IDLWAVE User Manual

Emacs major mode and shell for IDL
Edition 4.10, November 2001

by Carsten Dominik & J.D. Smith

This is edition 4.10 of the *IDLWAVE User Manual* for IDLWAVE version 4.10, November 2001.

Table of Contents

1 Introduction

IDLWAVE is a package which supports editing source files for the Interactive Data Language (IDL[1]), and for running IDL as an inferior shell. It also can be used for editing source for the related WAVE/CL language, but only with limited support. Note that this package has nothing to do with the Interface Definition Language, part of the Common Object Request Broker Architecture (CORBA).

IDLWAVE consists of two main parts: a major mode for editing command files (`idlwave-mode`) and a mode for running the IDL program as an inferior shell (`idlwave-shell-mode`). Both modes work together closely and form a complete development environment. Here is a brief summary of what IDLWAVE does:

- Code indentation and formatting.
- Three level syntax highlighting support.
- Context-sensitive display of calling sequences and keywords for more than 1000 native IDL routines, extendible to any number of additional routines in your local IDL libraries.
- Name space conflict search, with likelihood ranking.
- Fast, context-sensitive online help.
- Context sensitive completion of routine names and keywords.
- Easy insertion of code templates.
- Automatic type correction to enforce a variety of customizable coding standards.
- Integrity checks for logical blocks.
- 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.
- Compilation, execution and interactive 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.

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 29 for information on how to switch.

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. Some configuration examples are also given in the appendix.

[1] IDL is a registered trademark of Research Systems, Inc.

2 IDLWAVE in a Nutshell

Editing IDL Programs

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

Running the IDLWAVE Shell, Debugging Programs

C-c C-s	Start IDL as a subprocess and/or switch to the interaction buffer.
M-p	Cycle back through IDL command history.
M-n	Cycle forward.
M-⟨TAB⟩	Complete a procedure name, function name or keyword in the shell buffer.
C-c C-d C-c	Save and compile the source file in the current buffer.
C-c C-d C-x	Goto next syntax error.
C-c C-d C-b	Set a breakpoint at the nearest viable source line.
C-c C-d C-d	Clear the nearest breakpoint.
C-c C-d C-p	Print the value of the expression near point in IDL.

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
;; Start autoloading routine info after 2 idle seconds
(setq idlwave-init-rinfo-when-idle-after 2)
;; 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))
;; Specify the online help files' location.
(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 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 there are many more capabilities in IDLWAVE than covered here, which can be discovered by reading the entire manual.

It is assumed that you have access to Emacs or XEmacs with the full IDLWAVE package including online help (see Chapter 6 [Installation], page 29). 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, as shown in the mode line just below the editing window. Also, the menu bar should contain entries '`IDLWAVE`' and '`Debug`'.

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` followed by `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. 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 weekday 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 and 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, 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-`⟨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 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 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 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.

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.

You likely have your own indentation preferences for IDL code. For example, I like to indent the main block of an IDL program from the margin, 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 *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 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.

You may also find the key bindings used for the debugging commands too long and complicated. Often we have heard, "Do I really have to type *C-c C-d C-c* to get a single simple command?" Due to Emacs rules and conventions, shorter bindings cannot be default, but you can enable them. 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 ⟨HYPER⟩ key, you could even shorten that:

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

instead to get compilation on *H-c*.

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 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 23). 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 one be automatically loaded whenever you use IDLWAVE. You may find it necessary to rebuild the catalog on occasion as your local libraries change. 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 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

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.

Continuation lines normally receive a fixed indentation, but in pro/function definition statements and in procedure calls, continuation lines are aligned with the first character after the routine name. In lines with unmatched opening parenthesis, the indentation is given by the opening parenthesis. Here is an example.

