
GPS Documentation

Release 6.0.1

AdaCore

January 13, 2014

CONTENTS

1	Description of the Main Window	3
1.1	The Workspace	3
1.2	The Welcome Dialog	6
1.3	The Tip of the Day	8
1.4	The Menu Bar	8
1.5	The Tool Bar	9
1.6	The omni-search	10
1.7	The <i>Messages</i> window	13
1.8	The <i>Locations</i> view	14
1.9	The <i>Project</i> view	16
1.10	The <i>Scenario</i> view	20
1.11	The <i>Files</i> View	23
1.12	The <i>Windows</i> view	24
1.13	The <i>Outline</i> view	26
1.14	The <i>Clipboard</i> view	29
1.15	The <i>Call trees</i> view and <i>Callgraph</i> browser	30
1.16	The <i>Bookmarks</i> view	32
1.17	The <i>Shell</i> and <i>Python</i> Windows	33
1.18	The OS shell window	34
1.19	The Execution window	34
1.20	The <i>Task Manager</i>	34
1.21	The <i>Project Browser</i>	35
1.22	The <i>Dependency Browser</i>	36
1.23	The <i>Elaboration Circularities</i> browser	38
1.24	The <i>Entity Browser</i>	39
1.25	The File Selector	40
2	Multiple Document Interface	43
2.1	Window layout	43
2.2	Selecting Windows	43
2.3	Closing Windows	44
2.4	Splitting Windows	44
2.5	Floating Windows	45
2.6	Moving Windows	45
2.7	Perspectives	46
3	Editing Files	47
3.1	General Information	47
3.2	Editing Sources	50

3.3	Menu Items	50
3.4	Rectangles	56
3.5	Recording and replaying macros	58
3.6	Contextual Menus for Editing Files	58
3.7	Handling of case exceptions	59
3.8	Refactoring	59
3.9	Using an External Editor	64
3.10	Using the Clipboard	65
3.11	Saving Files	65
4	Source Navigation	67
4.1	Support for Cross-References	67
4.2	The Navigate Menu	68
4.3	Contextual Menus for Source Navigation	69
4.4	Navigating with hyperlinks	71
4.5	Highlighting dispatching calls	71
5	Project Handling	73
5.1	Description of the Projects	73
5.2	Supported Languages	75
5.3	Scenarios and Configuration Variables	76
5.4	Extending Projects	79
5.5	Disabling Project Edition Features	80
5.6	The Project Menu	80
5.7	The Project Wizard	81
5.8	The Project Dependencies Editor	89
5.9	The Project Properties Editor	90
5.10	The Switches Editor	92
6	Searching and Replacing	95
7	Compilation/Build	99
7.1	The Build Menu	99
7.2	The Target Configuration Dialog	101
7.3	The Build Mode	103
7.4	Working with two compilers	103
8	Debugging	105
8.1	The Debug Menu	105
8.2	The Call Stack Window	108
8.3	The Data Window	109
8.4	The Breakpoint Editor	113
8.5	The Memory Window	115
8.6	Using the Source Editor when Debugging	116
8.7	The Assembly Window	117
8.8	The Debugger Console	118
8.9	Customizing the Debugger	119
9	Version Control System	121
9.1	The VCS Explorer	122
9.2	The VCS Activities	124
9.3	The VCS Menu	125
9.4	The Version Control Contextual Menu	126
9.5	Working with global ChangeLog file	128
9.6	The Revision View	129

10	Tools	131
10.1	The Tools Menu	131
10.2	Coding Standard	132
10.3	Visual Comparison	133
10.4	Code Fixing	134
10.5	Documentation Generation	135
10.6	Working With Unit Tests	137
10.7	Metrics	138
10.8	Code Coverage	138
10.9	Stack Analysis	141
11	Working in a Cross Environment	145
11.1	Customizing your Projects	145
11.2	Debugger Issues	146
12	Using GPS for Remote Development	147
12.1	Requirements	147
12.2	Setup the remote servers	148
12.3	Setup a remote project	150
12.4	Limitations	152
13	Customizing and Extending GPS	153
13.1	The Preferences Dialog	153
13.2	GPS Themes	165
13.3	The Key Manager Dialog	166
13.4	The Plug-ins Editor	167
13.5	Customizing through XML and Python files	168
13.6	Adding support for new tools	207
13.7	Customization examples	216
13.8	Scripting GPS	218
13.9	Adding support for new Version Control Systems	234
13.10	The Server Mode	240
13.11	Adding project templates	241
14	Environment	243
14.1	Command Line Options	243
14.2	Environment Variables	244
14.3	Files	244
14.4	Reporting Suggestions and Bugs	246
14.5	Solving Problems	246
15	Scripting API reference for GPS	249
15.1	Function description	249
15.2	User data in instances	249
15.3	Hooks	250
15.4	Functions	250
15.5	Classes	259
16	Useful plug-ins	377
16.1	User plug-ins	377
16.2	Helper plug-ins	377
17	GNU Free Documentation License	385
17.1	PREAMBLE	385
17.2	APPLICABILITY AND DEFINITIONS	385

17.3	VERBATIM COPYING	386
17.4	COPYING IN QUANTITY	386
17.5	MODIFICATIONS	387
17.6	COMBINING DOCUMENTS	388
17.7	COLLECTIONS OF DOCUMENTS	388
17.8	AGGREGATION WITH INDEPENDENT WORKS	388
17.9	TRANSLATION	389
17.10	TERMINATION	389
17.11	FUTURE REVISIONS OF THIS LICENSE	389
17.12	ADDENDUM: How to use this License for your documents	389
18	Indices and tables	391
	Python Module Index	393
	Index	395



GPS is a complete integrated development environment that gives access to a wide range of tools and integrates them smoothly. It integrates especially well with AdaCore's tools, but can easily be extended to drive other tools, through small plug-ins written in python.

Here is a rough list of the GNAT Programming Studio features:

- *Multiple Document Interface*

GPS is based on a multiple document interface, which allows you to organize windows the way you want, float them to other screens, drag them to other places to reorganize your desktop, and of course restore the desktop the next time GPS is restarted.

- Built-in editor (*Editing Files*)

Fully customizable editor with syntax highlighting, smart completion of text, multiple views of the same file, automatic indentation, block-level navigation, support for Emacs keybindings, code folding, refactoring, visual comparison of files, alias expansion,...

- Support for compile/build/run cycle (*Compilation/Build*)

Any command line compiler can be integrated in GPS, with built in support for GNAT, gcc and make. Error messages are easily navigable, and automatic code fixing is provided for a number of typical error messages. This includes support for cross-compilers, or running compilers on separate hosts than the machine on which GPS itself is running.

- Project management (*Project Handling*)

Project files (editable either graphically or manually) are used to describe the location of sources, their naming schemes, how they should be built,.... Graphical browsers exist to analyze the dependencies between your projects and the sources within your projects.

- Integration with various *Version Control System*

CVS, subversion, git and clearcase are supported out of the box, but others can be added by customizing some XML plug-ins.

- Intelligent *Source Navigation*

By leveraging on information provided by the compilers, or using its own parses, GPS makes it possible to find the declaration of entities, their references,... It also provides advanced capabilities like call graphs, UML-like entity browsers,...

- Full debugger integration (*Debugging*)

GPS integrates fully with gdb, and provides multiple graphical views to monitor the state of your application, include a call stack, a visual display for the values of the variables, a breakpoint editor,...

- Integration with code analysis tools (*Tools*)

GPS integrates well with various command-line tools like gcov and GNATcoverage (for the coverage of your code), codepeer and Spark (to analyze your code),... In a lot of cases, it provides nice graphical rendering of their output, often integrated with the editor itself so that the information is available where you need it.

- Fully customizable (*Customizing and Extending GPS*)

GPS provides an extensive python API which allows you to customize existing features, or develop your own new plug-ins easily. Simpler customization can be done through one of the numerous preferences or local settings.

DESCRIPTION OF THE MAIN WINDOW

The GNAT Programming Studio has one main window, where most of your work will be performed. However, it is also very flexible in how it lets you organize your desktop, which will be discussed in a later section (*Multiple Document Interface*).

But there are also various other windows that might pop up at various times, and this section documents them.

1.1 The Workspace

The whole work space is based on a multiple document interface, *Multiple Document Interface*. It can contain any number of windows, the most important of which are probably the editors. However, GPS also provides a large number of views that can be added to the workspace. The following sections will list them all.

1.1.1 Common features of the views

Some views are part of the default desktop, and thus are visible by default. The other views can always be opened through one of the submenus of the *Tools* menu, most often *Tools* → *Views*.

Some of the view have their own local toolbar that contains shortcuts to the most often used features of the view.

To the right of these local toolbars, there is often a button to open a local settings menu. This menu can contain more actions that can be performed in this view, or various configuration settings that affect the behavior or the display of the view.

Some of the views also have a filter in their local toolbar. These filters can be used to reduce the amount of information that is displayed on the screen, by only leaving those lines that match the filter.

If you click on the left icon of the filter, this will bring up a popup menu to configure the filter:

- The first three entries are used to chose the search algorithm (from full text match, to regular expression, to fuzzy matching). These modes are similar to the ones used in the omni-search (*The omni-search*).
- The next entry is *Revert filter*. When this is selected, the lines that do not match the filter are displayed, as opposed to the ones that match the filter otherwise. This mode can also be enabled temporarily if you start the filter with the string *not:*. For instance, a filter in the *Locations* view that says *not:warning* will hide all warning messages.
- The last entry *Whole word* should be used when you only want to match on full words, not on substrings.

1.1.2 Common features of the browsers

A number of the views described below are interactive displays called browsers. They represent their information as boxes that can be manipulated with the mouse, and provide the following additional capabilities:

- Scrolling

When a lot of items are displayed in the canvas, the currently visible area might be too small to display all of them. In this case, scrollbars will be added on the sides, so that you can make other items visible. Scrolling can also be done with the arrow keys.

- Layout

A basic layout algorithm is used to organize the items. This algorithm is layer oriented: items with no parents are put in the first layer, then their direct children are put in the second layer, and so on. Depending on the type of browser, these layers are organized either vertically or horizontally. This algorithm tries to preserve as much as possible the positions of the items that were moved interactively.

The *Refresh layout* button in the local toolbar can be used to recompute the layout of items at any time, even for items that were previously moved interactively.

- Interactive moving of items

Items can be moved interactively with the mouse. Click and drag the item by clicking on its title bar. The links will still be displayed during the move, so that you can check whether it overlaps any other item. If you are trying to move the item outside of the visible part of the browser, the latter will be scrolled.

- Selecting items

Items can be selected by clicking on them. Multiple items can be selected by holding the `control` key while clicking in the item. Alternatively, you can click and drag the mouse inside the background of the browser. All the items found in the selection rectangle when the mouse is released will be selected.

Selected items are drawn with a different title bar color. All items linked to them also use a different title bar color, as well as the links. This is the most convenient way to understand the relationships between items when lots of them are present in the browser.

Buttons in the local toolbar are provided to remove either the selected items, or on the contrary the ones that are not selected.

- Links

Items can be linked together, and will remain connected when items are moved. Different types of links exist, see the description of the various browsers.

The local toolbar provides a button to hide the display of the links. This will keep the canvas more readable, at the cost of losing some information. You can also hide only a subset of the links. Even when the links are hidden, if you select an item then the items linked to it will still be highlighted.

The local settings menu in browsers has an option *straight links* which can be toggled if you prefer to have orthogonal links.

- export

The entire contents of a browser can be exported as a *PNG* or *SVG* images using the entry *Export to...* in the local toolbar.

- Zooming

Several different zoom levels are available. The local toolbar provides multiple buttons to change the zoom level: *zoom in*, *zoom out* and *zoom*. The latter is used to select directly the zoom level you want.

This zooming capability is generally useful when lots of items are displayed in the browser, to get a more general view of the layout and the relationships between the items.

- Hyper-links

Some of the items will contain hyper links, displayed in blue by default, and underlined. Clicking on these will generally display new items.

- contextual menus

Right-clicking on items will bring a contextual menu with actions that can be performed on that item. These actions are specific to the kind of item you clicked on.

- Grid

By default, a grid (small dots) is displayed in the background of the browsers. Using the local settings menu, it is possible to hide the grid (*Draw grid*) and to force items to align on the grid (*Align on grid*).

Icons for source language entities

Entities in the source code are presented with representative icons within the various GPS views (the *Outline* and *Project* views, for example). These icons indicate both the language categories of the entities, such as packages and methods, as well as compile-time visibility. In addition, the icons distinguish entity declarations from other entities. The same icons are used for all programming languages supported by the viewers, with language-specific interpretations for both compile-time visibility and recognizing declarations.

There are five language categories used for all supported languages:

- The *package* category's icon is a square.



- The *subprogram* category's icon is a circle.



- The *type* category's icon is a triangle.



- The *variable* category's icon is a dot.



- The *generic* category's icon is a diamond.



These basic icons are enhanced with decorators, when appropriate, to indicate compile-time visibility constraints and to distinguish declarations from completions. For example, the icons for entity declarations have a small 'S' decorator added, denoting a 'spec'.

With respect to compile-time visibility, icons for 'protected' and 'private' entities appear within an enclosing box indicating a visibility constraint. For entities with 'protected' visibility, this enclosing box is colored in gray. 'Private' entities are enclosed within a red box. The icons for 'public' entities have no such enclosing box. For example, a variable with 'private' visibility would be represented by an icon consisting of a dot enclosed within a red box.

These additional decorators are combined when appropriate. For example, the icon corresponding to the 'private' declaration of a 'package' entity would be a square, as for any package entity, with a small 'S' added, all enclosed within a red box.

Language constructs are mapped to the categories in a language-specific manner. For example, C++ namespaces and Ada packages correspond to the *package* category. C functions and Ada subprograms correspond to the *method*

category, and so on. The *generic* category is a general category representing other language entities, but note that not all possible language constructs are mapped to categories and icons. (Note also that the *generic* category does not correspond to Ada generic units or C++ templates.)

The names of the categories should not be interpreted literally in terms of language constructs because the categories are rather general, in order to limit the number used. The *variable* category includes both constants and variables in Ada, for example. Limiting the number of categories maintains a balance between presentation complexity and the need to support distinct programming languages.

Icons for a given entity may appear more than once within a view. For example, an Ada private type will have both a partial view in the visible part of the enclosing package as well as a full view in the private part of the package. Two triangle icons will therefore appear for the two occurrences of the type name, one with the additional decorator indicating the ‘private’ compile-time visibility.

1.2 The Welcome Dialog



When it starts, GPS is looking for a project file to load, so that it knows where to find the sources of your project. This project is in general specified on the command line (via a **-P** switch). Alternatively, if the current directory only contains one project file, GPS will select it automatically. Finally, if you specify the name of a source file to edit, GPS will load a default project and start the editing immediately. If no project file can be found, GPS displays a welcome dialog, which gives you the following choices:

Create new project from template If you select this option and then click the *OK* button, GPS will launch an assistant to create a project using one of the predefined project templates. This makes it easy to create GtkAda-based applications, or applications using the Ada Web Server, for instance.

Start with default project in directory

If you select this option and click on the *OK* button, GPS will first look for a project called `default.gpr` in the current directory and load it if found. Otherwise, it will copy in the current direc-

tory the default project found under `<prefix>/share/gps/default.gpr` and load it. GPS will remove this temporary copy when exiting or loading another project, if the copy has not been modified during the session.

The default project will contain all the Ada source files from the given directory (assuming they use the default GNAT naming scheme `.ads` and `.adb`).

If the current directory is not writable, GPS will instead load directly `<prefix>/share/gps/readonly.gpr`. In this case, GPS will work in a degraded mode, where some capabilities will not work (such as building and source navigation). This project does not contain any sources.

Create new project with wizard

Selecting this option and clicking on the *OK* button will start a wizard allowing you to specify most of the properties for a new project. Once the project is created, GPS will save it and load it automatically. See *The Project Wizard* for more details.

There are several kinds of wizards, ranging from creating a single project, to creating a set of project that attempt to adapt to an existing directory layout. The list of pages in the wizard will depend on the kind of project you want to create.

One of the wizard, *Project Tree*, will try and import a set of sources and object files, and attempt to create one or more project files so that building your application through these project files will put the objects in the same directory they are currently in. If you have not compiled your application when launching this wizard, GPS will create a single project file and all object files will be put in the same object directory. This is the preferred method when importing sources with duplicate file names, since the latter is only authorized in a single project file, not across various project files.

Open existing project

You can select an existing project by clicking on the *Browse* button, or by using a previously loaded project listed in the combo box. When a project is selected, click on the *OK* button to load this project and open the main window.

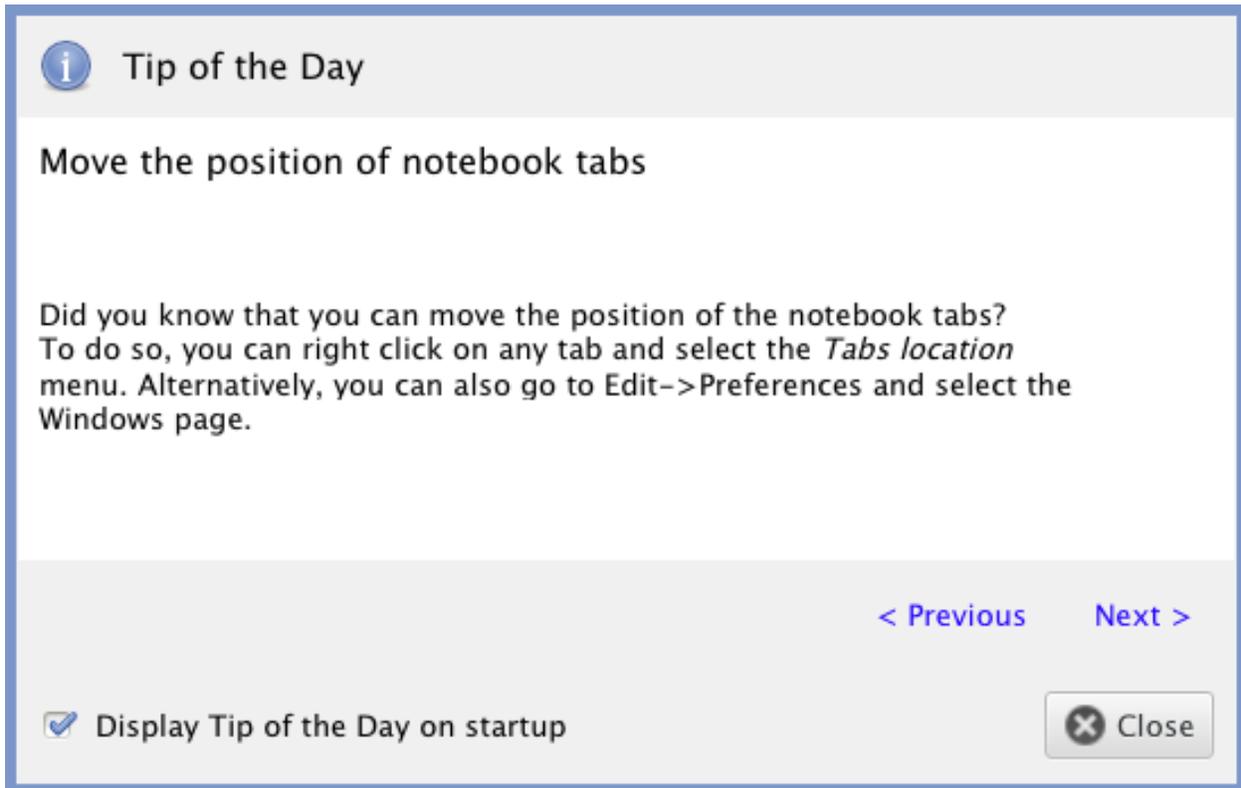
Always show this dialog when GPS starts

If unset, the welcome dialog won't be shown in future sessions. In this case, GPS will behave as follows: it will first look for a **-P** switch on the command line, and load the corresponding project if present; otherwise, it will look for a project file in the current directory and will load it if there is only one; if no project file was loaded, GPS will start with the default project, as if you had selected *Start with default project in directory* in the welcome dialog.

To reset this property, go to the menu *Edit* → *Preferences*.

Quit If you click on this button, GPS will terminate immediately.

1.3 The Tip of the Day



This dialog displays short tips on how to make the most efficient use of the GNAT Programming Studio. You can click on the *Previous* and *Next* buttons to access all tips, and close the dialog by either clicking on the *Close* button or pressing the `ESC` key.

You can also disable this dialog by unchecking the *Display Tip of the Day on startup* check box. If you would like to reenable this dialog, you can go to the *Edit* → *Preferences* dialog.

1.4 The Menu Bar

File Edit Navigate VCS Project Build Debug Tools Window Help

This is a standard menu bar that gives access to all the global functionalities of GPS. It is usually easier to access a given functionality using the various contextual menus provided throughout GPS: these menus give direct access to the most relevant actions given the current context (e.g. a project, a directory, a file, an entity, ...). Contextual menus pop up when the right mouse button is clicked or when using the special open contextual menu key on most PC keyboards.

The menu bar gives access to the following items:

- *File* (*The File Menu*)
- *Edit* (*The Edit Menu*)
- *Navigate* (*The Navigate Menu*)
- *VCS* (*The VCS Menu*)

- *Project (The Project Menu)*
- *Build (The Build Menu)*
- *Debug (The Debug Menu)*
- *Tools (The Tools Menu)*
- *SPARK*

If the SPARK toolset is installed on your system and available on your PATH, then this menu is available. See *Help* → *SPARK* → *Reference* → *Using SPARK with GPS* for more details.

