2. It *scans* all *buffers* of 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 which is 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 as inferior process of the current 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, and each time an Emacs buffer is compiled with `C-c C-d C-c`. The command `C-c C-i (idlwave-update-routine-info)` can be used to ask the shell again at any time.
4. IDLWAVE can scan all or selected library files and store the result in a file which will be automatically loaded just like `'idlw-rinfo.el'`. See Section A.3 [Library Catalog], page 30, for information how to scan library files.

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 Library Catalog

IDLWAVE can extract routine information from library modules and store that information in a file. To do this, the variable `idlwave-libinfo-file` needs to contain the path to a file in an existing directory (the default is `"~/idlcat.el"`). Since the file will contain lisp code, its name should end in `‘.el’`. Under Windows and MacOS, you also need to specify the search path for IDL library files 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`. Under Unix and GNU, these values will be automatically inferred from an IDLWAVE shell.

The command `M-x idlwave-create-libinfo-file` can then be used to scan library files. It brings up a widget in which you can select some or all directories on the search path. If you only want to have routine and completion info of some libraries, it is sufficient to scan those directories. However, if you want IDLWAVE to detect possible name conflicts with routines defined in other libraries, the whole pass should be scanned.

After selecting directories, click on the `‘[Scan & Save]’` button in the widget to scan all files in the selected directories and write the resulting routine information into the file `idlwave-libinfo-file`. In order to update the library information from the same directories, 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 libinfo file and rebuild IDLWAVEs internal lists.

A note of caution: Depending on your local installation, the IDL library can be very large. Parsing it for routine information will take time and loading this information into Emacs can require a significant amount of memory. However, having this information available will be a great help.

idlwave-libinfo-file	User Option
File for routine information of the IDL library.	
idlwave-library-path	User Option
IDL library path for Windows and MacOS. Not needed under GNU and Unix.	
idlwave-system-directory	User Option
The IDL system directory for Windows and MacOS. Not needed under GNU and Unix.	
idlwave-special-lib-alist	User Option
Alist of regular expressions matching special library directories.	

A.4 Load-Path Shadows

IDLWAVE can compile a list of routines which are defined in several different files. 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 10). The different definitions of a routine are listed in the sequence of *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:

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 properly you should have scanned the entire load path, not just selected directories. 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 of course also advisable for many other reasons.

Users of Windows and MacOS 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.

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 10).

A.5 Documentation Scan

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. The Online Help feature of IDLWAVE requires ASCII versions of some IDL manuals to be available in a specific format (`'idlw-help.txt'`), along with an Emacs-Lisp file `'idlw-help.el'` with supporting code and pointers to the ASCII file.

All 3 files can be derived from the IDL documentation. 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_rinfo'` which constructs these files by scanning selected files from the IDL documentation. Instructions on how to use `'get_rinfo'` are in the program itself.

Appendix B Configuration Examples

Question: So now 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: Hardly any. As the maintainer, I set the default of most options to what I think is 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 out there
- break with widely used standards.

To see what I mean, here is the *entire* configuration I have in my `‘.emacs’`:

```
(setq idlwave-shell-debug-modifiers '(control shift)
      idlwave-store-inquired-class t
      idlwave-shell-automatic-start t
      idlwave-main-block-indent 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, the options allow you to change almost every aspect of it. Here is an example of a much more extensive configuration of IDLWAVE. To say it again - this is not what I recommend, but 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

;;; Some setting can only be done from a mode hook. Here is an example:

(add-hook 'idlwave-mode-hook
  (lambda ()
    (setq abbrev-mode 1) ; Turn on abbrevs (-1 for off)
```

```

(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
;;
;; 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)
;; 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.
))

;;; 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

```

Index

!

!DIR, IDL variable 10, 30, 31
 !PATH, IDL variable 10, 29

.

‘.emacs’ 32

A

Abbreviations 16
 Acknowledgement 28
 Actions 17
 Actions, applied to foreign code 17
 Active text, in routine info 11
 Application, testing for shadowing 31
 Authors, of IDLWAVE 28
 auto-fill-mode 9

B

Block boundary check 17
 Block, closing 17
 Breakpoints 24
 Buffer, testing for shadowing 31
 Buffers, killing 16
 Buffers, scanning for routine info 10, 29
 Builtin list of routines 29

C

C-c ? 10
 C-c C-d 23
 C-c C-d C-b 24
 C-c C-d C-c 24
 C-c C-d C-p 25
 C-c C-h 19
 C-c C-i 10, 14
 C-c C-m 19
 C-c C-s 21
 C-c C-v 16
 CALL_EXTERNAL, IDL routine 29
 Calling sequences 10
 Calling stack, motion 25
 Case changes 18
 Case of completed words 14
 Categories, of routines 10
 cc-mode.el 1
 Changelog, in doc header 19
 Character input mode (Shell) 23
 Class ambiguity 15
 Class name completion 14