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

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

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). See Section 4.9 [Actions], page 19, 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 (`4`) 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 normal continuation lines.

idlwave-indent-to-open-paren (`t`) User Option
    Non-`nil` means, indent continuation lines to innermost open 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 '`$`'. If you would like to start a continuation line, use *M-*⟨RET⟩, which calls the command `idlwave-split-line`. It inserts the continuation character '`$`', 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.

```
; 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` (in 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 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.

4.2 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. This large set makes it difficult to remember the calling sequence and keywords of a specific command, but IDLWAVE can help. It contains a list of all builtin routines, with calling sequences and keywords[1]. It also scans Emacs buffers and library files for routine definitions, and queries the IDLWAVE-Shell for routines currently compiled in the shell. 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:

[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.

```
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 33). The information about the calling sequence and keywords is derived from the first source listed. Library routines are supported only if you have scanned your local IDL libraries (see Section A.3 [Library Catalog], page 32). The source entry consists of a *source category*, a set of *flags* and the path to the *source file*. The following categories exist:

System	A system routine of unknown origin. When the system library has been scanned (see Section A.3 [Library Catalog], page 32), this category will automatically split into the next two.
Builtin	A builtin system routine with no source code available.
SystemLib	A library system routine in the official lib directory '`!DIR/lib`'.
Obsolete	A library routine in the official lib directory '`!DIR/lib/obsolete`'.
Library	A routine in a file on IDL's search path `!PATH`.
Other	Any other routine with a file not known to be on the search path.
Unresolved	An otherwise unkown routine the shell lists as unresolved (referenced, but not compiled).

You can create additional categories based on the routine's filepath with the variable `idlwave-special-lib-alist`. This is useful for easy discrimination of various libraries, or even versions of the same library.

The flags `[CSB]` indicate the source of the information IDLWAVE has regarding the file: from a library catalog (`[C--]`, see Section A.3 [Library Catalog], page 32), from the IDL Shell (`[-S-]`) or from an Emacs buffer (`[--B]`). Combinations are possible (a compiled library routine visited in a buffer might read `[CSB]`. 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:

Usage	If online help is installed, a click with the *right* mouse button on the *Usage:* line will access the help for the routine (see Section 4.3 [Online Help], page 12).

Keyword Online help about keywords is also available with the *right* mouse button. Clicking on a keyword with the *middle* mouse button will insert this keyword in the buffer from where `idlwave-routine-info` was called. Holding down ⟨SHIFT⟩ while clicking also adds the initial '`/`'.

Source Clicking with the *middle* 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 *`C-c ?`* was called. If you use the *right* mouse button, the source will not be visited by a buffer, but displayed in the online help window.

Classes The *Classes* line is only included in the routine info window if the current class inherits from other classes. You can click with the *middle* 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, because usually IDLWAVE gets you to the right place in the docs directly, without any additional browsing and scrolling. For this online help to work, an ASCII version of the IDL documentation, which is not part of the standalone IDLWAVE distribution, is required. The necessary help files can be downloaded from the maintainers webpage. The text extracted from the PDF files is fine for normal text, but graphics and multiline equations will not be well formatted. See also Section A.5 [Documentation Scan], page 34.

Occasionally RSI releases a synopsis of new features in an IDL release, without simultaneously updating the documentation files, instead preferring a *What's New* document which describes the changes. These updates are incorporated directly into the IDLWAVE online help, and are delimited in `<NEW>..</NEW>` blocks.

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 (scrolling down to appropriate keywords, e.g.). Otherwise the routine definition statement (`pro`/`function`) is shown.

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 as context:

Routine name	The name of a routine (function, procedure, method).
Keyword Parameter	A keyword parameter of a routine.
System Variable	System variables like `!DPI`.
IDL Statement	Statements like `PRO`, `REPEAT`, `COMPILE_OPT`, etc.
Class name	A class name in an `OBJ_NEW` call.
Class Init	Beyond the class name in an `OBJ_NEW` call.
Executive Command	An executive command like `.RUN`. Mostly useful in the shell.
Default	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, but an attempt will be made to visit non-blue items in the directly in the originating source file.