- *CodePeer*

If the CodePeer toolset is installed on your system and available on your PATH, then this menu is available. See your CodePeer documentation for more details.

- *Window (Multiple Document Interface)*
- *Help*

1.5 The Tool Bar

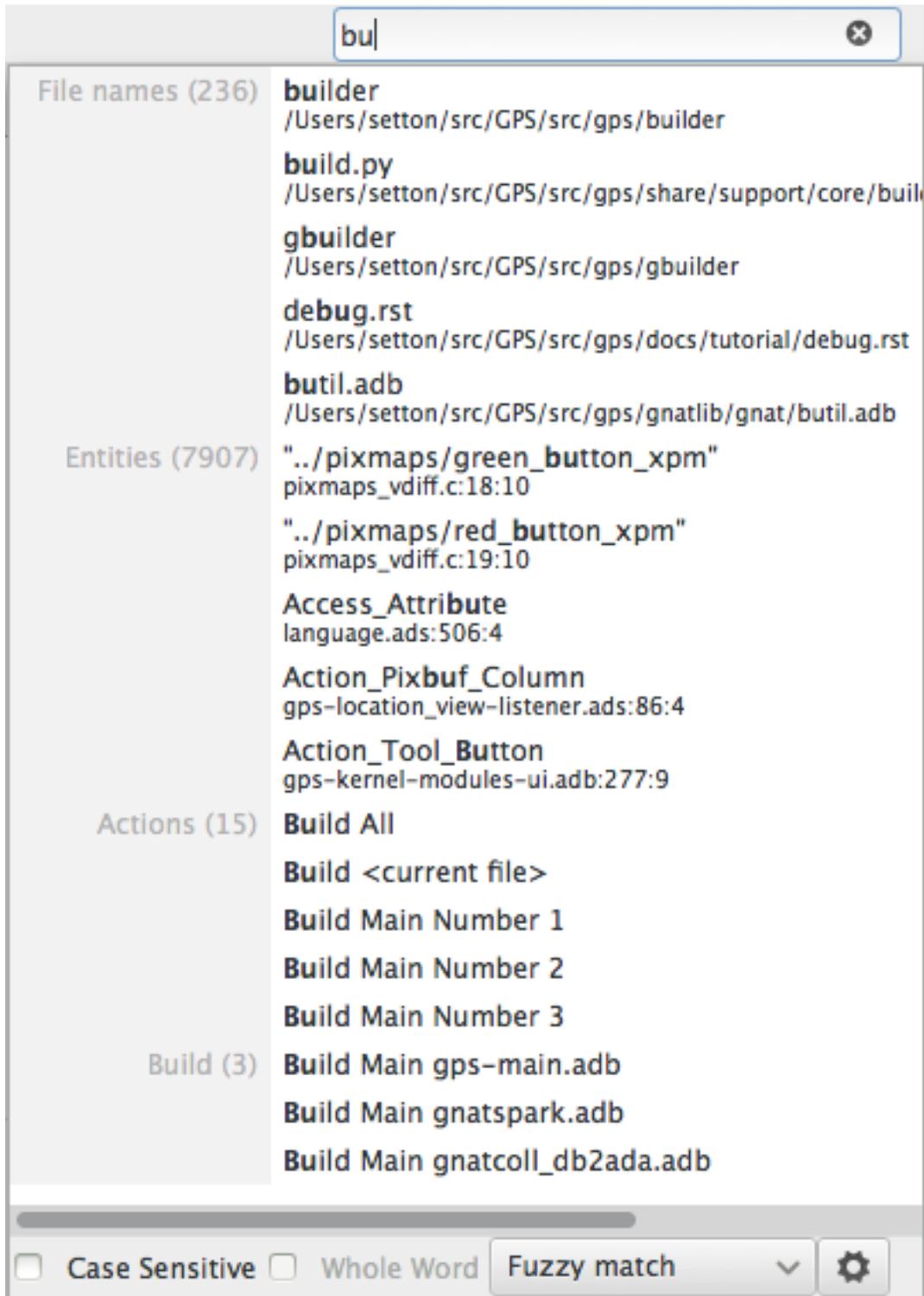


The tool bar provides shortcuts via buttons to some typical actions:

- creating a new file
- opening an existing file (see laso the omni-search on the right of the bar)
- saving the current file
- undo / redo last editing
- go to previous or next saved location
- multiple customizable buttons to build, clean or run your project
- when a debugger is started, multiple buttons to stop and continue the debugger, step to the next instruction,...

When GPS is performing background actions, like loading the cross-reference information, compiling or indeed all actions involving external processes, a progress bar is displayed in the toolbar. This shows when the current task(s) will be completed. A small *interrupt* button can be clicked on to interrupt all background tasks. Clicking on the progress bar will open the *Tasks* view (*The Task Manager*).

1.6 The omni-search



The final item in the toolbar is the omni-search. This is a search field that will search the text you type in various contexts in GPS, like filenames (for convenient access to the source files), the entities referenced in your application, your code,...

There are various ways to use the omni-search:

- The simplest is of course to click in it, and type the pattern you are interested in. GPS will immediately start searching in the background for possible matching open windows, file names, entities, GPS actions, bookmarks, and source files. For each context, GPS only displays the five matches with the highest score.

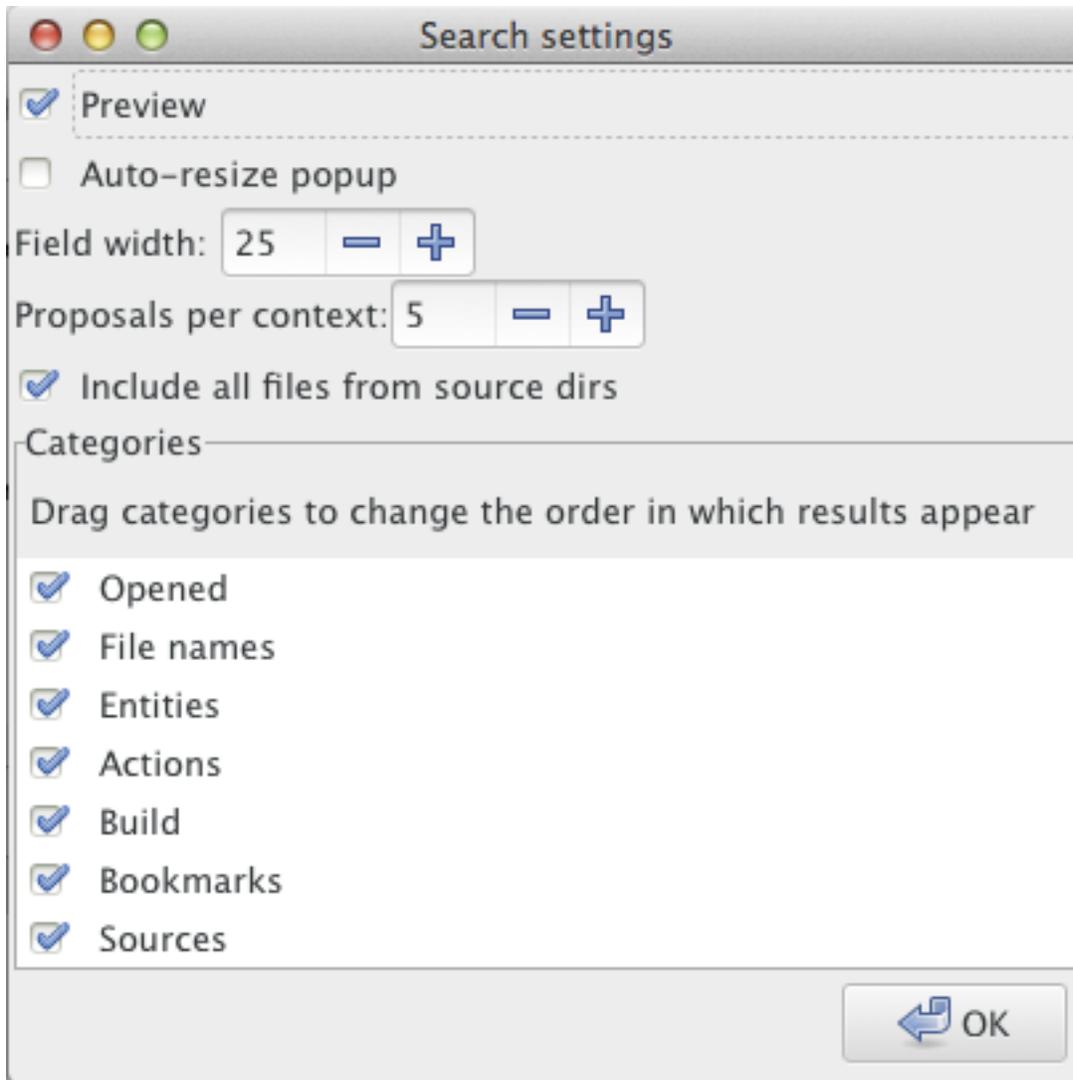
For each context, GPS tells you how many matches there. You can click on the name of the context to only search in that context. So for instance, if GPS tells you there are 20 file names matching your search (but only displaying the five first ones), you can click on *file names* to view all 20 names, and exclude the results from all the other contexts.

If you click again on the context, GPS is back to displaying the results in all contexts.

- If you are searching in a specific context, the above requires too many clicks. GPS defines a number of actions to which you can bind key shortcuts via the *Edit* → *Key Shortcuts* dialog. These actions are found in *Search* category, and are called *Global Search in context:*. GPS includes a menu for two of them by default: *File* → *Open From Project...* will search amongst filenames, whereas *Navigate* → *Goto Entity...* will search amongst all entities defined in your project.

Each context displays its results slightly differently, and clicking on a result will have different effects depending on a context. For instance, clicking on a file name will open the corresponding file, whereas clicking on an entity will jump to its declaration, and clicking on a bookmark will show the source file it is in.

Pressing `enter` at any point will select the top item in the list of search results, which is in general faster than clicking on it.



It is possible that you have no interest in some of the search contexts. You can choose to disable some of them by clicking on the *Settings* icon at the bottom-right corner of the completion popup. The resulting dialog shows you the list of all contexts that are searched, and clicking on any of the checkboxes next to the names will enable to disable the context. Note that this list is only displayed when you have accessed the omni-search by clicking directly into it. If you accessed it via `shift-F3` or the equivalent menu *File* → *Open From Project...*, then only a subset of the settings will be displayed.

Still in this settings dialog, you can also reorder the context. This influences both the order in which they are searched and the order in which they are displayed. We recommend keeping the *Sources* context last, because it is the slowest, and while GPS is searching it, it would not be able to search the other faster contexts.

In the settings dialog, you can choose whether to display a *Preview* for the matches. This preview is displayed when you use the `down` arrow key to select some of the search results. In general, it will display the corresponding source file, or the details for the matching GPS action or bookmark.

The settings dialog also allows you to select the number of results that should be displayed for each context when multiple contexts are displayed, or the size of the search field (which depends on how big your screen and the GPS window are).

One of the search context looks at file names, and is convenient for quickly opening files. By default, it will look at all files found in any of the source directories of your project, even if those files are not explicit sources of the project

(for instance because they do not match the naming scheme for any of the languages used by the project). This is often convenient because you can easily open support files like Makefiles or documentation, but it can also sometimes get in the way if the source directories include too many irrelevant files. The *Include all files from source dirs* setting can be used to control this behavior.

GPS proposes various algorithms to do the search:

- *Full Text* simply checks whether the text you typed appears exactly as is in the context (be it a file name, the contents of a file, the name of an entity,...)
- *Regular Expression* assumes the text you typed is a valid regular expression, and searches for it. If this isn't a valid regexp, it tries to search for the exact text.
- *Fuzzy Match* will try to find each of the characters you typed, in that order, but possibly with extra characters in between. This is likely to be the fastest way to search, but it might require a bit of getting used to. For instance, the text 'mykps' will match the file name 'MY_PacKage.adS' because the letters shown in upper cases match the text.

When searching in the source files, the algorithm is changed slightly, since otherwise there would obviously be too many matches. In that context, GPS only allows a few approximations between the text you typed and the text it tries to match (one or two extra characters or missing characters, for instance).

You can select the algorithm you wish to use by changing it at the bottom of the popup window that contains the search results.

Once it has found a match, GPS assigns it a score, so that it can order the results in the most meaningful way for you. Scoring is based on a number of criteria:

- length of the match

For instance, when searching file names, it is more likely that by typing 'foo' you intended to match 'foo.ads' rather than 'the_long_foo.ads'.

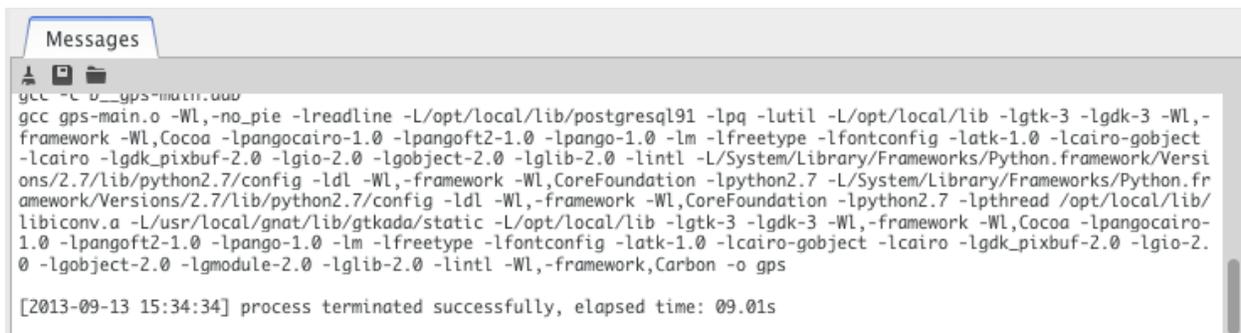
- the grouping of characters in match

As we have seen, when doing a fuzzy match, GPS allows extra characters in between the ones you typed. But the closer the ones you typed are in the match result, the more likely it is this is what you were looking for.

- when was the item last selected

If you recently selected an item (like a file name), GPS assumes you are more likely to want it again, and will raise its score appropriately.

1.7 The Messages window



```

Messages
gcc -c _o_gps-main.o
gcc gps-main.o -Wl,-no_pie -lreadline -L/opt/local/lib/postgresql91 -lpq -lutil -L/opt/local/lib -lgtk-3 -lgdk-3 -Wl,-framework -Wl,Cocoa -lpangocairo-1.0 -lpangoft2-1.0 -lpango-1.0 -lm -lfreetype -lfontconfig -latk-1.0 -lcairo-gobject -lcairo -lgdk_pixbuf-2.0 -lgio-2.0 -lgobject-2.0 -glib-2.0 -lintl -L/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/config -ldl -Wl,-framework -Wl,CoreFoundation -lpython2.7 -L/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/config -ldl -Wl,-framework -Wl,CoreFoundation -lpython2.7 -lpthread /opt/local/lib/libiconv.a -L/usr/local/gnat/lib/gtkada/static -L/opt/local/lib -lgtk-3 -lgdk-3 -Wl,-framework -Wl,Cocoa -lpangocairo-1.0 -lpangoft2-1.0 -lpango-1.0 -lm -lfreetype -lfontconfig -latk-1.0 -lcairo-gobject -lcairo -lgdk_pixbuf-2.0 -lgio-2.0 -lgobject-2.0 -lgmodule-2.0 -glib-2.0 -lintl -Wl,-framework,Carbon -o gps

[2013-09-13 15:34:34] process terminated successfully, elapsed time: 09.01s

```

The Messages window is used by GPS to display information and feedback about operations, such as build output, information about processes launched, error messages.

This is a read-only window, which means that only output is available, no input is possible.

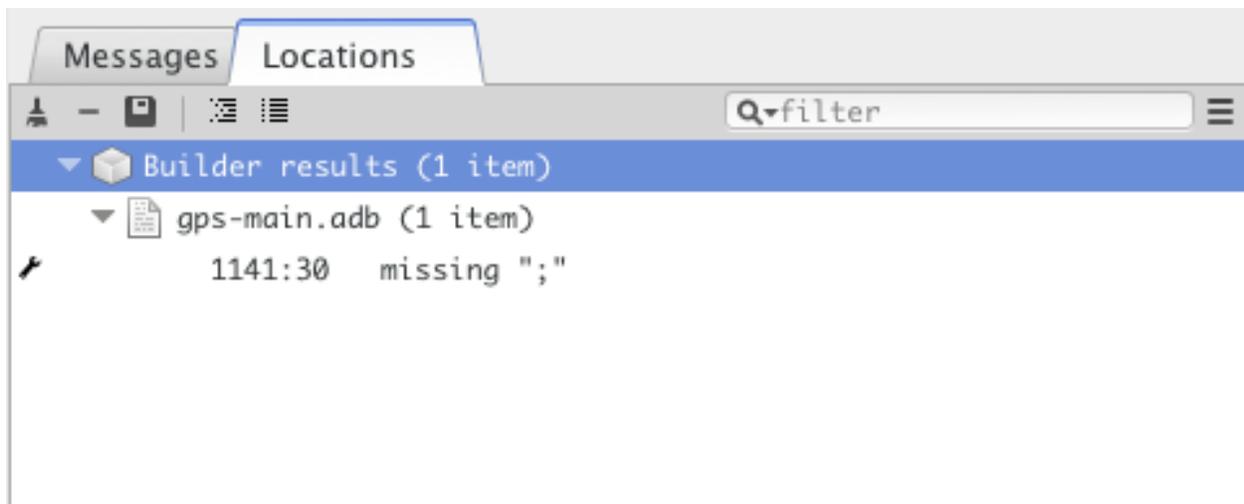
Its local toolbar contains buttons to *Clear* the contents of the window, as well as to *Save* and *Load* from files.

In general the output of the compilation is displayed in the *Messages* window, but will also be parsed and displayed more conveniently in the *Locations* window (*The Locations view*).

When a compilation finishes, GPS also displays the total elapsed time. If the process ended with errors, GPS will display the total progress (as is also displayed in the progress bar in the GPS toolbar), which is convenient to see how many files were compiled successfully.

The *Messages* window can not be closed, because it might contain important messages at any time. However, it might happen that it has been closed anyway, and in this case it can be reopened with the *Tools* → *Views* → *Messages* menu.

1.8 The *Locations* view



The *Location* window is used whenever GPS needs to display a list of locations in the source files (typically, when performing a global search, or displaying compilation results).

The *Locations* shows a hierarchy of categories, which contain files, which contain messages at specific locations. The category describes the type of messages (search results, build results,...). Clicking on a location item will bring up a file editor at the requested place.

Placing the mouse over an item automatically pop up a tooltip window with full text of the message if this text can't be completely shown in the window.

In general, each message in this window is associated with a special full line highlighting in the corresponding source editor, as well as a mark on the left side of editors to visually navigate between these locations.

The *Locations* view provides a local toolbar with the following buttons:

- *Clear* will remove all entries from the window. Depending on your settings, this might also close the window.
- *Remove* will remove the currently selected category, file or message. This of course removes the corresponding highlighting in the source editor.
- *Save* can be used to save the contents of the window to a text file, for later reference. This text file can not be imported by GPS into the locations view later. If you want to reload the contents of the locations (in the case of build errors, for instance), it is better to save and load the contents of the *Messages* window.
- *Expand All* and *Collapse All* can be used to quickly show or hide all messages in this window.

- a filter that can be used to show or hide some of the messages. Filtering is done on the text of the message itself (either as a text or as a regular expression). It can also be reversed, so that for instance typing *warning* in the filter field and reversing the filter will hide warning messages

The local settings menu contains the following entries:

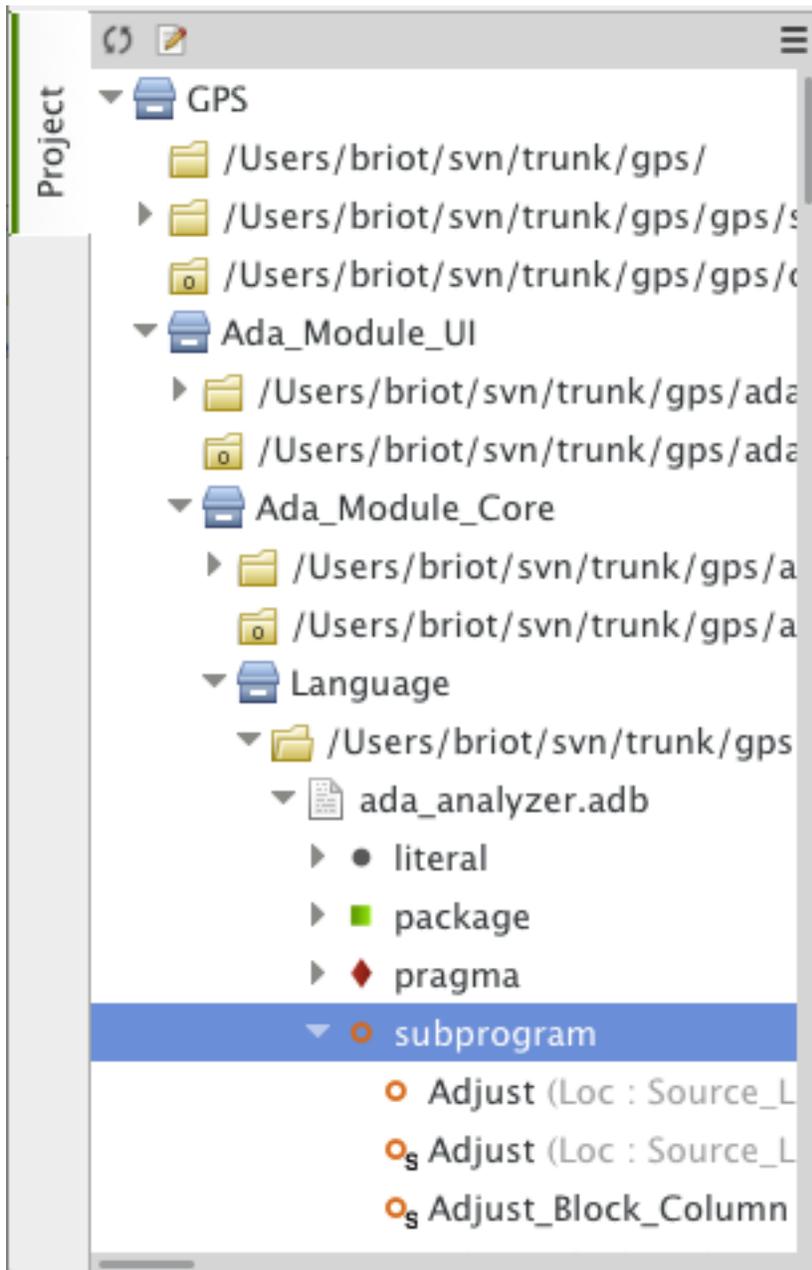
- *Sort by subcategory* Toggle the sorting of the entries by sub-categories. This is useful, for example, for separating the warnings from the errors in the build results. The error messages will appear first. The default is to sort the message by their location.
- *Sort files alphabetically* Force files to be sorted alphabetically. The default is that files are not sorted, which makes manipulation of the *Locations* window easier before all messages are added to it (otherwise the nodes might be switched while you are trying to click on them).
- *Jump to first location*: Every time a new category is created, as a result of a compilation or a search operation for example, the first entry of that category is automatically selected, and the corresponding editor opened.
- *Warp around on next/previous* controls the behavior of the *Previous tag* and *Next tag* menus (see below).
- *Auto close locations* will automatically close this window when it becomes empty.
- *Save locations on exit* controls whether GPS should save and restore the contents of this window between sessions. The loaded contents might not apply the next time, because for instance the source files have changed, or build errors have been fixed, so it might be an inconvenience to automatically reload the messages.