Class query, forcing 15
 Closing a block 17
 Code formatting 8
 Code indentation 8
 Code structure, moving through 19
 Code templates 16
 Coding standards, enforcing 17
 Comint 22
 Comint, Emacs package 21
 Comment indentation 8
 Compiling library modules 16
 Compiling programs 24
 Completion 14
 Completion, in the shell 22
 Completion, Online Help 14
 Completion, scrolling 14
 Configuration examples 32
 Context, for online help 12
 Continuation lines 9
 Contributors, to IDLWAVE 28
 Copyright, of IDL manual 27
 Copyright, of IDLWAVE 2
 CORBA (Common Object Request Broker
 Architecture) 1

D

Debugging 23
 Dedicated frame, for shell buffer 21
 Default command line, executing 24
 Default routine, for info and help 10
 Default settings, of options 32
 DocLib header 19
 DocLib header, as online help 12
 Documentation header 19
 Downcase, enforcing for reserved words 18
 Duplicate routines 11, 31

E

Emacs, distributed with IDLWAVE 27
 Email address, of Maintainer 28
 END type checking 17
 END, automatic insertion 17
 END, expanding 17
 Example configuration 32
 Executing a default command line 24
 Execution, controlled 24
 Expressions, help 25
 Expressions, printing 25
 External routines 29

F

Feature overview.....	1
Filling.....	9
Flags, in routine info.....	11
Font lock.....	10
Forcing class query.....	15
Foreign code, adapting.....	8, 17
Formatting, of code.....	8
Frame, for shell buffer.....	21
FTP site.....	27
'Func-menu', XEmacs package.....	19
Function definitions, jumping to.....	19
Function name completion.....	14

G

'get_rinfo'.....	31
Getting Started.....	3

H

Hanging paragraphs.....	8, 9
Header, for file documentation.....	19
Help application, key bindings.....	13
HELP, on expressions.....	25
Highlighting of syntax.....	10
Homepage for IDLWAVE.....	27
Hooks.....	20, 22

I

IDL library routine info.....	30
IDL manual, ASCII version.....	12
IDL variable !DIR.....	10, 30, 31
IDL variable !PATH.....	10, 29
IDL, as Emacs subprocess.....	21
'idl-shell.el'.....	1
'idl.el'.....	1
'idlw-help.el'.....	12, 31
'idlw-help.txt'.....	12, 31
'idlw-rinfo.el'.....	31
IDLWAVE in a Nutshell.....	2
IDLWAVE major mode.....	8
IDLWAVE shell.....	21
IDLWAVE, homepage.....	27
idlwave-abbrev-change-case.....	18
idlwave-abbrev-move.....	17
idlwave-abbrev-start-char.....	17
idlwave-auto-fill-split-string.....	9
idlwave-auto-routine-info-updates.....	30
idlwave-begin-line-comment.....	9
idlwave-block-indent.....	8
idlwave-class-arrow-face.....	15
idlwave-code-comment.....	9

idlwave-complete-empty-string-as-lower-case.....	14
idlwave-completion-case.....	14
idlwave-completion-fontify-classes.....	15
idlwave-completion-force-default-case.....	14
idlwave-completion-restore-window-configuration.....	15
idlwave-completion-show-classes.....	15
idlwave-continuation-indent.....	8
idlwave-default-font-lock-items.....	10
idlwave-do-actions.....	17
idlwave-doc-modifications-keyword.....	19
idlwave-doelib-end.....	19
idlwave-doelib-start.....	19
idlwave-end-offset.....	8
idlwave-expand-generic-end.....	17
idlwave-extra-help-function.....	13
idlwave-file-header.....	19
idlwave-fill-comment-line-only.....	9
idlwave-function-completion-adds-paren.....	15
idlwave-hang-indent-regex.....	9
idlwave-hanging-indent.....	9
idlwave-help-activate-links-agressively.....	13
idlwave-help-application.....	20
idlwave-help-directory.....	13
idlwave-help-fontify-source-code.....	13
idlwave-help-frame-parameters.....	13
idlwave-help-link-face.....	13
idlwave-help-source-try-header.....	13
idlwave-help-use-dedicated-frame.....	13
idlwave-highlight-help-links-in-completion.....	15
idlwave-keyword-completion-adds-equal.....	14
idlwave-libinfo-file.....	30
idlwave-library-path.....	30
idlwave-load-hook.....	20
idlwave-main-block-indent.....	8
idlwave-max-popup-menu-items.....	13
idlwave-mode-hook.....	20
idlwave-no-change-comment.....	8
idlwave-pad-keyword.....	18
idlwave-query-class.....	15
idlwave-query-shell-for-routine-info.....	30
idlwave-reindent-end.....	18
idlwave-reserved-word-upcase.....	19
idlwave-resize-routine-help-window.....	11
idlwave-rinfo-max-source-lines.....	11
idlwave-scan-all-buffers-for-routine-info.....	30
idlwave-shell-activate-prefix-keybindings.....	24
idlwave-shell-arrows-do-history.....	23
idlwave-shell-automatic-start.....	21
idlwave-shell-breakpoint-face.....	25
idlwave-shell-command-line-options.....	21
idlwave-shell-debug-modifiers.....	24