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

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

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

h	Jump to DocLib Header of the routine whose source is displayed as help.
H	Jump to the first DocLib Header in the file.
`.` (Dot)	Jump back and forth between the routine definition (the `pro`/`function` statement) and the description of the help item in the DocLib header.
F	Fontify the buffer like source code. See the variable `idlwave-help-fontify-source-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 popup 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 in IDLWAVE online help.

idlwave-help-activate-links-aggressively (`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, keywords, system variables, class structure tags and file names. As in many programming modes, completion is bound to *M-*⟨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-*⟨TAB⟩ would try to complete when the cursor is on the position marked with a '`_`'.

`plo_`	Procedure
`x = a_`	Function
`plot,xra_`	Keyword of `plot` procedure
`plot,x,y,/x_`	Keyword of `plot` procedure
`plot,min(_`	Keyword of `min` function
`obj -> a_`	Object method (procedure)
`a(2,3) = obj -> a_`	Object method (function)
`x = obj_new('IDL_`	Class name
`x = obj_new('MyCl',a_`	Keyword to `Init` method in class `MyCl`
`pro A_`	Class name
`pro _`	Fill in `Class::` of first method in this file
`!v_`	System variable
`!version.t_`	Structure tag of system variable

`self.g_`	Class structure tag in methods
`name = 'a_`	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. You can force completion of a function name such a location with a prefix arg: *C-u M-*⟨TAB⟩.

If the list of completions is too long to fit in the '`*Completions*`' window, the window can be scrolled by pressing *M-*⟨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 is almost always omitted in the calling source, IDLWAVE considers all available methods in all classes as possible method name completions. The combined keywords of the current method in all available classes 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`' 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, consider inheritance during 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 from nearby 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.

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 32). 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 31, for more information about how 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 keybindings:

C-c C-c	`CASE` statement template
C-c C-f	`FOR` loop template
C-c C-r	`REPEAT` loop template
C-c C-w	`WHILE` loop template

All code templates are also available as abbreviations (see Section 4.8 [Abbreviations], page 17).

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:

`\pr`	`PROCEDURE` template
`\fu`	`FUNCTION` template
`\c`	`CASE` statement template
`\f`	`FOR` loop template
`\r`	`REPEAT` loop template
`\w`	`WHILE` loop template
`\i`	`IF` statement template
`\elif`	`IF-ELSE` statement template
`\b`	`BEGIN`

String abbreviations:

`\b`	`begin`
`\cb`	`byte()`
`\cc`	`complex()`
`\cd`	`double()`
`\cf`	`float()`
`\cl`	`long()`
`\co`	`common`
`\cs`	`string()`
`\cx`	`fix()`
`\e`	`else`
`\ec`	`endcase`
`\ee`	`endelse`
`\ef`	`endfor`
`\ei`	`endif else if`
`\el`	`endif else`
`\en`	`endif`
`\er`	`endrep`
`\es`	`endswitch`
`\ew`	`endwhile`
`\g`	`goto,`
`\h`	`help,`
`\ine`	`if n_elements() eq 0 then`
`\inn`	`if n_elements() ne 0 then`
`\k`	`keyword_set()`
`\n`	`n_elements()`
`\np`	`n_params()`
`\oi`	`on_ioerror,`
`\or`	`openr,`
`\ou`	`openu,`
`\ow`	`openw,`
`\p`	`print,`
`\pt`	`plot,`
`\re`	`read,`
`\rf`	`readf,`
`\rt`	`return`
`\ru`	`readu,`
`\s`	`size()`
`\sc`	`strcompress()`
`\sl`	`strlowcase()`
`\sm`	`strmid()`
`\sn`	`strlen()`
`\sp`	`strpos()`
`\sr`	`strtrim()`
`\st`	`strput()`
`\su`	`strupcase()`
`\t`	`then`
`\u`	`until`

`\wc`	`widget_control,`
`\wi`	`widget_info()`
`\wu`	`writeu,`

You can easily add your own abbreviations with `define-abbrev` inserted your mode hook, e.g.:

```
(add-hook 'idlwave-mode-hook
          (lambda ()
            (define-abbrev idlwave-mode-abbrev-table
              (concat idlwave-abbrev-start-char "ik") "if keyword_set() then"
              (idlwave-keyword-abbrev 6))
```

Notice how the abbreviation start character is prepended to *ik*, and that the argument *6* to `idlwave-keyword-abbrev` specifies how far back to move the point (to put 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`.