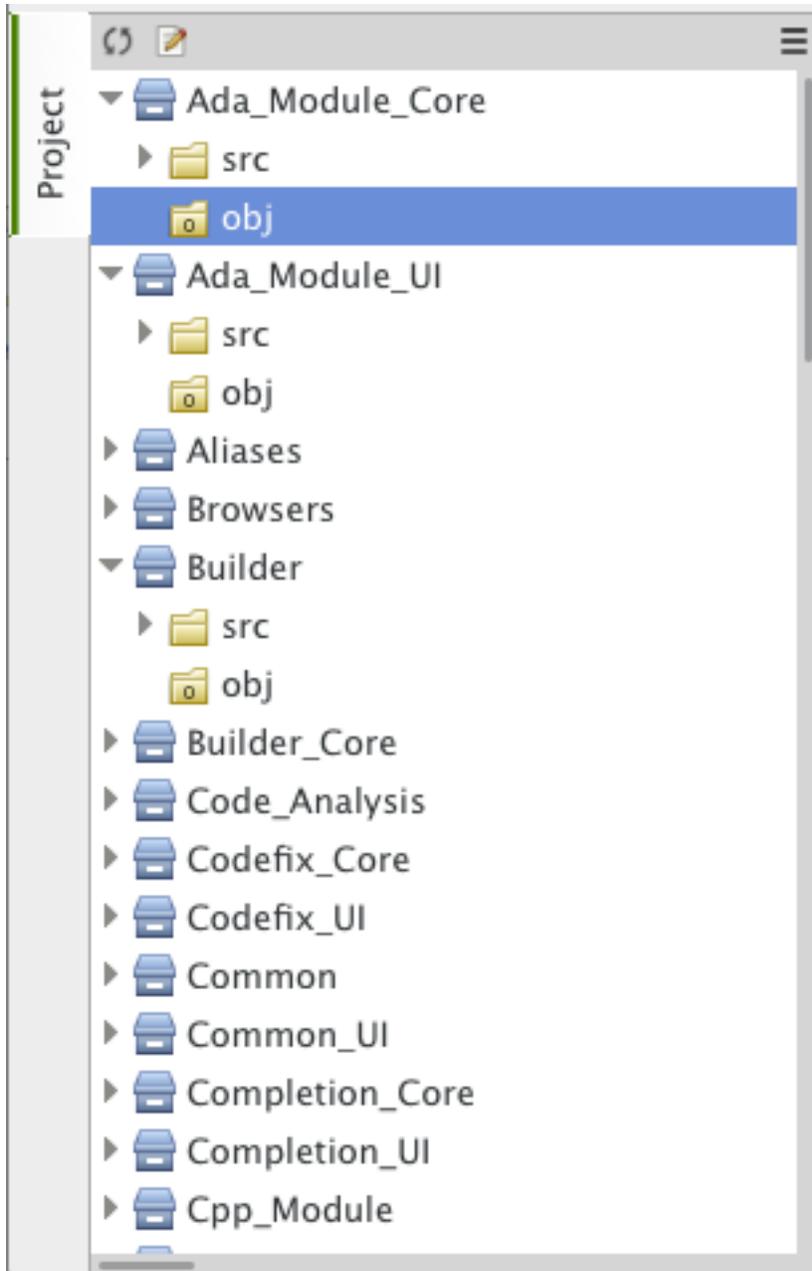
To navigate through the locations with the keyboard, GPS provides two menus: *Navigate* → *Previous Tag* and *Navigate* → *Next Tag*. Depending on your settings, they might wrap around after reaching the first or last message.

It is also possible to bind key shortcuts to these menus via the *Edit* → *Key Shortcuts* menu.

In some cases, a wrench icon will be visible on the left of a compilation message. See [Code Fixing](#) for more information on how to take advantage of this icon.

1.9 The *Project* view





The project view provides a representation of the various components of your project, as listed below. It is displayed by default on the left side of the workspace, and can be selected by using the *Project* → *Project View* or *Tools* → *Views* → *Project* menu items.

On Windows, it is possible to drop files (coming for instance from the Windows Explorer) directly in the project view. If you drop a project file, it will be loaded by GPS and replace the current project; if you drop a source file, it is opened in a new editor.

The project view, as well as the file and outline view provide an interactive search capability allowing you to quickly search in the information currently displayed. Just start typing the text to search when the view has the focus. Note however, that the contents of the *Project* view is computed lazily, so not all files are known to this search capability.

This will open a small window at the bottom of the view where you can interactively type names. The first matching name in the tree will be selected while you type it. You can then also use the up and down keys to navigate through all the items matching the current text.

The various components that are displayed are:

projects

All the sources you are working with are put under control of projects. These projects are a way to store the switches to use for the various tools, as well as a number of other properties like the naming schemes for the sources. They can be organized into a project hierarchy, where a root project can import other projects, each with their own set of sources (see *The Welcome Dialog* on how projects are loaded in GPS).

The *Project* view displays this project hierarchy: the top node is the root project of your application (generally, this is where the source file that contains the main subprogram will be located). Then a node is displayed for each imported project, and recursively for their own imported projects.

A given project might appear multiple times in the view, if it is imported by several other projects.

Likewise, if you have edited the project manually and have used the `limited with` construct to have cycles in the project dependencies, the cycle will expand infinitely. For instance, if project *a* imports project *b*, which in turns imports project *a* through a `limited with` clause, then expanding the node for *a* will show *b*. In turn, expanding the node for *b* will show a node for *a*, and so on.

A special icon with a pen mark is displayed if the project was modified, but not saved yet. You can choose to save it at any time by right-clicking on it. GPS will remind you to save it before any compilation, or save it automatically, if the corresponding preference is saved.

There exists a second display for this project view, which lists all projects with no hierarchy: all projects appear only once in the view, at the top level. This display might be useful for deep project hierarchies, to make it easier to find projects in the project view. This display is activated through the local settings menu to the right of the *Project* view toolbar.

directories

The files in a project are organized into several physical directories on the disk. These directories are displayed under each project node in the *Project* view

You can chose whether you want to see the absolute path names for the directories or paths relative to the location of the project. This is done using the local settings menu *Show absolute paths* of the *Project* view. In all cases, the tooltip that is displayed when the mouse hovers a file or directory will show the full path.

Special nodes are created for object and executables directories. No files are shown for these.

The local setting *Show hidden directories* can be used to filter the directories considered as hidden. This can be used to hide the version control directories like `CVS` or `.svn` for example.

files

The source files themselves are contained in the directories, and displayed under the corresponding nodes. Note that only the source files that actually belong to the project (i.e. are written in a language supported by that project and that follow its naming scheme) are actually visible. For more information on supported languages, see *Supported Languages*.

A given file might appear multiple times in the *Project* view, if the project it belongs to is imported by several other projects.

You can also drag a file anywhere into GPS. This will open a new editor if the file is not already edited, or move the existing editor otherwise. If you press `shift` at the same time, and the file is already edited, a new view of the existing editor is created instead.

entities

If you open the node for a source file, the file is parsed by one of the fast parsers integrated in GPS so that all entities declared in the file can be shown. These entities are grouped into various categories, which depend on the language. Typical categories include subprograms, packages, types, variables, tasks, ...

Double-clicking on a file, or simple clicking on any entity will open a source editor and display respectively the first line in this file or the line on which the entity is defined.

If you open the search dialog through the *Navigate* → *Find or Replace...* menu, you have the possibility to search for anything in the *Project* view, either a file or an entity. Note that searching for an entity can be slow if you have lots of files, and/or big files.

A contextual menu, named *Locate in Project View*, is also provided in source editors. This will automatically search for the first entry for this file in the *Project* view. This contextual menu is also available in other modules, e.g. when selecting a file in the *Dependency Browser*.

The local toolbar of the *Project* view contains a convenient button to reload the project. This is useful when you have created or removed source files from other applications, and want to let GPS know that there might have been changed on the file system that impact the contents of the current project.

It also includes a button to graphically edit the attributes of the selected project, like the tool switches, the naming schemes,... It behaves similarly to the *Project* → *Edit Project Properties* menu. See *The Project Properties Editor* for more information.

If you right click on a project node, a contextual menu appears which contains, among others, the following entries that are useful to understand or modify your project:

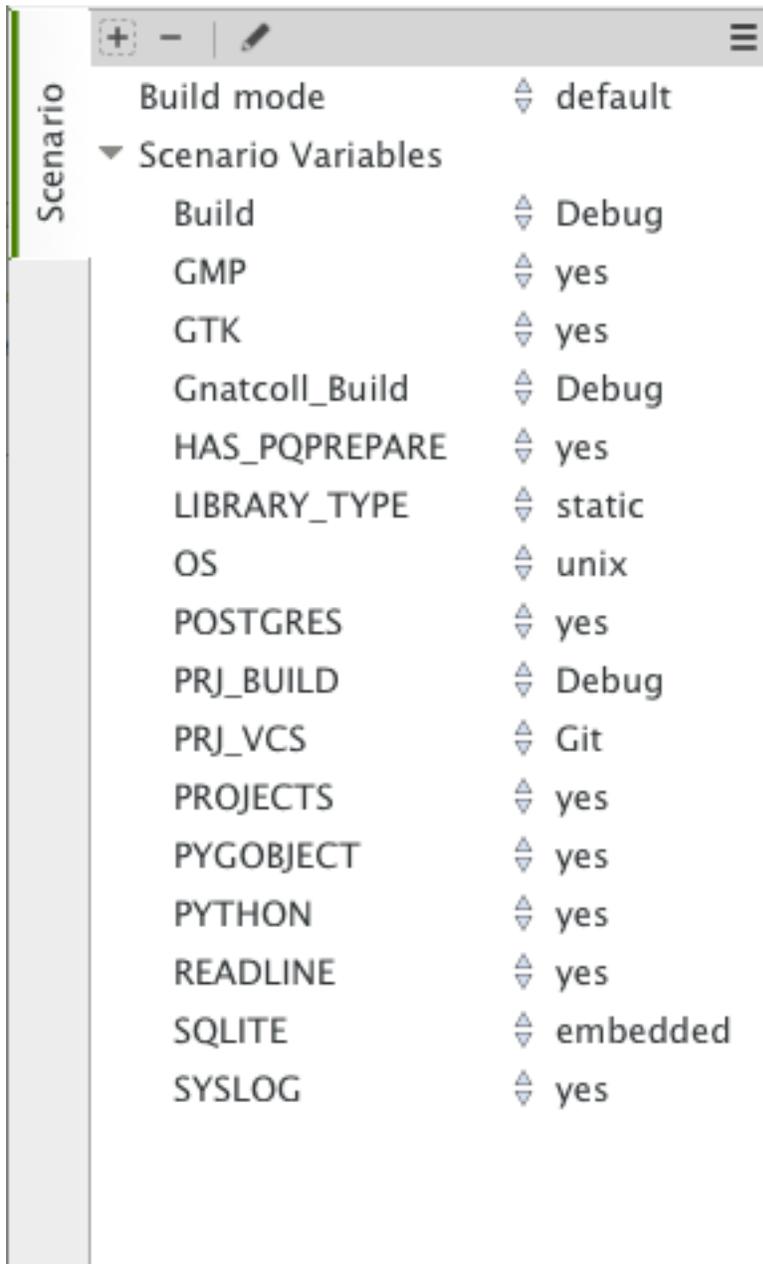
- *Show projects imported by...*
- *Show projects depending on...* These two menus will open a new window, the *Project browser*, which displays graphically the relationships between each project in the hierarchy (see *The Project Browser*).
- *Project* → *Properties* This menu opens a new dialog to interactively edit the attributes of the project (tool switches, naming schemes,...) and is similar to the local toolbar button.
- *Project* → *Save project...* This item can be selected to save a single project in the hierarchy after it was modified. Modified but unsaved projects in the hierarchy have a special icon (a pen mark is drawn on top of the standard icon). If you would rather save all the modified projects in a single step, use the menu bar item *Project* → *Save All*.

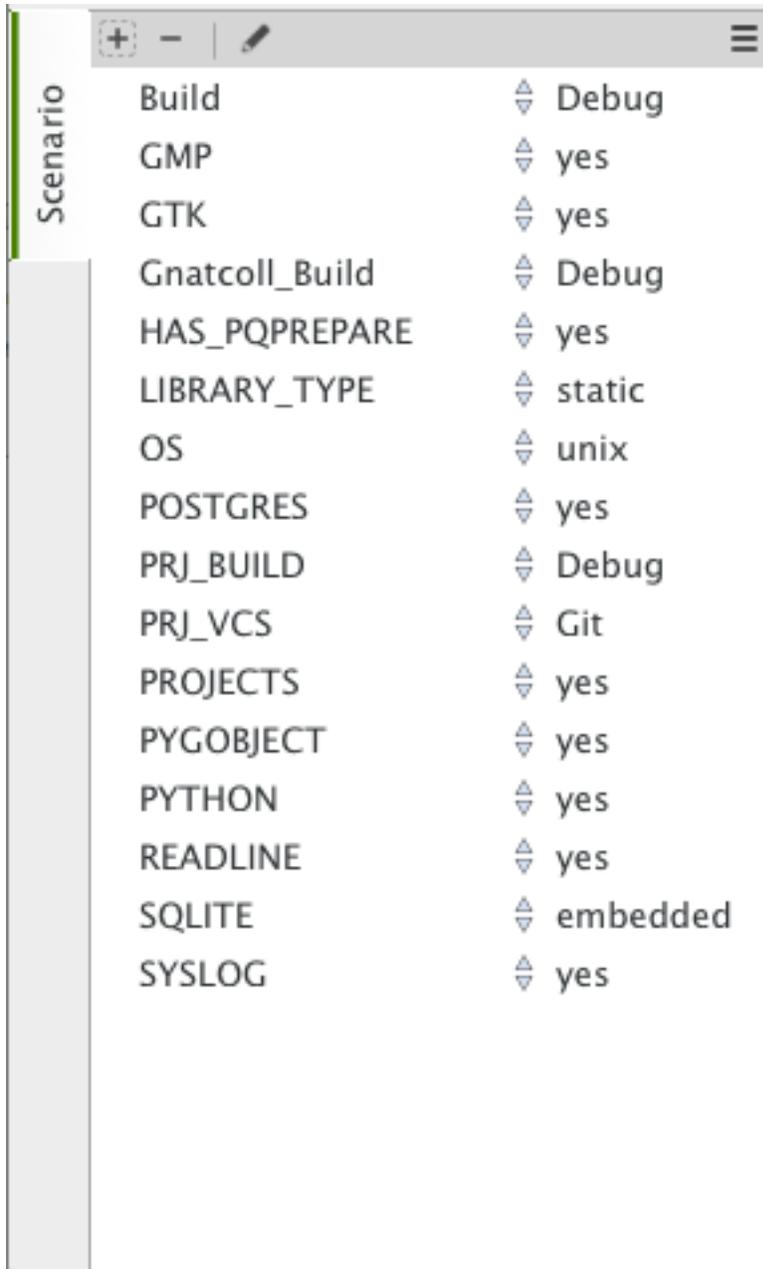
Any time one or several projects are modified, the contents of the project view is automatically refreshed. No project is automatically saved. This provides a simple way to temporarily test new values for the project attributes. Unsaved modified projects are shown with a special icon in the project view, displaying a pen mark on top of the standard icon:



- *Project* → *Edit source file* This menu will load the project file into an editor, so that you can manually edit it. This should be used if you need to access some features of the project files that are not accessible graphically (renames statements, variables, ...).
- *Project* → *Dependencies* This menu opens the dependencies editor for the selected project (*The Project Dependencies Editor*).
- *Project* → *Add scenario variable* This menu item should be used to add new scenario variables to the project (see *Scenarios and Configuration Variables*). It might be more convenient in general to use the *Scenario* view for that purpose.

1.10 The *Scenario* view





As described in the GNAT User's Guide, the project files can be configured through external variables (typically environment variables). This means that e.g. the exact list of source files, or the exact switches used to compile the application can be changed when the value of these external variables is changed.

GPS provides a simple access to these variables, through a window called the *Scenario View*. These variables are called *Scenario Variables*, since they provide various scenarios for the same set of project files.

Each such variable is listed on its own line, along with its current value. You can change the current value by clicking on it, and then selecting the new value among the valid ones that pop up. GPS does not remember the current value from one session to the next. Instead, the variables' initial values come from the project files themselves (where a default value can be specified) or from the environment in which GPS is started, just as is the case when spawning command line tools like **gprbuild**.

Whenever you change the value of one of the variables, the project is automatically recomputed, and the list of source files or directories is changed dynamically to reflect the new status of the project. Starting a new compilation at that

point will use the new switches, and all the aspects of GPS are immediately affected according to the new setup.

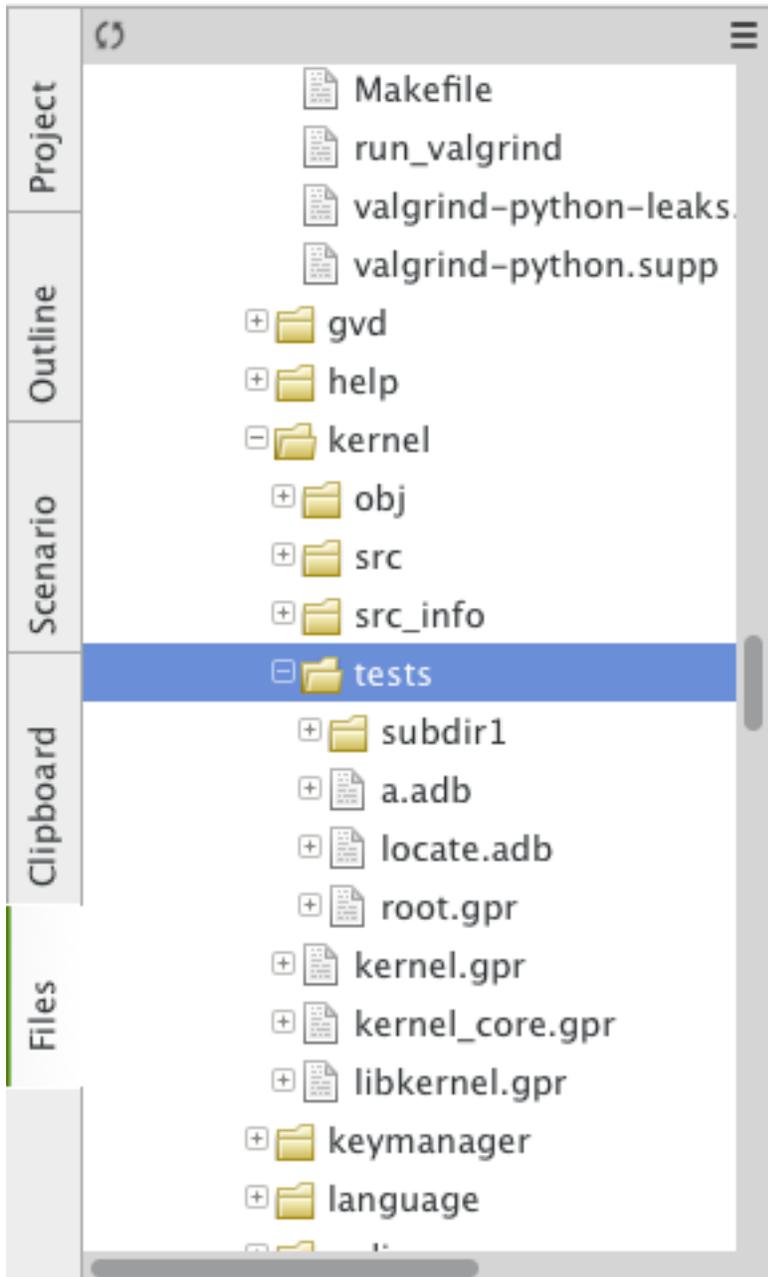
New scenario variables can be created by selecting the + icon in the local toolbar of the *Scenario* view. You can also edit the list of possible values for a variable by clicking on the *edit* button in that toolbar, and of course delete an existing variable by clicking on the - button.