idlwave-shell-explicit-file-name 21
 idlwave-shell-expression-face 25
 idlwave-shell-file-name-chars 23
 idlwave-shell-frame-parameters 22
 idlwave-shell-graphics-window-size 23
 idlwave-shell-initial-commands 21
 idlwave-shell-input-mode-spells 23
 idlwave-shell-mark-breakpoints 25
 idlwave-shell-mark-stop-line 24
 idlwave-shell-mode-hook 22
 idlwave-shell-overlay-arrow 24
 idlwave-shell-prefix-key 23
 idlwave-shell-print-expression-function .. 26
 idlwave-shell-process-name 21
 idlwave-shell-prompt-pattern 21
 idlwave-shell-stop-line-face 24
 idlwave-shell-temp-pro-prefix 22
 idlwave-shell-use-dedicated-frame 21
 idlwave-shell-use-input-mode-magic 23
 idlwave-shell-use-toolbar 24
 idlwave-show-block 17
 idlwave-special-lib-alist 11, 30
 idlwave-split-line-string 9
 idlwave-startup-message 20
 idlwave-store-inquired-class 15
 idlwave-support-inheritance 15
 idlwave-surround-by-blank 18
 idlwave-system-directory 30
 idlwave-timestamp-hook 19
 idlwave-use-last-hang-indent 9
 ‘Imenu’, Emacs package 19
 Indentation 8
 Indentation, of foreign code 8
 Input mode 23
 Inserting keywords, from routine info 11
 Installation 27
 Installing online help 12, 27
 Interactive Data Language 1
 Interface Definition Language 1
 Interview, with the maintainer 32
 Introduction 1

K

Key bindings, in help application 13
 Keybindings for debugging 23
 Keyword completion 14
 Keywords of a routine 10
 Killing autoloaded buffers 16

L

Library catalog 30
 Library scan 30
 Line input mode (Shell) 23
 Line splitting 9

LINKIMAGE, IDL routine 29
 Load-path shadows 10, 31

M

M-? 12
 M-C-\ 8
 M-q 9
 M-(RET) 9
 M-(TAB) 14, 27
 MacOS 21, 30, 31
 Magic spells, for input mode 23
 Maintainer, of IDLWAVE 28
 Major mode, `idlwave-mode` 8
 Major mode, `idlwave-shell-mode` 21
 Method completion 14
 Mixed case completion 14
 Modification timestamp 19
 Module source file 16
 Motion commands 19
 Mouse binding to print expressions 25
 Multiply defined routines 11, 31

N

Nutshell, IDLWAVE in a 2

O

OBJ_NEW, special online help 12
 Object method completion 14
 Object methods 15
 Old variables, renaming 27
 Online Help 12
 Online Help from the routine info buffer 11
 Online Help in ‘*Completions*’ buffer 14
 Online Help, in the shell 22
 Online Help, Installation 12, 27
 Operators, padding with spaces 18

P

Padding operators with spaces 18
 Paragraphs, filling 8
 Paragraphs, hanging 8
 Perl program, to create ‘`idlw-rinfo.el`’ 31
 PRINT expressions 25
 Printing expressions, on calling stack 25
 Procedure definitions, jumping to 19
 Procedure name completion 14
 Program structure, moving through 19
 Programs, compiling 24

Q

Quick-Start 3

R

Renaming old variables 27

RESOLVE_ROUTINE 16

Restrictions for expression printing 25

Routine definitions 29

Routine definitions, multiple 11, 31

Routine info 10

Routine info sources 29

Routine info, in the shell 22

Routine source file 16

Routine source information 10

ROUTINE_NAMES, IDL procedure 25

Routines, resolving 16

S

Scanning buffers for routine info 10, 29

Scanning the documentation 31

Scrolling the ***Completions*** window 14**self** object, default class 15

Shadows, load-path 10, 31

Shell, basic commands 22

Shell, querying for routine info 10, 29

Shell, starting 21

Source code, as online help 12

Source file, access from routine info 11

Source file, of a routine 16

Sources of routine information 29

Space, around operators 18

Speed, of online help 12

Spells, magic 23

Splitting, of lines 9

Starting the shell 21

Stepping 24

String splitting 9

Subprocess of Emacs, IDL 21

Summary of important commands 2

Syntax highlighting 10

T

Templates 16

Thanks 28

Timestamp, in doc header 19

Toolbar 23

Tutorial 3

U

Uppcase, enforcing for reserved words 18

Updating routine info 10, 29

Upgrading from old **'idl.e1'** 27

URL, homepage for IDLWAVE 27

W

Windows 21, 30, 31

X

XEmacs package IDLWAVE 27