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.

4.9 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 *`C-M-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.

4.9.1 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 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.

4.9.2 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 '`->`'[2], 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.

[2] Operators longer than one character can only be padded during line indentation.

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`.

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`.

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. 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.

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

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

4.12 Miscellaneous Options

idlwave-help-application User Option

The external application providing reference help for programming.

idlwave-startup-message (`t`) 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 Emacs[1] - so the IDLWAVE shell currently only works under 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. 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, define an 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.

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

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 (`nil`) User Option

Non-`nil` means attempt to invoke idlwave-shell if not already running.

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

idlwave-shell-initial-commands User Option
: Initial commands, separated by newlines, to send to IDL.

idlwave-shell-save-command-history (`t`) User Option
: Non-`nil` means preserve command history between sessions.

idlwave-shell-command-history-file (‘`~/.idlwhist`’) User Option
: The file in which the command history of the idlwave shell is saved.

idlwave-shell-use-dedicated-frame (`nil`) User Option
: Non-`nil` means, IDLWAVE should use a special frame to display shell buffer.

idlwave-shell-frame-parameters User Option
: The frame parameters for a dedicated idlwave-shell frame.

idlwave-shell-raise-frame (`t`) User Option
: Non-`nil` means, ‘idlwave-shell’ raises the frame showing the shell window.

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 `idlwave-shell-mode`.

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[2]. The history is preserved between emacs sessions. Here is a list of commonly used commands.

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

[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`.

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 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.

⟨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 =	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 Debugging IDL Programs

Programs can be compiled, run, and debugged directly from the source buffer in Emacs. The IDLWAVE shell installs keybindings 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 keybindings 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 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 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 (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, the breakpoint will only be active the `nth` time it is hit. 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`). Breakpoint lines are highlighted in the source code.

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; *C-c C-d C-n* to do one step but treat procedure and function calls as a single step; *C-c C-d C-h* to continue execution to the line where the cursor is in and *C-c C-d C-r* to continue execution. Here is a summary of the breakpoint and stepping commands:

C-c C-d C-b	Set breakpoint (`idlwave-shell-break-here`)
C-c C-d C-i	Set breakpoint in function named here (`idlwave-shell-break-in`)
C-c C-d C-d	Clear current breakpoint (`idlwave-shell-clear-current-bp`)
C-c C-d C-a	Clear all breakpoints (`idlwave-shell-clear-all-bp`)
C-c C-d C-s	Step, into function calls (`idlwave-shell-step`)
C-c C-d C-n	Step, over function calls (`idlwave-shell-stepover`)
C-c C-d C-k	Skip one statement (`idlwave-shell-skip`)
C-c C-d C-u	Continue to end of block (`idlwave-shell-up`)
C-c C-d C-m	Continue to end of function (`idlwave-shell-return`)
C-c C-d C-o	Continue past end of function (`idlwave-shell-out`)
C-c C-d C-h	Continue to line at cursor position (`idlwave-shell-to-here`)
C-c C-d C-r	Continue execution to next breakpoint (`idlwave-shell-cont`)
C-c C-d C-up	Show higher level in calling stack (`idlwave-shell-stack-up`)
C-c C-d C-down	Show lower level in calling stack (`idlwave-shell-stack-down`)

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 Walking the Calling Stack

When debugging a program, it can be very useful to check in what context the current routine was called, and why the arguments of the call are the way they are. For this one needs 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 through the calling stack. The mode line of the shell window will indicate where you are on the stack with a token like '`[-3:MYPRO]`', and the line of IDL code which did the current call will be highlighted. When you continue execution, IDLWAVE will automatically return to the current level. See Section 5.3.4 [Examining Variables], page 28, for information how to examine the value of variables and expressions on higher calling stack levels.