Note that any of these changes impacts the actual project file (.gpr), so you might not want to do that if the project file was written manually (the impacts can be significant).

The first line in the *Scenario* view is the current mode. This impacts various aspects of the build, including compiler switches and object directories (see *The Build Mode*). As for scenario variables, the mode can be changed by clicking on the value and selecting a new value in the popup window.

If you are not using build modes and want to save some space on the screen, you can use the local settings menu *Show build modes* to disable the display.

1.11 The *Files* View

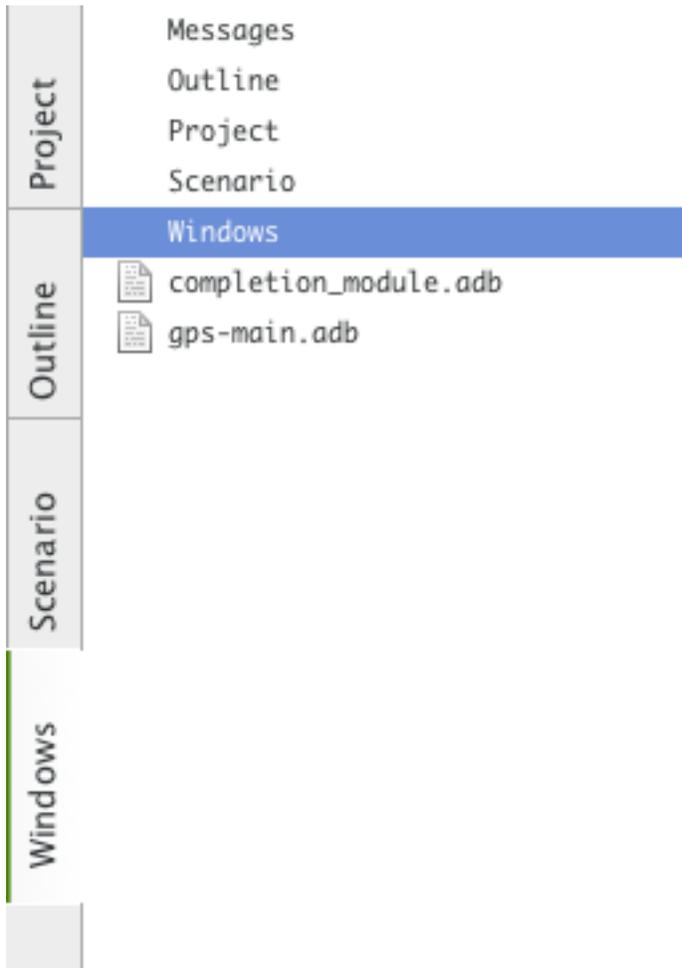


In addition to the *Project* view, GPS also provides a *Files* view through the *Tools* → *Views* → *Files* menu.

In this view, directories are displayed exactly as they are organized physically on the disk (including Windows drives). Each source file can also be explored as described in *The Project view*. Drag and drop of files is also possible from the files view, to conveniently open a file.

By default, the *Files* view will display all the files that exist on the disk. Filters can be set through the local settings menu to restrict the display to the files and directories that belong to the project (use the *Show files from project only* menu).

1.12 The *Windows* view





The *Windows* view displays the currently opened windows. It is opened through the *Tools* → *Views* → *Windows* menu.

In the contextual menu, you can configure the display in one of two ways:

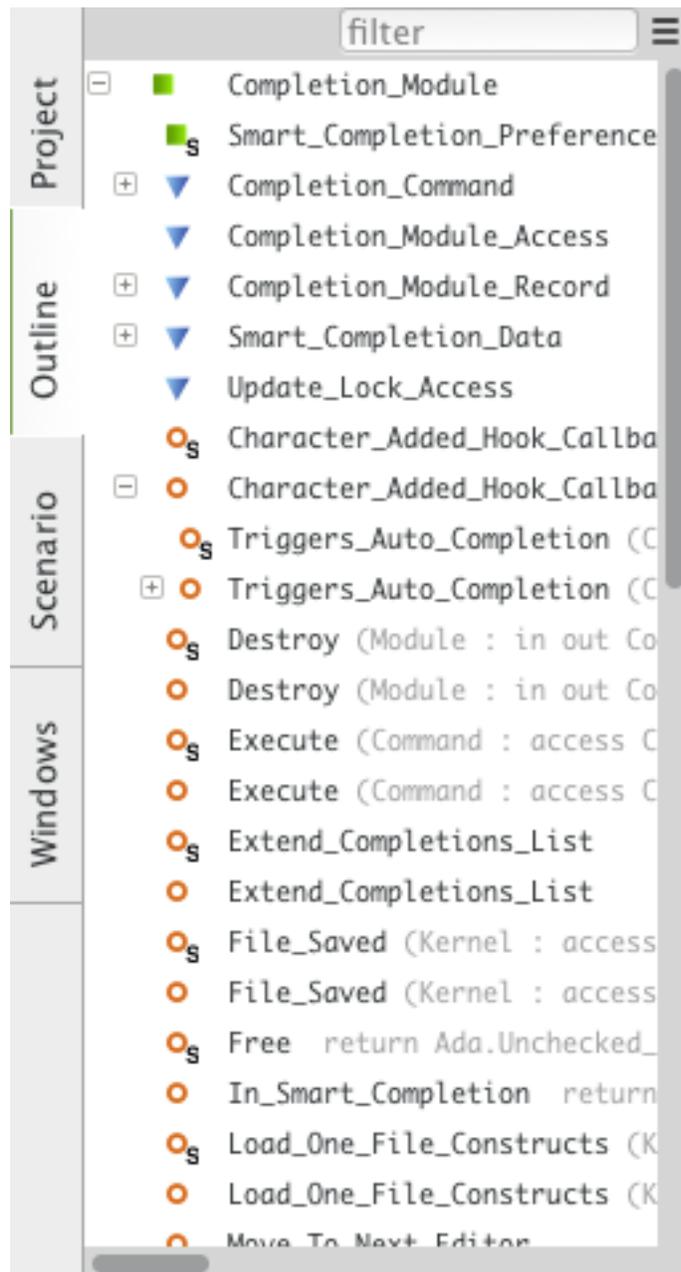
- Sorted alphabetically
- Organized by notebooks, as in the GPS window itself. This view is mostly useful if you have lots of opened windows.

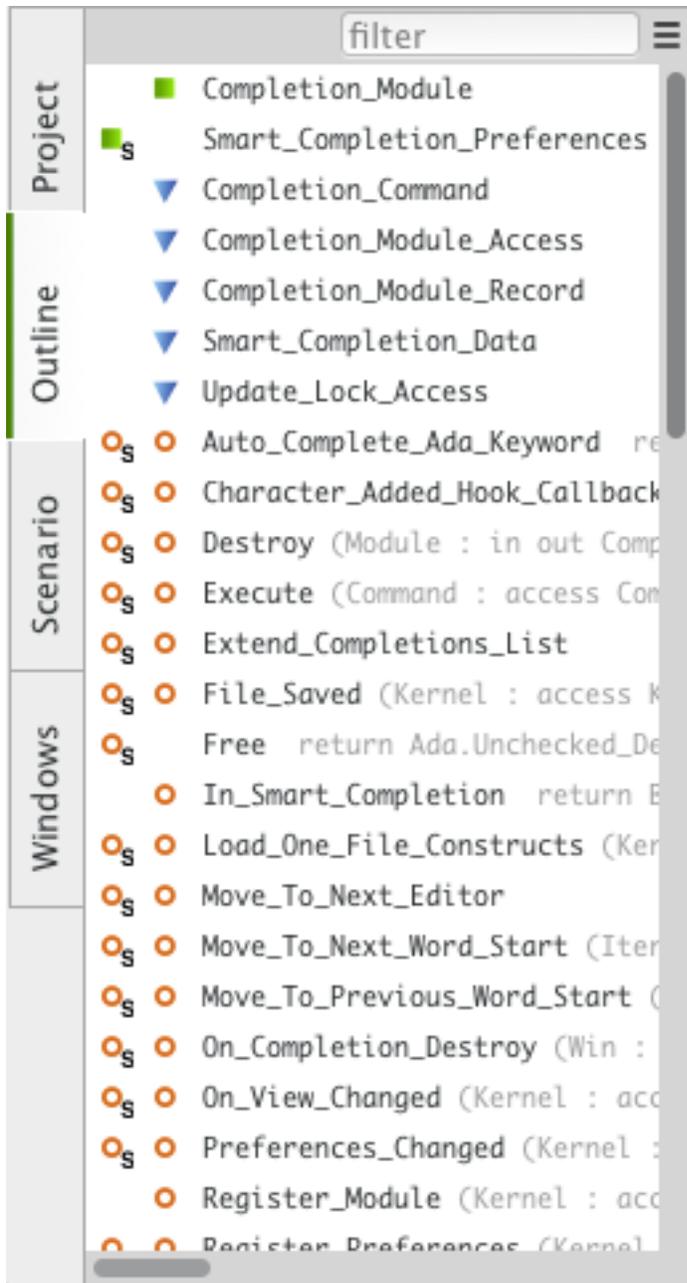
You can also choose, through the contextual menu, whether only the source editors should be visible, or whether all windows should be displayed.

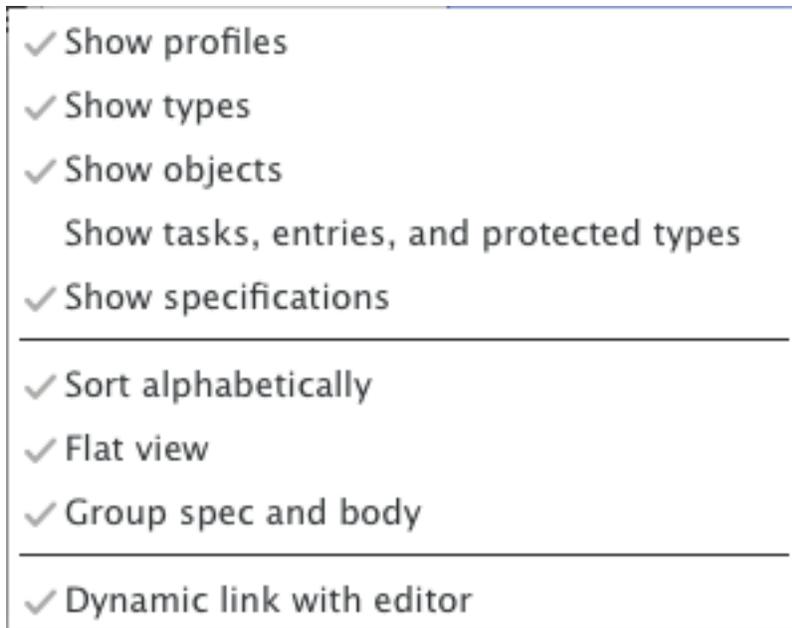
This view allows you to quickly select and focus on a particular window, by clicking on the corresponding line with the left mouse button. If you click and leave the mouse button pressed, this starts a drag and drop operation so that you can also move the window to some other place in the desktop (see the description of the *Multiple Document Interface*)

Multiple windows can be selected by clicking with the mouse while pressing the control or shift keys. The Window view provides a contextual menu to easily close all selected windows at once, which is a very fast way to cleanup your desktop after you have finished working on a task.

1.13 The *Outline* view







The *Outline* view, which you can choose to activate through the *Tools* → *Views* → *Outline* menu, shows the contents of the current file.

The exact semantics depends on the language you are seeing. For Ada, C and C++ files, this is the list of entities that are declared at the global level in your current file (Ada packages, C++ classes, subprograms, Ada types, ...).

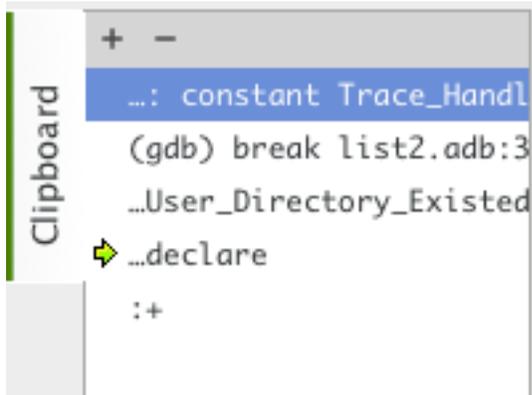
The contents of this view is refreshed every time the current editor is modified.

Clicking on any entity in this view will automatically jump to the right line in the file (either to the spec or the body).

The local settings menu contains multiple check boxes that alter the display of the outline view:

- *Show profiles* indicates whether the list of parameters of the subprograms should be displayed. This is in particular useful for languages that allow overriding of entities.
- *Show types*, *Show objects*, *Show tasks*, *entries and protected types* controls the display of specific categories of entities.
- *Show specifications* indicates whether GPS should display a line for the specification (declaration) of entities, in addition to the location of their bodies.
- *Sort alphabetically* controls the order in which the entities are displayed (either alphabetically or in the same order as in the source file)
- *Flat View* controls whether the entities are always displayed at the top level of the outline view. When this is disabled, nested subprograms are displayed below the subprogram in the scope of which they are declared.
- *Group spec and body* can be enabled to display up to two icons on each line (one for the spec, one for the body in case both occur in the file). You can then click directly on one or the other icon to go directly to that location. If you click on the name of the entity you are first taken to its declaration, unless this is already the current location in the editor in which case you are moved to the body.
- *Dynamic link with editor*: If this option is set, the current subprogram will be selected in the outline view every time the cursor position changes in the current editor. This option requires some computation for GPS, and you might want to avoid the slow down by disabling it.

1.14 The *Clipboard* view



GPS has an advanced mechanism for handling copy/paste operations.

When you select the menus *Edit* → *Copy* or *Edit* → *Cut*, GPS adds the current selection to the clipboard. As opposed to what lots of applications do, it doesn't discard the previous contents of the clipboard, but save it for future usage. It saves a number of entries this way, up to 10 by default. This value is configurable through the *Clipboard Size* preference.

When you select the menu *Edit* → *Paste*, GPS will paste the last entry made in the clipboard at the current location in the editor. If you immediately select *Edit* → *Paste Previous*, this newly inserted text will be removed, and GPS will instead insert the second to last entry added to the clipboard. You can keep selecting the same menu to get access to older entries.

This is a very powerful mechanism, since it means you can copy several distinct lines from a place in an editor, move to an other editor and paste all these separate lines, without having to go back and forth between the two editors.

The *Clipboard* view provides a graphical mean of seeing what is currently stored in the clipboard. It can be opened via *Tools* → *Views* → *Clipboard*.

It appears as a list of lines, each of which is associated with one level of the clipboard. The text that shows in these lines is the first line of the selection at that level that contains non blank characters. Leading characters are discarded. [...] is prepended or appended in case the selection has been truncated.

If you bring the mouse over a line in the *Clipboard* view, a tooltip will pop up showing the entire selection corresponding to the line by opposition to the possibly truncated one.

In addition, one of the lines has an arrow on its left. This indicates the line that will be pasted when you select the menu *Edit* → *Paste*. If you select instead the menu *Edit* → *Paste Previous*, then the line below that one will be inserted instead.

If you double-click on any of these lines, GPS will insert the corresponding text in the current editor, and make the line you clicked on the current line, so that selecting *Edit* → *Paste* or the equivalent shortcut will now insert that line.

The local toolbar in the clipboard view provides two buttons:

- **Append To Previous.** If you select this button, the select line will be append to the one below, and removed from the clipboard. This means that selection *Edit* → *Paste* will in fact paste the two entries at the same time. This is in particular useful when you want to copy lines from separate places in the initial file, merge them, and then paste them together one or more times later on, through a single operation.
- **Remove.** If you select this button, the selected line is removed from the clipboard.

The Clipboard View content is preserved between GPS sessions. As an exception, huge entries are removed and replaced with an entry saying “[Big entry has been removed]”.

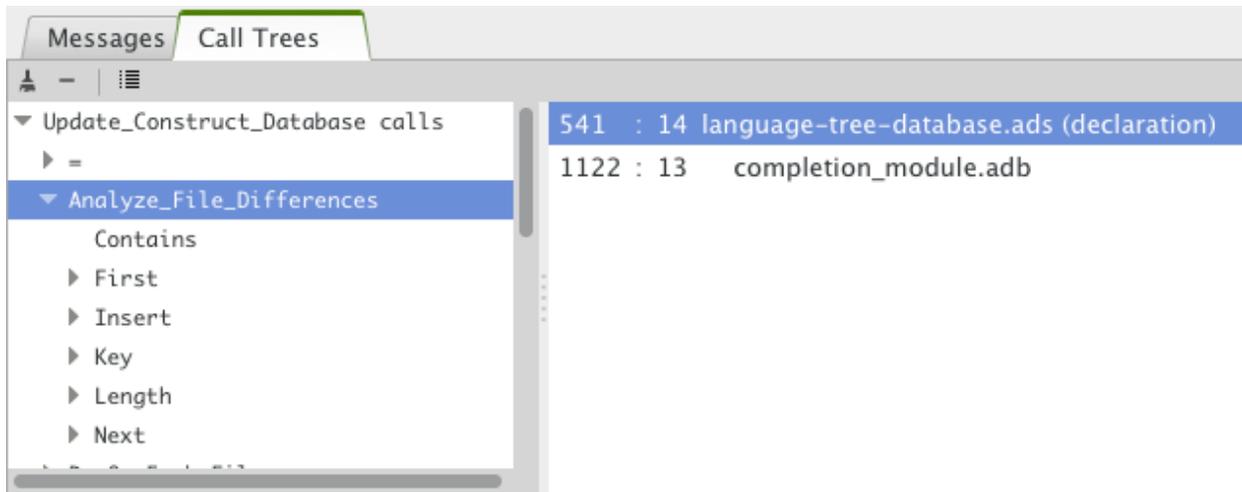
1.15 The *Call trees* view and *Callgraph* browser

These two views play a similar role. They display the same information about entities, but in two different ways: the callgraph view displays the information in a tree, easily navigable and perhaps easier to manipulate when lots of entities are involved; the callgraph browser displays the information as graphical boxes that can be manipulated on the screen, and is best suited to generate a diagram that can be later exported to your own documents.

These views are used to display the information about what subprograms are called by a given entity, and, opposite, what entities are calling a given entity.

Some references might be reported with an additional ” (dispatching)” text. In such a case, this indicates that the call to the entity is not explicit in the sources, but could occur through dynamic dispatching. This of course depends on what arguments are passed to the caller at run time, and it is possible that the subprogram is in fact never dispatched to.

1.15.1 Call Trees



The *Call trees* are displayed when you select one of the contextual menus `<entity> calls` and `<entity> is called by`. Every time you select one of these menus, a new view is opened to display that entity.

Whenever you expand a node from the tree by clicking on the small expander arrow on the left of the line, further callgraph information is computed for the selected entity, which makes it very easy to get information for a full callgraph tree.

Closing and expanding a node again will recompute the callgraph for the entity.

On the right side of the main tree, a list displays the locations of calls for the selected entity. Clicking on entries in this list opens editors showing the corresponding location.

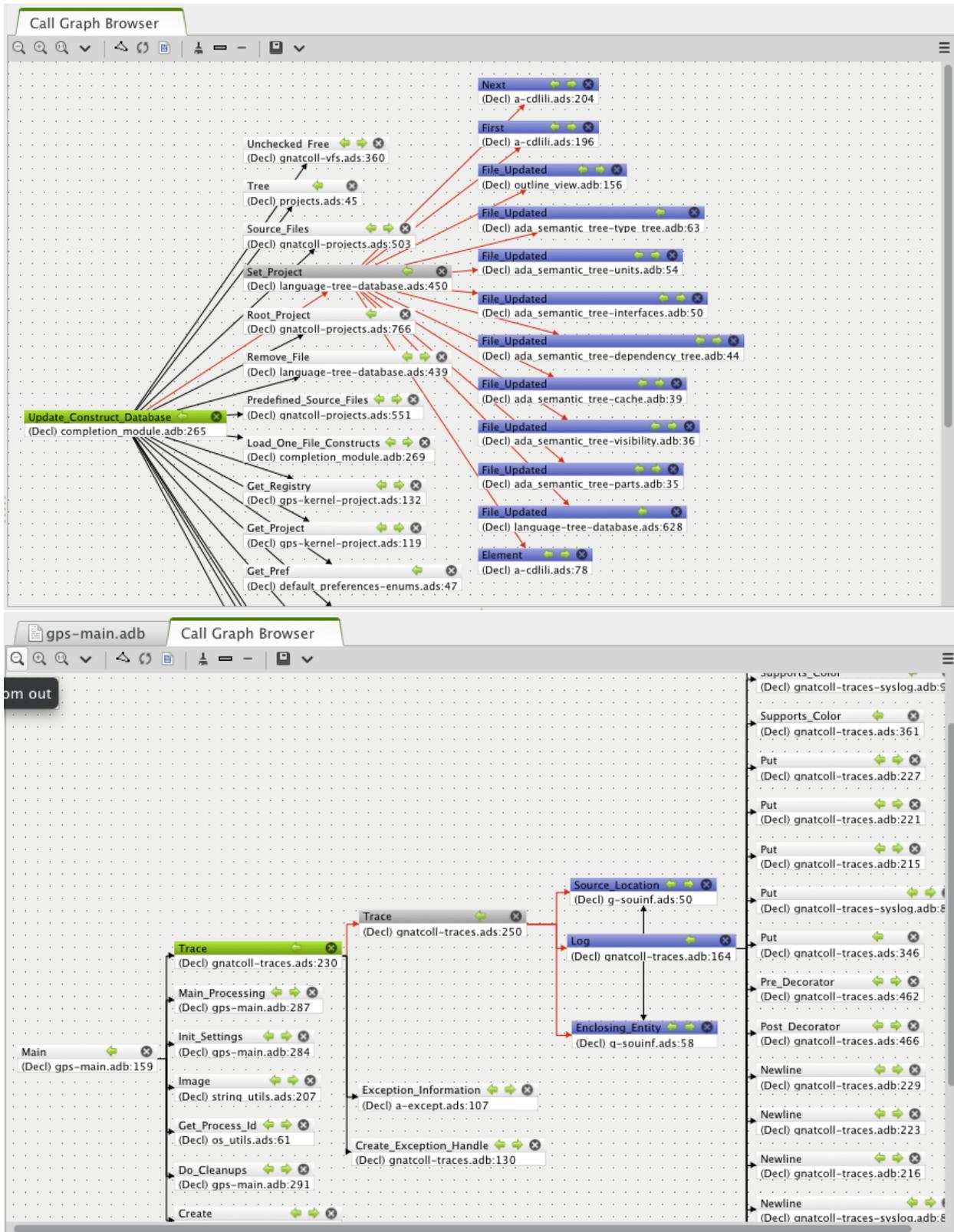
The *Calltree* supports keyboard navigation: `Up` and `Down` keys navigate between listed locations, `Left` collapses the current level, `Right` expands the current level, and `Return` jumps to the currently selected location.

The contents of the calltree is not restored the next time GPS is restarted, because its contents might be misleading if the sources have changed in-between, and GPS would be wasting time loading the information again.

The local toolbar provides the following buttons:

- *Clear* Remove all entries from the Callgraph View.
- *Remove entity* Remove the selected entity from the Callgraph View.
- *Collapse all* Collapse all the entities in the Callgraph View.

1.15.2 Callgraph browser



The callgraph shows graphically the relationship between subprogram callers and callees. A link between two items indicate that one of them is calling the other.

A special handling is provided for renaming entities (in Ada): if a subprogram is a renaming of another one, both items will be displayed in the browser, with a special hashed link between the two. Since the renaming subprogram doesn't have a proper body, you will then need to ask for the subprograms called by the renamed to get the list.

In this browser, clicking on the right arrow in the title bar will display all the entities that are called by the selected item.

Clicking on the left arrow will display all the entities that call the selected item (i.e. its callers).

This browser is generally opened by right-clicking on the name of an entity in source editors or *Project* view, and selecting one of *Browsers* → *<entity> calls*, *Browsers* → *<entity> calls (recursive)* or *Browsers* → *<entity> is called by*.

All boxes in this browser list several information: the location of their declaration, and the list of all their references in the other entities currently displayed in the browser. If you close the box for an entity that calls them, the matching references are also hidden, to keep the contents of the browser simpler.

If you right-click on the title of one of the entity boxes, you will get the same contextual menu as when you click on the name of an entity in an editor, with the additional items:

- *Go To Spec* Selecting this item will open a source editor that displays the declaration of the entity.
- *Go To Body* Selecting this item will open a source editor that displays the body of the entity.
- *Locate in Project View* Selecting this menu entry will move the focus to the project view, and select the first node representing the file in which the entity is declared. This makes it easier to see which other entities are declared in the same file.

See also *Common features of the browsers* for more capabilities of the GPS browsers.

1.16 The *Bookmarks* view



Bookmarks are a convenient way to remember places in your code or in your environment so that you can go back to them at any point in the future. These bookmarks are saved automatically whenever they are modified, and restored when GPS is reloaded, so that they exist across GPS sessions.

Bookmarks will automatically remember the exact location in an editor, not in terms of line/column, but in terms of which word they point to. If you modify the file through GPS, the bookmark will be automatically updated to keep referring to the same place. Likewise if you close and reopen the file. However, when the file is modified outside of GPS, the bookmark will not be aware of that change, and will thus reference another place in the file.

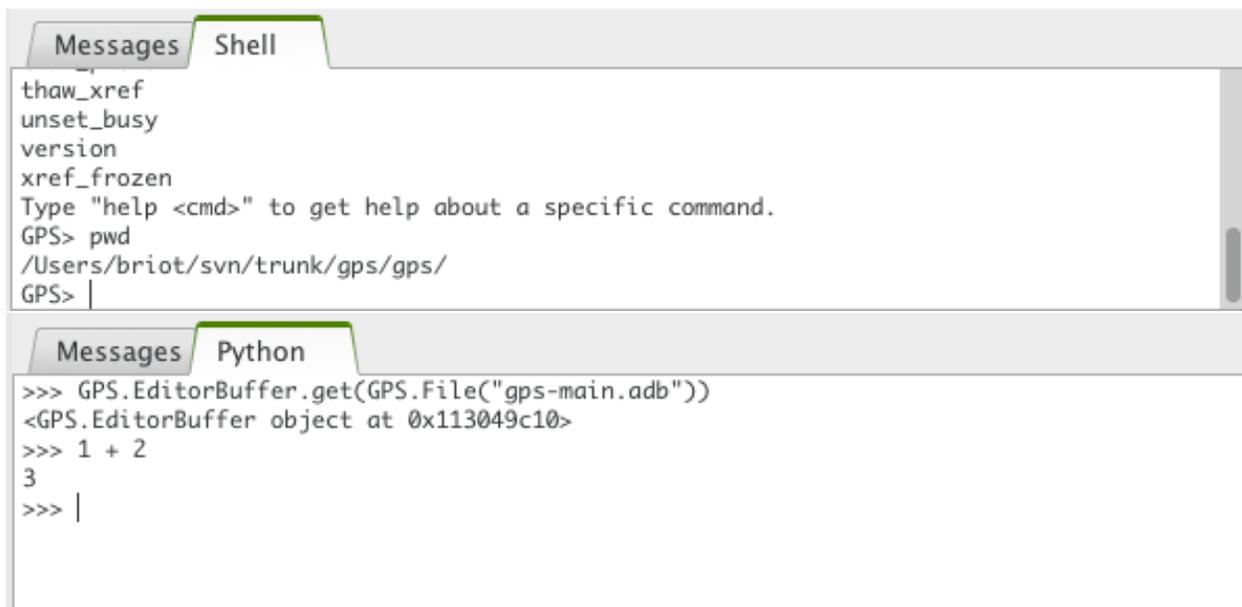
The menu *Edit* → *Create Bookmark* allows you to create a bookmark at the current location (either in the editor, or the browser for instance).

All the bookmarks you have created will be visible in the *Tools* → *Views* → *Bookmarks* window. Clicking on the line will immediately open an editor with the cursor at that position.

In the *Bookmarks* window, the local toolbar provides three buttons to act on the bookmarks:

- *Create* is similar to the *Edit* → *Create Bookmark* and will create a bookmark at the current location. After pressing this button, you can immediately start typing a custom name for the new bookmark (or just press `enter` to keep the default name, which is based on the name of the enclosing subprogram).
- *Rename* can be used to rename the currently selected bookmark. Editing is inline, so you can immediately start typing the new name and press `enter` when done.
- *Remove* is used to delete the selected bookmark.

1.17 The *Shell* and *Python* Windows



```

Messages Shell
thaw_xref
unset_busy
version
xref_frozen
Type "help <cmd>" to get help about a specific command.
GPS> pwd
/Users/briot/svn/trunk/gps/gps/
GPS> |

Messages Python
>>> GPS.EditorBuffer.get(GPS.File("gps-main.adb"))
<GPS.EditorBuffer object at 0x113049c10>
>>> 1 + 2
3
>>> |

```

These windows give access to the various scripting languages supported by GPS, and allow you to type interactive commands such as editing a file or compiling without using the menu items or the mouse.

The menu *Tools* → *Consoles* → *GPS Shell* can be used to open the shell console. The GPS shell is a custom language that was mostly used when GPS did not have python support, and is obsolete at this point.

The menu *Tools* → *Consoles* → *Python* opens the python console. Python is the preferred language to customize your GPS (and many more details will be provided in later sections of this documentation). The console is mostly useful for testing interactive commands before you use them in your own scripts.

See *Scripting GPS* for more information on using scripting languages within GPS.

In both these consoles, GPS provides a history of previously typed commands. You can use the `up` and `down` keys to navigate through the history of commands.

1.18 The OS shell window

```

Messages /bin/zsh
(~/svn/trunk/gps/gps)- (briot@ambre)-
(12:08) echo $OSTYPE
darwin12.0
(~/svn/trunk/gps/gps)- (briot@ambre)-
(12:08) |
(~/svn/trunk/gps/gps)- (briot@ambre)- (3master)-

```

An OS shell window is also available in GPS, providing a simple access to the underlying OS shell as defined by the `SHELL` or `COMSPEC` environment variables.

This console is opened via the *Tools* → *Consoles* → *OS Shell* menu. This menu is available only if the plug-in `shell.py` was loaded in GPS (which is the default).

This console behaves like the standard shell on your system, including support for ANSI sequences (and thus color output). For instance, it has been used to run `vi` within GPS.

Check the documentation of that plug-in, which lists a few settings that might be useful.

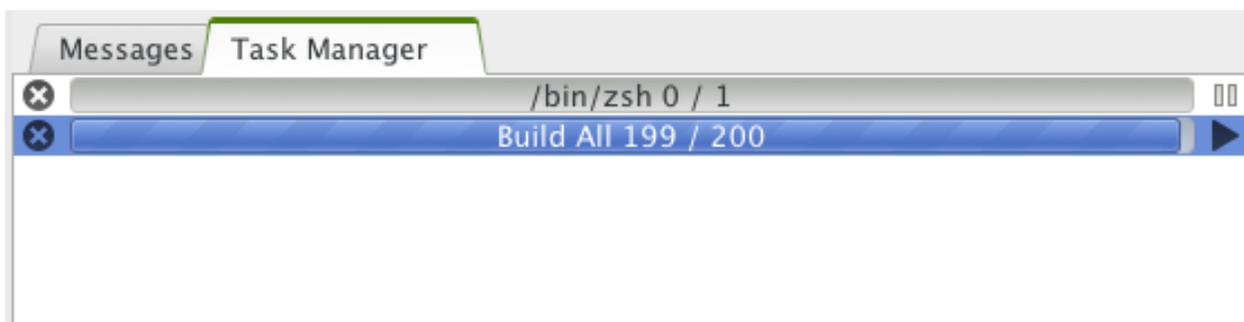
1.19 The Execution window

Each time a program is launched using the menu *Build* → *Run*, a new execution window is created to provide input and output for this program.

In order to allow post mortem analysis and copy/pasting, the execution windows are not destroyed when the application terminates. It must be closed explicitly.

If you close the execution window while the application is still running, a dialog window is displayed, asking whether you want to kill the application, or to cancel the close operation.

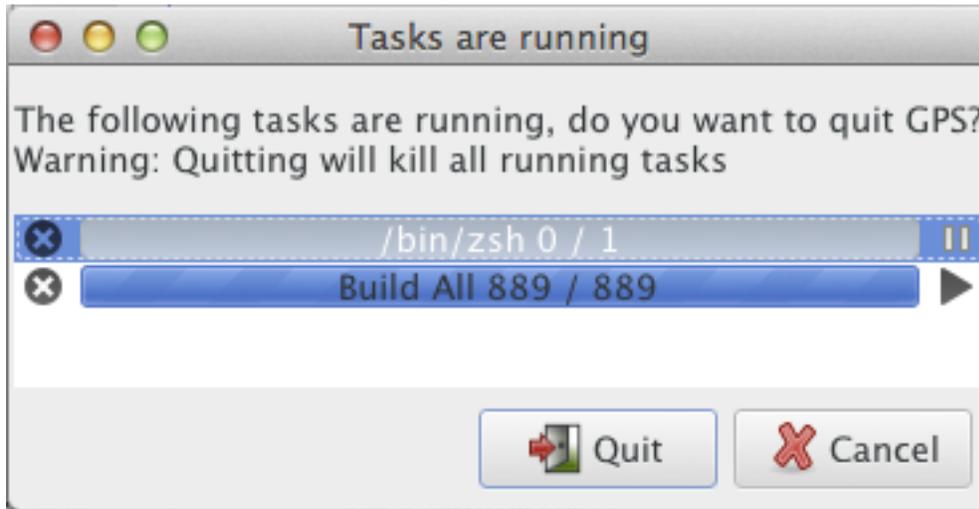
1.20 The Task Manager



The Task Manager window lists all the currently running GPS operations that run in the background, such as builds, searches or VCS commands.

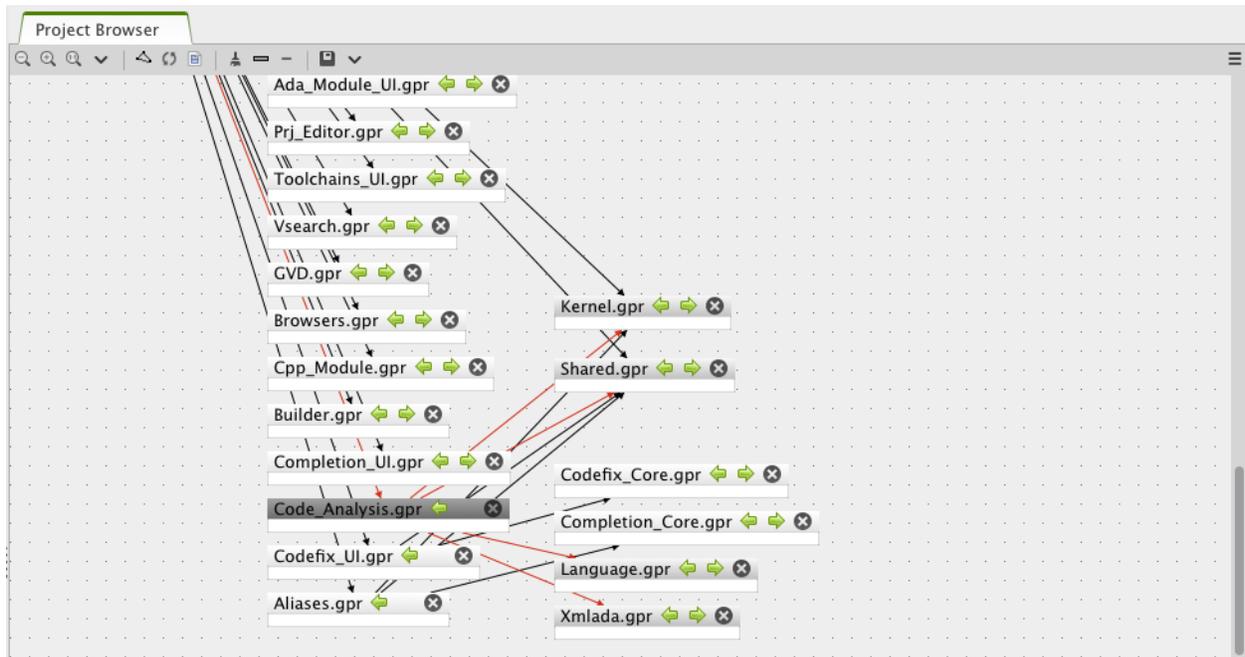
For each of these tasks, the Task Manager shows the status of the task, and the current progress. The execution of these tasks can be suspended by clicking on the small *pause* button next to the task. The tasks can also be killed by clicking on the *interrupt* button.

The Task Manager is opened by double clicking on the progress bar in the main toolbar, or using the *Tools* → *Views* → *Tasks* menu, and can be put anywhere in your desktop.



When exiting GPS, if there are tasks running in the Task Manager, a window will display those tasks. You can force the exit at any time by pressing the confirmation button, which will kill all remaining tasks, or continue working in GPS by pressing the *Cancel* button.

1.21 The *Project Browser*



The project browser shows the dependencies between all the projects in the project hierarchy. Two items in this browser will be linked if one of them imports the other.

It is accessed through the contextual menu in the *Project* view, by selecting the *Show projects imported by...* item, when right-clicking on a project node.

Clicking on the left arrow in the title bar of the items will display all the projects that import that project. Similarly, clicking on the right arrow will display all the projects that are imported by that project.

The contextual menu obtained by right-clicking on a project item contains several items. Most of them are added by the project editor, and gives direct access to editing the properties of the project, adding dependencies...

Some new items are added to the menu:

- *Locate in Project View*

Selecting this menu will switch the focus to the *Project* view, and highlight the first project node found that matches the project in the browser item. This is a convenient way to get information like the list of directories or source files for that project.

- *Show projects imported by...*

This menu plays the same role as the right arrow in the title bar, and display all the projects in the hierarchy that are imported directly by the selected project

- *Show projects imported by ... (recursively)*

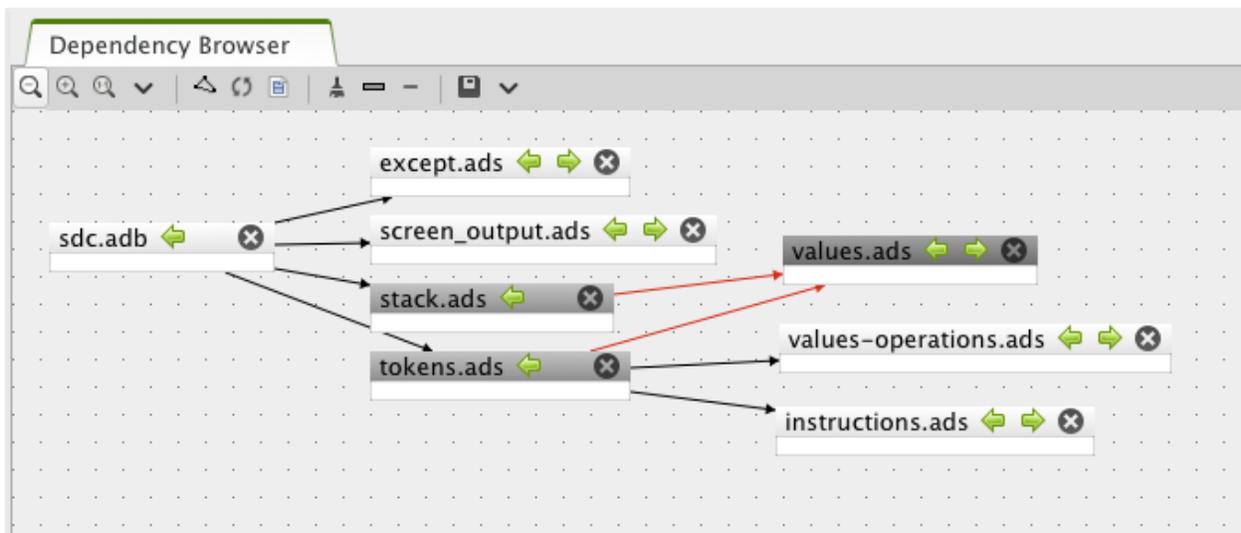
This menu will display all the dependencies recursively for the project (i.e. the projects it imports directly, the projects that are imported by them, and so on).

- *Show projects importing...*

This item plays the same role as the left arrow in the title bar, and displays all the projects that directly import the selected project.

See also *Common features of the browsers* for more capabilities of the GPS browsers.

1.22 The *Dependency Browser*



The dependency browser shows the dependencies between source files. Each item in the browser represents one source file.

In this browser, clicking on the right arrow in the title bar will display the list of files that the selected file depends on. A file depend on another one if it explicitly imports it (*with* statement in Ada, or *#include* in C/C++). Implicit dependencies are currently not displayed in this browser, since the information is accessible by opening the other direct dependencies.

Clicking on the left arrow in the title bar will display the list of files that depend on the selected file.

This browser is accessible through the contextual menu in the *Project* view, by selecting one of the following items:

- *Show dependencies for ...*

This has the same effect as clicking on the right arrow for a file already in the browser, and will display the direct dependencies for that file.

- *Show files depending on ...*

This has the same effect as clicking on the left arrow for a file already in the browser, and will display the list of files that directly depend on that file.

The background contextual menu in the browser adds a few entries to the standard menu:

- *Open file...*

This menu entry will display an external dialog in which you can select the name of a file to analyze.

- *Recompute dependencies*

This menu entry will check that all links displays in the dependency browser are still valid. If not, they are removed. The arrows in the title bar are also reset if necessary, in case new dependencies were added for the files.

The browser is not refreshed automatically, since there are lots of cases where the dependencies might change (editing source files, changing the project hierarchy or the value of the scenario variables, ...)

It also recomputes the layout of the graph, and will change the current position of the boxes.

- *Show system files*

This menu entry indicates whether standard system files (runtime files for instance in the case of Ada) are displayed in the browser. By default, these files will only be displayed if you explicitly select them through the *Open file* menu, or the contextual menu in the project view.

- *Show implicit dependencies*

This menu entry indicates whether implicit dependencies should also be displayed for the files. Implicit dependencies are files that are required to compile the selected file, but that are not explicitly imported through a `with` or `#include` statement. For instance, the body of generics in Ada is an implicit dependency. Any time one of the implicit dependencies is modified, the selected file should be recompiled as well.