5.3.4 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. Two prefix arguments (*C-u C-u C-c C-d C-p*) will use the current region as expression.

It is very convenient to click with the mouse on expressions to retrieve their value. Use *S-mouse-2* to print an expression and *C-S-mouse-2* to get help on an expression. I.e. you need to hold down ⟨SHIFT⟩ and ⟨CONTROL⟩ while clicking with the middle mouse button. If you simply click, the expression will be selected in the same way as described above. But you can also drag the mouse in order to define the expression.

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. See Section 5.3.3 [Walking the Calling Stack], page 28, for information on how to step back to higher levels on the calling stack. Commands which print values of variables and expressions will then use the values of variables in the calling routine. 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 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 separately[1] (see Section 6.2 [Installing Online Help], page 29).

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

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. 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 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**, the original author. Chris wrote '`idl.el`' and '`idl-shell.el`' 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. I helped shape object method completion and most new features introduced in versions 4.x.

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 oder 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 4.10 is distributed with a list of 1322 routines and 5952 keywords, reflecting IDL version 5.5. 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 34, 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 32, for information how to scan library files.

Loading routine and catalog information is a time consuming process. Depending on the system and network configuration it takes between 2 and 30 seconds. In order to minimize the waiting time upon your first completion or routine info command in a session, IDLWAVE uses Emacs idle time to do the initialization if 5 steps. If this gets into your way, set the variable `idlwave-init-rinfo-when-idle-after` to 0 (the number zero).

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 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`, like this[1]:

```
(setq idlwave-library-path
        '("+c:/RSI/IDL54/lib/" "+c:/user/me/idllibs" ))
(setq idlwave-system-directory "c:/RSI/IDL54/")
```

Under UNIX, 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 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[2]. You can continue to work, and the library catalog will be re-read when it is ready.

[1] The initial '`+`' leads to recursive expansion of the path, just like in IDL

[2] Unix systems only, I think.

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 Unix.

idlwave-system-directory User Option

The IDL system directory for Windows and MacOS. Not needed under 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 commands 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 it 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.

One particularly frustrating situation occurs when a new IDL version is released without the associated documentation updates. Instead, a *What's New* file containing new and updated documentation is shipped alongside the previous version's reference material. The '`get_rinfo`' script can merge this new information into the standard help text and routine information, as long as it is pre-formatted in a simple way. See '`get_rinfo`' for more information.

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-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, 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 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
```

Appendix C Windows and MacOS

IDLWAVE was developed on a UNIX system. However, due 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 problem really is that RSI does not provide a command-line version of IDL for Windows or MacOS which IDLWAVE could communicate with[1]. Therefore 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: You must specify path names which on a UNIX can be 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\IDL55`'.

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

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

;; location of the IDL system directory (try "print,!DIR")
(setq idlwave-system-directory "c:/RSI/IDL55/")

;; location of the IDLWAVE help files idlw-help.el and idlw-help.txt.
(setq idlwave-help-dir "c:/IDLWAVE")

;; file in which to store the user catalog info
(setq idlwave-libinfo-file "c:/IDLWAVE/idlcat.el")
```

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 files under the names '`idlwave.tar.gz`' and '`idlwave-help-tar.gz`'.
- Be sure that your software for untarring/ungzipping is *NOT* doing smart CR/LF conversion (WinZip users will find this in Options:Configuration:Miscellaneous, change the setting, then re-open the archive). This adds one byte per line, throwing off the byte-counts for the help file lookups and defeating fast online help lookup.
- M-TAB switches among running programs - use Esc-TAB instead. FIXME: unfinished.

[1] Call your RSI representative and complain - it should be trivial for them to provide one. And if enough people ask for it, maybe they will.

Index