The contextual menu available by right clicking on an item also adds a number of entries:

- *Analyze other file*

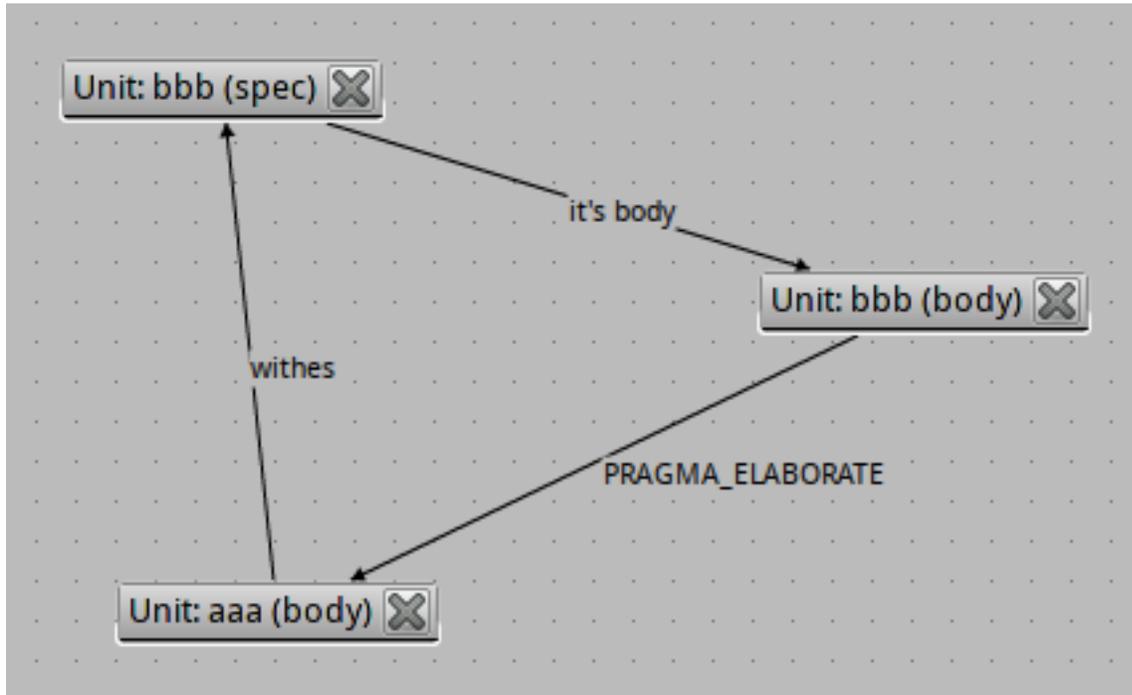
This will open a new item in the browser, displaying the complement file for the selected one. In Ada, this would be the body if you clicked on a spec file, or the opposite. In C, it depends on the naming conventions you specified in the project properties, but you would generally go from a `.h` file to a `.c` file and back.

- *Show dependencies for ...*

These play the same role as in the project view contextual menu

See also *Common features of the browsers* for more capabilities of the GPS browsers.

1.23 The *Elaboration Circularities* browser



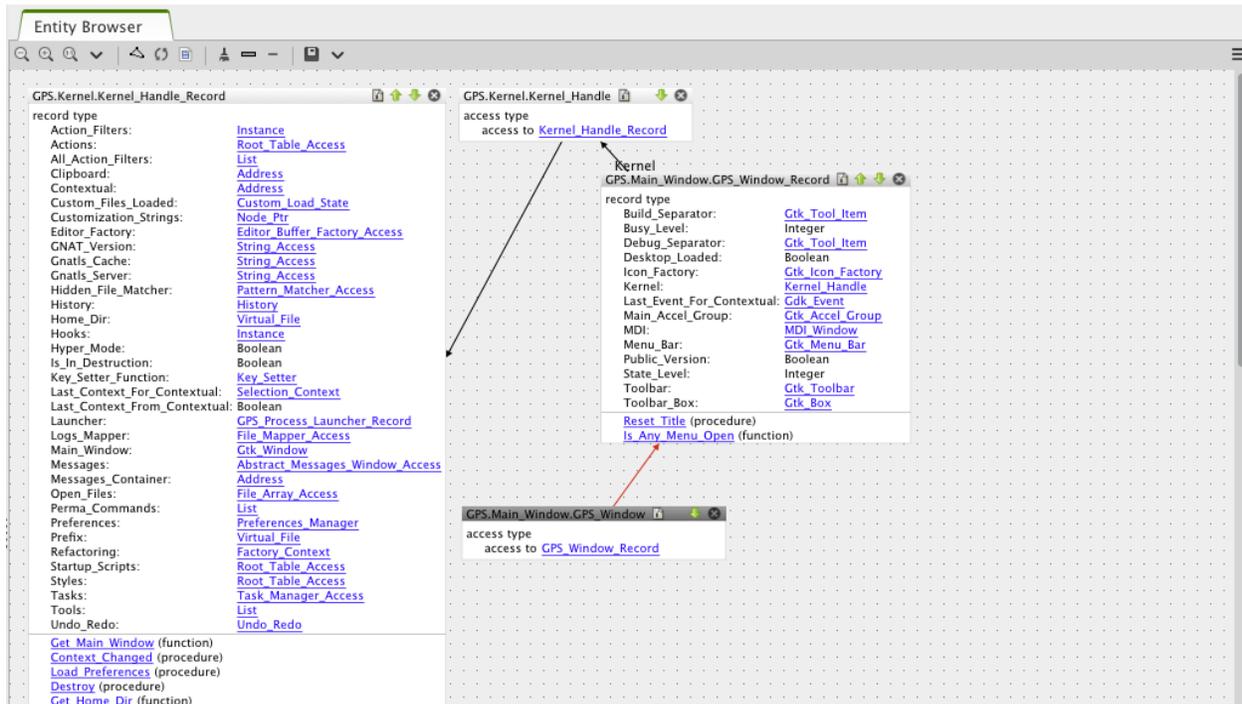
GPS can detect elaboration cycles reported by build processes, and construct a visual representation of elaboration dependencies, in an *Elaboration Cycles* browser.

This visual representation represents program units as items in the browsers, and direct dependencies between program units as links. All units involved in a dependency cycle caused by the presence of a pragma `Elaborate_All` (whether explicit or implicit) are also presented in the browser and connected by links with labels “body” and “with”.

The preference *Browsers* → *Show elaboration cycles* controls whether to automatically create a graph from cycles listed in build output.

See also *Common features of the browsers* for more capabilities of the GPS browsers.

1.24 The Entity Browser



The entity browser displays static information about any source entity. The exact content of the items depend on the type of the item. For instance:

- Ada record / C struct
 - The list of fields, each as an hyper link, is displayed. Clicking on one of the fields will open a new item for the type.
- Ada tagged type / C++ class
 - The list of attributes and methods is displayed. They are also click-able hyper-links.
- Subprograms
 - The list of parameters is displayed
- Packages
 - The list of all the entities declared in that package is displayed
- and more...

This browser is accessible through the contextual menu *Browsers* → *Examine entity* in the project view and source editor, when clicking on an entity.

Most information in the items are clickable (by default, they appear as underlined blue text). Clicking on one of these hyper links will open a new item in the entity browser for the selected entity.

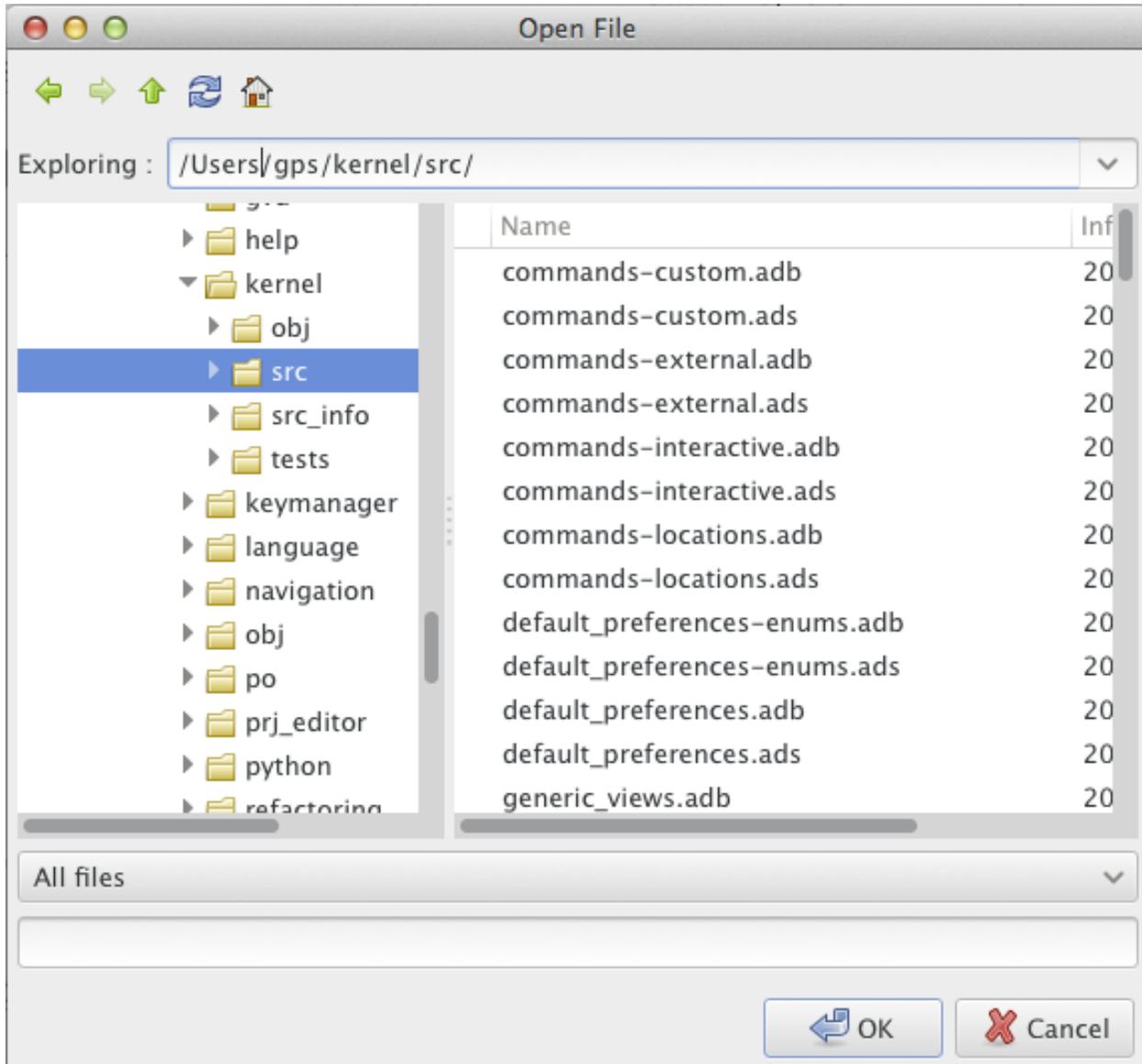
This browser can display the parent entities for an item. For instance, for a C++ class or Ada tagged type, this would be the types it derives from. This is accessible by clicking on the up arrow in the title bar of the item.

Likewise, children entities (for instance types that derive from the item) can be displayed by clicking on the down arrow in the title bar.

An extra button appear in the title bar for the C++ class or Ada tagged types, which toggles whether the inherited methods (or primitive operations in Ada) should be displayed. By default, only the new methods, or the ones that override an inherited one, are displayed. The parent's methods are not shown, unless you click on this title bar button.

See also *Common features of the browsers* for more capabilities of the GPS browsers.

1.25 The File Selector



The file selector is a dialog used to select a file. On Windows, the default is to use the standard file selection widget. On other platforms, the file selector is a built-in dialog:

This dialog provides the following areas and capabilities:

- A tool bar on the top composed of five buttons giving access to common navigation features:
 - *left arrow* go back in the list of directories visited
 - *right arrow* go forward

- *up arrow* go to parent directory
- *refresh* refresh the contents of the directory
- *home* go to home directory (value of the HOME environment variable, or / if not defined)
- A list with the current directory and the last directories explored. You can modify the current directory by modifying the text entry and hitting `Enter`, or by clicking on the right arrow and choose a previous directory in the pop down list displayed.
- A directory tree. You can open or close directories by clicking on the + and - icons on the left of the directories, or navigate using the keyboard keys: `up` and `down` to select the previous or the next directory, + and - to expand and collapse the current directory, and `backspace` to select the parent directory.
- A file list. This area lists the files contained in the selected directory. If a filter is selected in the filter area, only the relevant files for the given filter are displayed. Depending on the context, the list of files may include additional information about the files, e.g. the kind of a file, its size, etc...
- A filter area. Depending on the context, one or several filters are available to select only a subset of files to display. The filter *All files* which is always available will display all files in the directory selected.
- A file name area. This area will display the name of the current file selected, if any. You can also type a file or directory name directly, and complete the name automatically by using the `Tab` key.
- A button bar with the *OK* and *Cancel* buttons. When you have selected the right file, click on *OK* to confirm, or click on *Cancel* at any time to cancel and close the file selection.

MULTIPLE DOCUMENT INTERFACE

All the windows that are part of the GPS environment are under control of what is commonly called a multiple document interface (MDI for short). This is a common paradigm on windowing systems, where related windows are put into a bigger window which is itself under control of the system or the windows manager.

This means that, by default, no matter how many editors, browsers, views, ... windows you have opened, your system will still see only one window (On Windows systems, the task bar shows only one icon). However, you can organize the GPS windows exactly the way you want, all inside the GPS main window.

This section will show the various capacities that GPS provides to help you organize your workspace.

2.1 Window layout

The desktop area is organized into various area.

The most important distinction is that of the central area (which in general occupies the most space) and the side areas.

The central area is meant to contain the source editors (which in fact can only go into that area) as well as the bigger views like the browsers. The property of the central area is that its contents is preserved when switching perspectives (see below).

Some windows in GPS area restricted to either the central or the side areas.

Each of these areas can be further split into smaller area, as will be described below. They can contain any number of window, which are then organized into notebooks (with tabs that can show the names).

Right-clicking on the tab displays a contextual menu that provides various capabilities, including showing the *Tabs location* and *Tabs rotation*. This allows you to display the tabs on any side of the notebook, and make them vertical if you want to save screen space.

2.2 Selecting Windows

At any time, there is only one selected window in GPS (the **active window**). You can select a window either by clicking in its title bar, which will then get a different color, or by selecting its name in the menu *Window*.

An alternative is to use the *Windows* view (see *:The_Windows_View*), which also provides a convenient solution to close multiple windows at once.

Alternatively, windows can be selected with the omni-search, that is the search field in the global toolbar. One of the contexts where the search is done is the list of opened window. To make things more convenient, you can bind a key shortcut via the *Edit* → *Key Shortcuts* menu (the name of the action is *Search* → *Global Search in context: Opened*). If you load the `emacs.py` plugin, a standard key binding is set for `control-xb`.

The filter is matched by any window whose name contains the letter you have typed. For instance, if you are currently editing the files `unit1.adb` and `file.adb`, pressing `t` will only leave `unit1.adb` selectable.

2.3 Closing Windows

Wherever the windows are displayed, they are always closed in the same manner, by clicking on the small *X* icon in their tab.

If you have chosen to display the title bars for the windows, you can also click either in the *X* button in the title bar, or double-click on the icon to the left of the title bar (when there is such an icon).

When a window is closed, the focus is given to the window in the same notebook that previously had the focus. Therefore, if you simply open an editor as a result of a cross-reference query, you can simply close that editor to go back to where you were before.

Alternatively, you can also select the window by clicking anywhere in its title bar, and then select the menu *Window* → *Close*.

Finally, a window can be closed by right-clicking in the associated notebook tab (if the tabs are visible), and select *Close* in the contextual menu.

In the notebook tab (when you are in an editor), you will also find a *Close all other editors* menu, which, as its name implies, will keep a single editor open, the one you are clicking on.

2.4 Splitting Windows

Windows can be split at will, through any combination of horizontal and vertical splits. This feature requires at least two windows (text editors, browsers, ...) to be superimposed in a given notebook. Selecting either the *Window* → *Split Horizontally* or *Window* → *Split Vertically* menus will then split the selected window in two. In the left (resp. top) pane, the currently selected window will be left on its own. The rest of the previously superimposed windows will be put in the right (resp. bottom) pane. You can then in turn split these remaining windows to achieve any layout you want.

All split windows can be resized interactively by dragging the handles that separate them. A preference (menu *Edit* → *Preferences*) controls whether this resizing is done in opaque mode or border mode. In the latter case, only the new handle position will be displayed while the mouse is dragged.

You may want to bind the key shortcuts to the menus *Window* → *Split Horizontally* as well as *Window* → *Split Vertically* using the key manager. In addition, if you want to achieve an effect similar to e.g. the standard Emacs behavior (where `control-x 2` splits a window horizontally, and `control-x 3` splits a window vertically), you can use the key manager (*The Key Manager Dialog*).

Moving Windows will show how to do the splitting through drag-and-drop and the mouse, which in general is the fastest way to do.

Several editors or browsers can be put in the same area of the MDI. In such a case, they will be grouped together in a notebook widget, and you can select any of them by clicking on the corresponding tab. Note that if there are lots of windows, two small arrows will appear on the right of the tabs. Clicking on these arrows will show the remaining tabs.

In some cases GPS will change the color and size of the title (name) of a window in the notebook tab. This indicates that the window content has been updated, but the window wasn't visible. Typically, this is used to indicate that new messages have been written in the messages or console window.

2.5 Floating Windows

Although the MDI, as described so far, is already extremely flexible, it is possible that you prefer to have several top-level windows under direct control of your system or window manager. This would be the case for instance if you want to benefit from some extra possibilities that your system might provide (virtual desktops, different window decoration depending on the window's type, transparent windows, multiple screens, ...).

GPS is fully compatible with this behavior, since windows can also be **floating windows**. Any window that is currently embedded in the MDI can be made floating at any time, simply by selecting the window and then selecting the menu *Window* → *Floating*. The window will then be detached, and can be moved anywhere on your screen, even outside of GPS's main window.

There are two ways to put a floating window back under control of GPS. The more general method is to select the window through its title in the menu *Window*, and then unselect *Window* → *Floating*.

The second method assumes that the preference *Destroy Floats* in the menu *Edit* → *Preferences* has been set to false. Then, you can simply close the floating window by clicking in the appropriate title bar button, and the window will be put back in GPS. If you actually want to close it, you need to click once again on the cross button in its title bar.

A special mode is also available in GPS, where all windows are floating. The MDI area in the main window becomes invisible. This can be useful if you rely on windows handling facilities supported by your system or window manager but not available in GPS. This might also be useful if you want to have windows on various virtual desktops, should your window manager support this.

This special mode is activated through the *Windows* → *All Floating* preference.

2.6 Moving Windows

As we have seen, the organization of windows can be changed at any time by selecting a notebook containing several editors or browsers, and selecting one of the Split menus in the *Window* menu.

A more intuitive method is also provided, based on the drag-and-drop paradigm. The idea is simply to select a window, wherever it is, and then, by clicking on it and moving the mouse while keeping the left button pressed, drop it anywhere else inside GPS.

Selecting an item so that it can be dragged is done simply by clicking with the left mouse button in its title bar, and keep the button pressed while moving the mouse.

If the window is inside a notebook, you can also choose to select the notebook tab to start dragging the window around. In such a case, the windows within the notebook can also be reordered: select the tab, then start moving left or right to the new position the window should have. Note that your mouse must remain within the tab area, since otherwise GPS will enter in the mode where the window can be put in other notebooks.

Here are the various places where a window can be dropped:

- Inside the MDI

While you keep the mouse button pressed, and move the mouse around, the target area is highlighted. This shows precisely where the window would be put if you were to release the mouse button at that point. The background color of the highlight indicates whether the window will be preserved (if the color is the same as the title bar) or not when changing perspectives (for instance when starting a debug session). You can drag to one of the sides of a notebook to split that notebook.

If you move your mouse all the way to the side of the desktop, and then drop the window, that window will occupy the full width (resp. height) of the desktop on that side.

- System window

If you drop a window outside of GPS (for instance, on the background of your screen), the window will be floated.

If you maintain the `shift` key pressed while dropping the window, this might result in a copy operation instead of a simple move. For instance, if you are dropping an editor, a new view of the same editor will be created, resulting in two views present in GPS: the original one is left at its initial location, and a second view is created at the new location.

If you maintain the `control` key pressed while dropping the window, all the windows that were in the same notebook are moved, instead of the single one you selected. This is the fastest way to move a group of windows to a new location, instead of moving them one by one.

2.7 Perspectives

GPS supports the concept of perspectives. These are activity-specific desktops, each with their own set of windows, but sharing some common windows like the editors.

Depending on the activity you want to perform (debugging, version control,...) you could switch to another perspective. For instance, in the context of the debugger, the new perspective would by default contain the call stack window, the data window, the debugger consoles,... each at your favorite location. Whenever the debug starts, you therefore do not have to open these windows again.

The perspectives have names, and you switch perspectives by selecting the menu */Window/Perspectives/*. You can also create a new perspective by selecting the menu */Window/Perspectives/Create New*.

GPS will sometimes automatically change perspectives. For instance, if you start a debugger, it will switch to the perspective called *Debug* (if it exists). When the debugger terminates, you are switched back to the “Default” perspective (again, if it exists).

When you leave a perspective, GPS automatically saves its contents (which windows are opened, their location,...), so that when you are going back to the same perspective you find the same layout.

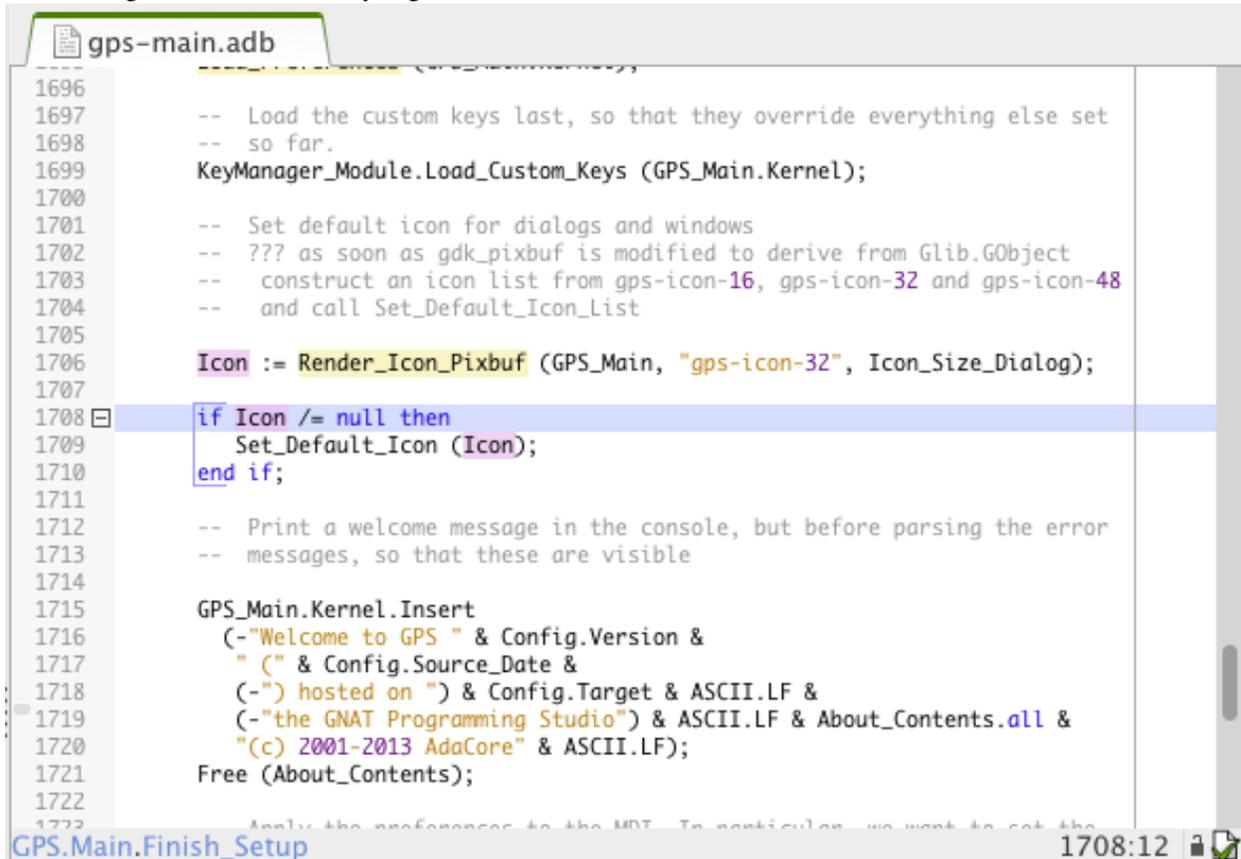
Likewise, when GPS exits, it will save the layout of all perspectives into a file called `perspectives6.xml`, so that it can restore them when you restart GPS. This behavior is controlled by the *General* → *Save desktop on exit* preference, and can be disabled.

One of the difficulties in working with perspectives is knowing which windows will be preserved when you switch to another perspective, and which windows will be hidden. There is a central area where all preserved windows are found. Typically, it only contains editors (including if you have split them side by side for instance). If you drag and drop another window on top or to the sides of an editor, that window will be preserved when changing perspectives, unless it was already found elsewhere in the new perspective. The color of the highlight that appears on the screen while you drag and drop will tell you whether the window (if dropped at the current location) will be visible in other perspectives or not.

EDITING FILES

3.1 General Information

Source editing is one of the central parts of GPS, giving in turn access to many other functionalities, including extended source navigation and source analyzing tools.



```
gps-main.adb
1696
1697 -- Load the custom keys last, so that they override everything else set
1698 -- so far.
1699 KeyManager_Module.Load_Custom_Keys (GPS_Main.Kernel);
1700
1701 -- Set default icon for dialogs and windows
1702 -- ??? as soon as gdk_pixbuf is modified to derive from Glib.GObject
1703 -- construct an icon list from gps-icon-16, gps-icon-32 and gps-icon-48
1704 -- and call Set_Default_Icon_List
1705
1706 Icon := Render_Icon_Pixbuf (GPS_Main, "gps-icon-32", Icon_Size_Dialog);
1707
1708 if Icon /= null then
1709     Set_Default_Icon (Icon);
1710 end if;
1711
1712 -- Print a welcome message in the console, but before parsing the error
1713 -- messages, so that these are visible
1714
1715 GPS_Main.Kernel.Insert
1716   (-"Welcome to GPS " & Config.Version &
1717    " (" & Config.Source_Date &
1718    (-") hosted on ") & Config.Target & ASCII.LF &
1719    (-"the GNAT Programming Studio") & ASCII.LF & About_Contents.all &
1720    "(c) 2001-2013 AdaCore" & ASCII.LF);
1721 Free (About_Contents);
1722
1723 Apply the preferences to the MPI. In particular, we want to set the
GPS.Main.Finish_Setup 1708:12
```

The source editor provides an extensive set of features, including:

Title bar Showing the full name of the file including path information in the title bar of the GPS window.

Line number information This is the left area of the source editor. Line numbers can be disabled with the *Editor* → *Display line numbers* preference. Note that this area can also display additional information, such as the current line of execution when debugging, or cvs annotations.

Scrollbar Located on the right of the editor, it allows you to scroll through the source file. When you scroll, GPS displays a convenient tooltip that shows the subprogram the cursor currently is in.

Speed column This column, when visible, is located on the left of the editor. It allows you to view all the highlighted lines in a file, at a glance. For example, all the lines containing compilation errors are displayed in the Speed Column. The preference *Editor* → *Speed column policy* can be used to control the display of this area. It can sometimes be convenient to keep it visible at all time (to avoid resizing of the editors when new information becomes available), or on the other hand to hide it automatically when it is not needed to save some space on the screen.

Status bar Giving information about the file. It is divided in two sections, one on the left and one on the right of the window.

- the left part of the status bar shows the current subprogram name for languages that support this capability. Currently *Ada*, *C* and *C++* have this ability. The preference *Editor* → *Display subprogram names* controls this display.
- The right section contains multiple pieces of information:
 - The box displays the position of the cursor in the file by a line and a column number. When a selection is performed in the editor, this area also displays the size of the selection (number of lines and number of characters).
 - next to it is an icon that shows whether the file is writable or read-only. You can change this state by clicking on the icon directly: this will toggle between *Writable* and *Read Only*. Note that this will not change the permissions of the file on disk, it will only change the writable state of the source editor within GPS.

When trying to save a file which is read only on the disk, GPS will ask for confirmation, and if possible, will force saving of the file, keeping its read only state.
 - If the file is maintained under version control, and version control is supported and enabled in GPS, the next icon will show VCS information on the file: the VCS kind (e.g. CVS or subversion), followed by the revision number, and if available, the status of the file.

Contextual menu Displayed when you right-click on any area of the source editor. See in particular *Contextual Menus for Source Navigation* for more details.

Syntax highlighting Based on the programming language associated with the file, reserved words and languages constructs such as comments and strings are highlighted in different colors and fonts.

By default, GPS knows about many languages. You can also easily add support for other languages through plug-ins. Most languages supported by GPS will provide syntax highlighting in the editor.

Automatic indentation When enabled, lines are automatically indented each time you press the `Enter` key, or by pressing the indentation key. The indentation key is `Tab` by default, and can be changed in the key manager dialog, *The Key Manager Dialog*.

If a set of lines is selected when you press the indentation key, this whole set of lines will be indented.

Tooltips When you leave the mouse over a word in the source editor, a small window will automatically pop up if there are relevant contextual information to display about the word.

The type of information displayed depends on the current state of GPS.

In normal mode, the entity kind and the location of declaration is displayed when this information is available. That is, when the cross-reference information about the current file has been generated. If there is no relevant information, no tooltip is displayed. See *Support for Cross-References* for more information.

In addition, the documentation for the entity is displayed. This is the block of comments just before or just after the entity's declaration of body. There mustn't be any blank line between the two. For instance, the following are valid documentation for Ada and C:

```
-- A comment for A
A : Integer;

B : Integer;
-- A comment for B

C : Integer;

-- Not a comment for C, there is a blank linke
```

When comments appear both before and after the entity, GPS will chose the one given by the preference *Documentation* → *Leading documentation*.

In debugging mode, the value of the variable under the mouse is displayed in the pop up window if the variable is known to the debugger. Otherwise, the normal mode information is displayed.

You can disable the automatic pop up of tool tips via the preference *Editor* → *Tooltips*.

Code completion GPS provides two kinds of code completion: a *smart code completion* based on semantic information, and a text completion.

The simple text completion is useful when editing a file and using often the same words to get automatic word completion. This is possible by typing the `Ctrl-/` key combination (customizable through the key manager dialog) after a partial word: the next possible completion will be inserted in the editor. Typing this key again will cycle through the list of possible completions.

Text completions are searched in all currently open source files, by first looking at the closest words and then looking further in the source as needed.

Delimiter highlighting When the cursor is moved before an opening delimiter or after a closing delimiter, then both delimiters will be highlighted. The following characters are considered delimiters: `()[]{}.` You can disable highlighting of delimiters with the preference *Editor* → *Highlight delimiters*.

You can also jump to a corresponding delimiter by using the `Ctrl-'` key, that can be configured in the shortcuts editor. Typing twice on this key will move the cursor back to its original position.

Current line highlighting You can configure the editor to highlight the current line with a certain color (see the preference *Editor* → *Fonts & Colors* → *Current line color*).

Current block highlighting If the preference *Editor* → *Block highlighting* is enabled, the editor will highlight the current block of code, e.g. the current *begin...end* block, or loop statement, etc... with a vertical bar to its left side.

The block highlighting will also take into account the changes made in your source code, and will recompute automatically the current block when needed.

This capability is currently implemented for Ada, C and C++ languages.

Block folding When the preference *Editor* → *Block folding* is enabled, the editor will display - icons on the left side, corresponding to the beginning of subprograms. If you click on one of these icons, all the lines corresponding to this block are hidden, except the first one. As for the block highlighting, these icons are recomputed automatically when you modify your sources and are always kept up to date.

This capability is currently implemented for Ada, C and C++ languages.

Auto save You can configure the editor to periodically save modified files. See *Autosave delay* for a full description of this capability.

Automatic highlighting of entities When the cursor is positioned on an entity in the source editor, GPS will highlight all references to this entity in the current editor.

When the cursor moves away from the entity, the highlighting is removed.

This is controlled by the plugin `auto_highlight_occurrences.py`: it can be deactivated by disabling the plugin (*The Plug-ins Editor*).

Details such as presence of indications in the Speed Column or highlighting color can be customized in the *Plugins* section of *The Preferences Dialog*.

GPS also integrates with existing third party editors such as *Emacs* or *vi*. *Using an External Editor*.

3.2 Editing Sources

3.2.1 Key bindings

In addition to the standard keys used to navigate in the editor (up, down, right, left, page up, page down), the integrated editor provides a number of key bindings allowing easy navigation in the file.

There are also several ways to define new key bindings, see *Defining text aliases* and *Binding actions to keys*.

Ctrl-Shift-u	Pressing these three keys and then holding Ctrl-Shift allow you to enter characters using their hexadecimal value. For example, pressing
Ctrl-Shift-u-2	will insert a space character (ASCII 32, which is 20 in hexadecimal).
Ctrl-x Shift-delete	Cut to clipboard
Ctrl-c Shift-insert	Copy to clipboard
Ctrl-v Shift-insert	Paste from clipboard
Ctrl-s	Save file to disk
Ctrl-z	Undo previous insertion/deletion
Ctrl-r	Redo previous insertion/deletion
Insert	Toggle overwrite mode
Ctrl-a	Select the whole file
Home Ctrl-Pgup	Go to the beginning of the line
End Ctrl-Pgdown	Go to the end of the line
Ctrl-Home	Go to the beginning of the file
Ctrl-End	Go to the end of the file
Ctrl-up	Go to the beginning of the line, or to the previous line if already at the beginning of the line.
Ctrl-down	Go to the end of the line, or to the beginning of the next line if already at the end of the line.
Ctrl-delete	Delete end of the current word.
Ctrl-backspace	Delete beginning of the current word.

3.3 Menu Items

The main menus that give access to extended functionalities related to source editing are described in this section.

3.3.1 The *File* Menu

File → **New** Open a new untitled source editor. No syntax highlighting is performed until the file is saved, since GPS needs to know the file name in order to choose the programming language associated with a file.

When you save a new file for the first time, GPS will ask you to enter the name of the file. In case you have started typing Ada code, GPS will try to guess based on the first main entity in the editor and on the current naming scheme, what should be the default name of this new file.

File → **New View** Create a new view of the current editor. The new view shares the same contents: if you modify one of the source views, the other view is updated at the same time. This is particularly useful when you want to display two separate parts of the same file, for example a function spec and its body.

A new view can also be created by keeping the `shift` key pressed while drag-and-dropping the editor (see [Moving Windows](#)). This second method is preferred, since you can then specify directly where you want to put the new view. The default when using the menu is that the new view is put on top of the editor itself.

File → **Open...** Open a file selection dialog where you can select a file to edit. On Windows, this is the standard file selector. On other platforms, this is a built-in file selector described in [The File Selector](#).

File → **Open From Project...** Moves the focus to the [The omni-search](#) field, where you can immediately start typing part of the file name you wish to open. This is the fastest way to select files to open.

File → **Open From Host...** Open a file selector dialog where you can specify a remote host, as defined in [The remote configuration dialog](#). You have access to a remote host file system, can specify a file which can be edited in GPS. When you hit the save button or menu, the file will be saved on the remote host.

See also [Using GPS for Remote Development](#) for a more efficient way to work locally on remote files.

File → **Recent** Open a sub menu containing a list of the ten most recent files opened in GPS, so that you can reopen them easily.

File → **Save** Save the current source editor if needed.

File → **Save As...** Same current file under a different name, using the file selector dialog. [The File Selector](#).

File → **Save More** Give access to extra save capabilities:

- **File** → **Save More** → **All** Save all items, including projects, etc...
- **File** → **Save More** → **Desktop** Save the desktop to a file. The desktop includes information about files, graphs, ... and their window size and position in GPS. The desktop is saved per top level project, so that if you reload the same project you get back to the same situation you were in when you left GPS. Instead, if you load a different project another desktop will be loaded (or the default desktop). Through the preference *General* → *Save Desktop On Exit*, you can also automatically save this desktop when you quit GPS.

File → **Change Directory...** Open a directory selection dialog that lets you change the current working directory.

File → **Locations** This sub menu gives access to functionalities related to the *Locations* window.

- **File** → **Locations** → **Export Locations to Editor** List the contents of the *Locations* view in a standard text editor.

File → **Print** Print the current window contents, optionally saving it interactively if it has been modified. The Print Command specified in the preferences is used if it is defined. On Unix this command is required; on Windows it is optional.

On Windows, if no command is specified in the preferences the standard Windows print dialog box is displayed. This dialog box allows the user to specify the target printer, the properties of the printer, which pages to print (all, or a specific range of pages), the number of copies to print, and, when more than one copy is specified, whether the pages should be collated. Pressing the *Cancel* button on the dialog box returns to GPS without printing the window contents; otherwise the specified pages and copies are printed on the selected printer. Each page is printed with a header containing the name of the file (if the window has ever been saved). The page number is printed on the bottom of each page.

See also: [ref:Print Command](#) <*Print_Command*>.

File → **Close** Close the current window. This applies to all GPS windows, not just source editors.

File → **Exit** Exit GPS after confirmation and if needed, confirmation about saving modified windows and editors.

3.3.2 The *Edit* Menu

Edit → **Cut** Cut the current selection and store it in the clipboard.

Edit → **Copy** Copy the current selection to the clipboard.

Edit → **Paste** Paste the contents of the clipboard to the current cursor position.

Edit → **Paste previous** GPS stores a list of all the text that was previously copied into the clipboard through the use of *Copy* or *Cut*.

By default, if you press *Paste*, the newest text will be copied at the current position. But if you select *Paste Previous* immediately after (one or more times) you can instead paste text that was copied previously in the clipboard.

For instance, if you copy through *Edit* → *Copy* the text “First”, then copy the text “Second”, you can then select *Edit* → *Paste* to insert “Second” at the current location. If you then select *Edit* → *Paste Previous*, “Second” will be replaced by “First”.

Selecting this menu several times will replace the text previously pasted by the previous one in the list saved in the clipboard. When reaching the end of this list, GPS will start from the beginning, and insert again the last text copied into the clipboard.

The size of this list is controlled by the *General* → *Clipboard Size* preference.

For more information, [The Clipboard view](#).

Edit → **Undo** Undo previous insertion/deletion in the current editor.

Edit → **Redo** Redo previous insertion/deletion in the current editor.

Edit → **Rectangles...** See the section [Rectangles](#) for more information on rectangles.

Edit → **Rectangles... -> Serialize** Increment a set of numbers found on adjacent lines. The exact behavior depends on whether there is a current selection or not.

If there is no selection, then the set of lines considered is from the current line on and includes all adjacent lines that have at least one digit in the original columns. In the following example, ‘|’ marks the place where the cursor is at the beginning:

```
AAA |10 AAA
CCC 34567 CCC
DDD DDD
```

then only the first two lines will be modified, and will become:

```
AAA 10 AAA
CCC 11 CCC
DDD DDD
```

If there is a selection, all the lines in the selection are modified. For each line, the columns that had digits in the first line are modified, no matter what they actually contain. In the example above, if you select all three lines, the replacement becomes:

```
AAA 10 AAA
CCC 11567 CCC
DDD 12D
```

ie only the fifth and sixth columns are modified since only those columns contained digits in the first line. This feature assumes that you are selecting a relevant set of lines. But it allows you to transform blank lines more easily. For instance, if you have:

```
AAA 1
BBB
CCC
```

this is transformed into:

```
AAA 1
BBB 2
CCC 3
```

Edit → **Select all** Select the whole contents of the current source editor.

Edit → **Insert File...** Open a file selection dialog and insert the contents of this file in the current source editor, at the current cursor location.

Edit → **Insert Shell Output...** Open an input window at the bottom of the GPS window where you can specify any external command. The output of the command will be inserted at the current editor location in case of success. If text is selected, the text is passed to the external command and replaced by the command's output.

Edit → **Format selection** Indent and format the selection or the current line. *The Preferences Dialog*, for preferences related to source formatting.

Edit → **Smart completion** Complete the identifier prefix under the cursor, and list the results in a pop-up list. Used with Ada sources this command can take advantage of an entity database as well as Ada parsers embedded in GPS which analyze the context, and offer completions from the entire project along with documentation extracted from comments surrounding declarations. To take full advantage of this feature, the smart completion preference must be enabled, which will imply the computation of the entity database at GPS startup.

The support for C and C++ is not as powerful as the support for Ada since it relies completely on the xref information files generated by the compiler, does not have into account the C/C++ context around the cursor, and does not extract documentation from comments around candidate declarations. To take advantage of this feature, in addition to enable the smart completion preference, the C/C++ application must be built with *-fdump-xref*.

In order to use this feature, open any Ada, C or C++ file, and begin to type an identifier. It has to be an identifier declared either in the current file (and accessible from the cursor location) or in one of the packages of the project loaded. Move the cursor right after the last character of the incomplete identifier and hit the completion key (which is `control-space` by default). GPS will open a popup displaying all the known identifiers beginning with the prefix you typed. You can then browse among the various proposals by clicking on the `up` and `down` keys, or using the left scrollbar. For each entity, a documentation box is filled. If the location of the entity is known, it's displayed as an hyperlink, and you can jump directly to its declaration by clicking on it.

Typing new letters will reduce the range of proposal, as long as there remain solutions. Once you've selected the expected completion, you can validate by pressing `Enter`.

Typing control characters (ie, characters which cannot be used in identifiers) will also validate the current selection.

GPS is also able to complete automatically subprogram parameter or dotted notations. For example, if you type:

```
with Ada.
```

the smart completion window will appear automatically, listing all the child and nested packages of Ada. You can configure the time interval after which the completion window appears (*The Preferences Dialog*).

You can also write the beginning of the package, e.g.:

```
with Ada.Text
```

pressing the completion key will offer you Text_IO.

If you are in a code section, you will be able to complete the fields of a record, or the contents of a package, e.g.:

```
declare
  type R is record
    Field1 : Integer;
    Field2 : Integer;
  end record;

  V : R;
begin
  V.
```

Completing V. will propose Field1 and Field2.

The smart completion can also give you the possible parameters of a call you're currently making. For example, in the following code:

```
procedure Proc (A, B, C : Integer);
begin
  Proc (1,
```

If you hit the completion key after the comma, the smart completion engine will propose you to complete with the named parameters “B =>”, “C =>” or directly to complete with all the remaining parameters, which in this case will be “B =>, C =>”).

```
procedure Ref (Entity : Entity_Information; Reason : String) is
begin
  Language.
  if Entity
  Trace
  end if;
  Ref (Enti
end Ref;

-----
-- Unref --
-----

procedure Un
begin
  if Entity
  Trace
  (Ref
  & "
  & E
  end if;
  Unref (En
end Unref;
```

Declaration at: [language.ads:253:4](#)

Same as Skip_To_Current_Comment_Block_Start, except we move forward to the beginning of the last line of comments in the block. If Ignore_Blank_Lines is set to True, blocks separated from one another with blank lines are considered as a single one.

Parameters:

Context	: in	Language_Context
Buffer	: in	String
Index	: in out	Natural
Ignore_Blank_Lines	: in	Boolean := False]

Limitations:

- This feature is currently only available for Ada, C and C++. Using the smart completion on sources of other languages behaves as the *identifier completion* does.
- Smart completion for C and C++ is based on the xref information generated by the compiler. Therefore, GPS has no knowledge on recently edited files. You must rebuild with `-fdump-xref` to update the completion database.
- Smart completion for C and C++ is only triggered at the beginning of an expression (that is, it is not triggered on special characters such as ‘(’, ‘->’, or the C++ operator ‘::’) and it may propose too much candidates since it does not have into account the C/C++ syntax context. Typing new letters will reduce the range of proposal, as long as there remain solutions.
- Smart completion of subprogram parameters, fields and dotted notation are not available yet for C and C++.

Edit → **More Completion** This submenu contains more ways to automatically complete code

- *Edit* → *More Completion* → *Expand alias*

Consider the current word as an alias and expand according to aliases defined in *Defining text aliases*.

- *Edit* → *More Completion* → *Completion Identifier*

Complete the identifier prefix under the cursor. This command will cycle through all identifiers starting with the given prefix.

- *Edit* → *More Completion* → *Complete block*

Close the current statement (if, case, loop) or unit (procedure, function, package). This action works only on an Ada buffer.

Edit → **Selection** This submenu contains actions that apply to the current selection in the editor.

- *Edit* → *Selection* → *Comment lines*

Comment the current selection or line based on the current programming language syntax.

- *Edit* → *Selection* → *Uncomment lines*

Remove the comment delimiters from the current selection or line.

- *Edit* → *Selection* → *Refill*

Refill text on the selection or current line: rearrange line breaks in the paragraph so that line lengths do not exceed the maximum length, as set in the “Right margin” preference (*The Preferences Dialog*).

- *Edit* → *Selection* → *Sort*

Sort the selected lines alphabetically. This is particularly useful when editing non source code, or for specific parts of the code, like with clauses in Ada.

- *Edit* → *Selection* → *Sort Reverse*

Sort the selected lines in reverse alphabetical order

- *Edit* → *Selection* → *Pipe in external program...*

Open an input window at the bottom of the GPS window where you can specify any external command, which will take the current selection as input. The output of the command will replace the contents of the selection on success.

- *Edit* → *Selection* → *Untabify*

Replace all tabs in the current selection (or in the whole buffer if there is no selection) by the appropriate number of spaces

- *Edit* → *Selection* → *Move Right*

- *Edit* → *Selection* → *Move Left*

Shift the currently selected lines (or the current line if there is no selection) one character to the right or to the left

Edit → **Fold all blocks** Collapse all the blocks in the current file.

Edit → **Unfold all blocks** Uncollapse all the blocks in the current file.

Edit → **Create bookmark** Creates a new Bookmark at cursor position. For more information, *The Bookmarks view*.

Edit → **Pretty Print** Pretty print the current source editor by calling the external tool *gnatpp*. It is possible to specify *gnatpp* switches in the switch editor. *The Switches Editor*.

Edit → **Generate Body** Generate Ada body stub for the current source editor by calling the external tool *gnatstub*.

Edit → **Edit with external editor** *Using an External Editor*.

Edit → **Aliases** Display the Aliases editor. *Defining text aliases*.

Edit → **Key shortcuts** Give access to the key manager dialog, to associate commands with special keys. *The Key Manager Dialog*.

Edit → **Preferences** Give access to the preferences dialog. *The Preferences Dialog*.

3.4 Rectangles

Rectangle commands operate on a rectangular area of the text, that is all the characters between two columns in a certain range of lines.

A rectangle is selected using the standard selection mechanism. You can therefore use either the mouse to highlight the proper region, or `shift` and the cursor keys to extend the selection, or the Emacs selection (with the mark and the current cursor location) if you have activated the `emacs.py` plugin.

Visually, a selected rectangle is exactly the same as the standard selection. In particular, the characters after the last column, on each line, will also be highlighted. The way the selection is interpreted (either as a full text or as a rectangle) depends on the command you then chose to manipulate the selection.

If you chose one of the commands from the *Edit* → *Rectangles* menu, the actual rectangle will extend from the top-left corner down to the bottom-right corner. All characters to the right of the right-most column, although they are highlighted, are not part of the rectangle.

Consider for instance the following initial text:

```
package A is
  procedure P;

  procedure Q;
end A;
```

and assume we have selected from the character “p” in “procedure P”, down to the character “c” in “procedure Q”.

The following commands can then be used (either from the menu, or you can assign key shortcuts to them via the usual *Edit* → *Key shortcuts* menu).

- *Edit* → *Rectangles* → *Cut* or *Edit* → *Rectangles* → *Delete*

These commands will remove the selected text (and have no effect on empty lines within the rectangle). The former will in addition copy the rectangle to the clipboard, so that you can paste it later. In our example, we end up with:

```
package A is
  edure P;

  edure Q;
end A;
```

- *Edit* → *Rectangles* → *Copy* This command has no visual effect, but copies the contents of the rectangle into the clipboard.
- *Edit* → *Rectangles* → *Paste* Pastes the contents of the clipboard as a rectangle: each line from the clipboard is treated independently, and inserted on successive lines in the current editor. They all start in the same column (the one where the cursor is initially in), and existing text in the editor lines is shifted to the right). If for instance you now place the cursor in the second line, first column, and paste, we end up with:

```
package A is
  procedure P;

  procedure Q;
end A;
```

- *Edit* → *Rectangles* → *Clear* Replaces the contents of the selected rectangle with spaces. If we start from our initial example, we end up with the following. Note the difference with *Edit* → *Rectangles* → *Delete*:

```
package A is
  procedure P;

  procedure Q;
end A;
```

- *Edit* → *Rectangles* → *Open* Replaces the contents of the selected rectangle with spaces, but shifts the lines to the right to do so. Note the difference with *Edit* → *Rectangles* → *Clear*:

```
package A is
  procedure P;

  procedure Q;
end A;
```

- *Edit* → *Rectangles* → *Replace With Text* This is similar to *Edit* → *Rectangles* → *Clear*, but the rectangle is replaced with user-defined text. The lines will be shifted left or right if the text you insert is shorter (resp. longer) than the width of the rectangle. If for instance we replace our initial rectangle with the text TMP, we end up with the following. Note that the character “c” has disappeared, since TMP is shorter than our rectangle width (4 characters). This command will impact lines that are empty in the initial rectangle:

```
package A is
  TMPprocedure P;
  TMP
  TMPprocedure Q;
end A;
```

- *Edit* → *Rectangles* → *Insert Text* This inserts a text to the left of the rectangle on each line. The following example inserts TMP. Note the difference with *Edit* → *Rectangles* → *Replace With Text*. This command will also insert the text on lines that are empty in the initial rectangle:

```
package A is
  TMPprocedure P;
  TMP
  TMPprocedure Q;
end A;
```

- *Edit* → *Rectangles* → *Sort* This sorts the selected lines according to the key which starts and ends on the corresponding rectangle’s columns:

```
aaa 15 aa
bbb 02 bb
ccc 09 cc
```

With a selection starting from the 1 on the first line and ending on the 9 on the last one, sorting will result with the following content:

```
bbb 02 bb
ccc 09 cc
aaa 15 aa
```

- *Edit* → *Rectangles* → *Sort reverse*

As above but in the reverse order.

3.5 Recording and replaying macros

It is often convenient to be able to repeat a given key sequence a number of times.

GPS supports this with several different methods:

- Repeat the next action

If there is a single key press that you wish to repeat a number of times, you should first use the GPS action “*Repeat Next*” (bound by default to `control-u`, but this can be changed as usual through the *Edit* → *Key Shortcuts* menu), then entering the number of times you wish to repeat, and finally pressing the key you want.

For instance, the following sequence `control-u 79 -` will insert 79 characters ‘-’ in the current editor. This proves often useful to insert separators.

If you are using the emacs mode (see *Tools* → *Plug-ins* menu), you can also use the sequence `control-u 30 control-k` to delete 30 lines.

- Recording macros

If you wish to repeat a sequence of more than 1 key, you should record this sequence as a macro. All macro-related menus are found in *Tools* → *Macros*, although it is often more convenient to use these through key bindings, which you can of course override.

You must indicate to GPS that it should start recording the keys you are pressing. This is done through the *Tools* → *Macros* → *Start Keyboard Macro* menu. As its name indicates, this only records keyboard events, not mouse events. Until you select *Tools* → *Macros* → *Stop Macro*, GPS will keep recording the events.

In Emacs mode, the macro actions are bound to `control-x (`, `control-x)` and `control-x e` key shortcuts. For instance, you can execute the following to create a very simple macro that deletes the current line, wherever your cursor initially is on that line:

- `control-x (` (start recording
- `control-a` go to beginning of line
- `control-k` delete line
- `control-x)` stop recording

3.6 Contextual Menus for Editing Files

Whenever you ask for a contextual menu (using e.g. the right button on your mouse) on a source file, you will get access to a number of entries, displayed or not depending on the current context.

Menu entries include the following categories:

Source Navigation *Contextual Menus for Source Navigation.*

Dependencies *The Dependency Browser.*

Entity browsing *The Entity Browser.*

Project view *The Project view.*

Version control *The Version Control Contextual Menu.*

Debugger *Using the Source Editor when Debugging.*

Case exceptions *Handling of case exceptions.*

Refactoring *Refactoring.*

In addition, an entry *Properties...* is always visible in this contextual menu. When you select it, a dialog pops up that allows you to override the language used for the file, or the character set.

This can be used for instance if you want to open a file that does not belong to the current project, but where you want to benefit from the syntax highlighting.

It is not recommended to override the language for source files that belong to the project. Instead, you should use the *Project → Edit Project Properties* menu and change the naming scheme if appropriate. This will ensure better consistency between GPS and the compiler in the way they manipulate the file.

3.7 Handling of case exceptions

GPS keeps a set of case exceptions that is used by all case insensitive languages. When editing or reformatting a buffer for such a language the case exception dictionary will be checked first. If an exception is found for this word or a substring of the word, it will be used; otherwise the specified casing for keywords or identifiers is used. A substring is defined as a part of the word separated by underscores.

Note that this feature is not activated for entities (keywords or identifiers) for which the casing is set to *Unchanged* in the preferences *Editor → Ada → Reserved word casing* or *Editor → Ada → Identifier casing*.

A contextual menu named *Casing* has the following entries:

Casing → *Lower *entity** Set the selected entity in lower case.

Casing → *Upper *entity** Set the selected entity in upper case.

Casing → *Mixed *entity** Set the selected entity in mixed case (set the first letter and letters before an underscore in upper case, all other letters are set to lower case).

Casing → *Smart Mixed *entity** Set the selected entity in smart mixed case. Idem as above except that upper case letters are kept unchanged.

Casing → *Add exception for *entity** Add the current entity into the case exception dictionary.

Casing → *Remove exception for *entity** Remove the current entity from the case exception dictionary.

To add or remove a substring exception into/from the dictionary you need to first select the substring on the editor. In this case the last two contextual menu entries will be:

Casing → *Add substring exception for *str** Add the selected substring into the case substring exception dictionary.

Casing → *Remove substring exception for *str** Remove the selected substring from the case substring exception dictionary.

3.8 Refactoring

GPS includes basic facilities for refactoring your code. Refactoring is the standard term used to describe manipulation of the source code that do not affect the behavior of the application, but help reorganize the source code to make it more readable, more extendable, ...

Refactoring technics are generally things that programmers are used to do by hand, but which are faster and more secure to do automatically through a tool.

One of the basic recommendations when you refactor your code is to recompile and test your application very regularly, to make sure that each of the small modifications you made to it didn't break the behavior of your application. This is particularly true with GPS, since it relies on the cross-references information that is generated by the compiler. If some of the source files have not been recompiled recently, GPS will print warning messages indicating that the renaming operation might be dangerous and/or only partial.

One of the reference books that was used in the choice of refactoring methods to implement is “Refactoring”, by Martin Fowler (Addison Wesley).

3.8.1 Rename Entity

Clicking on an entity in a source file and selecting the *Refactoring* → *Rename* contextual menu will open a dialog asking for the new name of the entity. GPS will rename all instances of the entity in your application. This includes the definition of the entity, its body, all calls to it, etc... Of course, no comment is updated, and you should probably check manually that the comment for the entity still applies.

GPS will handle primitive operations by also renaming the operations it overrides or that overrides it. This means that any dispatching call to that operation will also be renamed, and the application should still work as before. If you are renaming a parameter to a subprogram, GPS can also rename parameters with similar names in overriding or overridden subprograms.

The behavior when handling read-only files can be specified: by default, GPS will not do any refactoring in these files, and will display a dialog listing all of them; but you can also choose to make them writable just as if you had clicked on the “Read-Only” button in the status bar of the editor and then have GPS perform the renaming in them as well.

3.8.2 Name Parameters

If you are editing Ada code and click on a call to a subprogram, GPS will display a contextual menu *Refactoring* → *Name parameters*, which will replace all unnamed parameters by named parameters, as in:

```
Call (1, 2)
=>
Call (Param1 => 1, Param2 => 2);
```

3.8.3 Extract Subprogram

This refactoring is used to move some code from one place to a separate subprogram. The goal is to simplify the original subprogram, by moving part of its code elsewhere.

Here is an example from the “Refactoring” book. The refactoring will take place in the body of the package `pkg.adb`, but the spec is needed so that you can compile the source code (a preliminary step mandatory before you can refactor the code):

```
pragma Ada_05;

with Ada.Containers.Indefinite_Doubly_Linked_Lists;
with Ada.Strings.Unbounded;

package Pkg is

    type Order is tagged null record;
    function Get_Amount (Self : Order) return Integer;

    package Order_Lists is new
        Ada.Containers.Indefinite_Doubly_Linked_Lists (Order);
```

```

type Invoice is tagged record
  Orders : Order_Lists.List;
  Name   : Ada.Strings.Unbounded.Unbounded_String;
end record;

procedure Print_Owing (Self : Invoice);

end Pkg;

```

The initial implementation for this code is given by the following code:

```

pragma Ada_05;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Text_IO;          use Ada.Text_IO;

package body Pkg is
  use Order_Lists;

  -----
  -- Get_Amount --
  -----

  function Get_Amount (Self : Order) return Integer is
  begin
    return 0;
  end Get_Amount;

  -----
  -- Print_Owing --
  -----

  procedure Print_Owing (Self : Invoice) is
    E : Order_Lists.Cursor := First (Self.Orders);
    Outstanding : Natural := 0;
    Each : Order;
  begin
    -- <<< line 30
    -- Print Banner

    Put_Line ("");
    Put_Line (" Customer Owes          ");
    Put_Line (""); -- << line 35

    -- Calculate Outstanding

    while Has_Element (E) loop
      Each := Element (E);
      Outstanding := Outstanding + Each.Get_Amount;
      Next (E);
    end loop;

    -- Print Details

    Put_Line ("Name: " & To_String (Self.Name));
    Put_Line ("Outstanding:" & Outstanding'Img);
  end Print_Owing;
end Pkg;

```

The procedure *Print_Owing* is too long and does several independent actions. We will perform a series of three successive refactoring steps to extract the code and move it elsewhere.

The first is the code that prints the banner. Moving it is easy, since this code does not depend on any context. We could just do a copy-paste, but then we would have to create the new subprogram. Instead, we select lines 30 to 35, and then select the contextual menu *Refactoring* → *Extract Subprogram*. GPS will then automatically change *Print_Owing* and create a new procedure *Print_Banner* (the name is specified by the user, GPS does not try to guess it). Also, since the chunk of code that is extracted starts with a comment, GPS automatically uses that comment as the documentation for the new subprogram. Here is part of the resulting file:

```
package body Pkg is

  procedure Print_Banner;
  -- Print Banner

  -----
  -- Print_Banner --
  -----

  procedure Print_Banner is
  begin
    Put_Line ("");
    Put_Line (" Customer Owes          ");
    Put_Line ("");
  end Print_Banner;

  ... (code not shown)

  procedure Print_Owing (Self : Invoice) is
    E : Order_Lists.Cursor := First (Self.Orders);
    Outstanding : Natural := 0;
    Each : Order;
  begin
    Print_Banner;

    -- Calculate Outstanding

    while Has_Element (E) loop
      Each := Element (E);
      Outstanding := Outstanding + Each.Get_Amount;
      Next (E);
    end loop;

    -- Print Details   <<< line 54

    Put_Line ("Name: " & To_String (Self.Name));
    Put_Line ("Outstanding:" & Outstanding'Img); -- line 57
  end Print_Owing;
end Pkg;
```

A more interesting example is when we want to extract the code to print the details of the invoice. This code depends on one local variable and the parameter to *Print_Owing*. When we select lines 54 to 57 and extract it into a new *Print_Details* subprogram, we get the following result. GPS automatically decides which variables to extract, and whether they should become parameters of the new subprogram, or local variables. In the former case, it will also automatically decide whether to create “in”, “out” or “in out” parameters. If there is a single “out” parameter, it will automatically create a function rather than a procedure.

GPS will use, for the parameters, the same name that was used for the local variable. Very often, it will make sense to recompile the new version of the source, and then apply the *Refactoring* → *Rename Entity* refactoring to have

more specific names for the parameters, or the *Refactoring* → *Name Parameters* refactoring so that the call to the new method uses named parameters to further clarify the code:

... code **not** shown

```

procedure Print_Details
  (Self : Invoice'Class;
   Outstanding : Natural);
-- Print_Details

-----
-- Print_Details --
-----

procedure Print_Details
  (Self : Invoice'Class;
   Outstanding : Natural)
is
begin
  Put_Line ("Name: " & To_String (Self.Name));
  Put_Line ("Outstanding:" & Outstanding'Img);
end Print_Details;

procedure Print_Owing (Self : Invoice) is
  E : Order_Lists.Cursor := First (Self.Orders);
  Outstanding : Natural := 0;
  Each : Order;
begin
  Print_Banner;

  -- Calculate Outstanding

  while Has_Element (E) loop
    Each := Element (E);
    Outstanding := Outstanding + Each.Get_Amount;
    Next (E);
  end loop;

  Print_Details (Self, Outstanding);
end Print_Owing;

```

Finally, we want to extract the code that computes the outstanding amount. When this code is moved, the variables *E* and *Each* become useless in *Print_Owing* and are moved into the new subprogram (which we will call *Get_Outstanding*. Here is the result of that last refactoring (the initial selection should include the blank lines before and after the code, to keep the resulting *Print_Owing* simpler). GPS will automatically ignore those blank lines:

... code **not** shown

```

procedure Get_Outstanding (Outstanding : in out Natural);
-- Calculate Outstanding

-----
-- Get_Outstanding --
-----

procedure Get_Outstanding (Outstanding : in out Natural) is
  E : Order_Lists.Cursor := First (Self.Orders);
  Each : Order;
begin

```

```
while Has_Element (E) loop
  Each := Element (E);
  Outstanding := Outstanding + Each.Get_Amount;
  Next (E);
end loop;
end Get_Outstanding;

procedure Print_Owing (Self : Invoice) is
  Outstanding : Natural := 0;
begin
  Print_Banner;
  Get_Outstanding (Outstanding);
  Print_Details (Self, Outstanding);
end Print_Owing;
```

Note that the final version of *Print_Owing* is not perfect. For instance, passing the initial value 0 to *Get_Outstanding* is useless, and in fact that should probably be a function with no parameter. But GPS already saves a lot of time and manipulation.

Finally, a word of caution: this refactoring does not check that you are giving a valid input. For instance, if the text you select includes a *declare* block, you should always include the full block, not just a part of it (or select text between *begin* and *end*). Likewise, GPS does not expect you to select any part of the variable declarations, just the code.

3.9 Using an External Editor

GPS is integrated with a number of external editors, in particular *Emacs* and *vi*. The choice of the default external editor is done in the preferences, via *Editor* → *External editor*.

The following values are recognized:

gnuclient This is the recommended client. It is based on Emacs, but needs an extra package to be installed. This is the only client that provides a full integration in GPS, since any extended lisp command can be sent to the Emacs server.

By default, gnuclient will open a new Emacs frame for every file that is opened. You might want to add the following code to your `.emacs` file (create one if needed) so that the same Emacs frame is reused every time:

```
(setq gnuclient-frame (car (frame-list)))
```

See <http://www.hpl.hp.com/personal/ange/gnuclient/home.html> for more information.

emacsclient This is a program that is always available if you have installed Emacs. As opposed to starting a new Emacs every time, it will reuse an existing Emacs session. It is then extremely fast to open a file.

emacs This client will start a new Emacs session every time a file needs to be opened. You should use *emacsclient* instead, since it is much faster, and makes it easier to copy and paste between multiple files. Basically, the only reason to use this external editor is if your system doesn't support *emacsclient*.

vim *Vim* is a vi-like editor that provides a number of enhancements, for instance syntax highlighting for all the languages supported by GPS. Selecting this external editor will start an xterm (or command window, depending on your system) with a running *vim* process editing the file.

Note that one limitation of this editor is that if GPS needs to open the same file a second time, it will open a new editor, instead of reusing the existing one.

To enable this capability, the xterm executable must be found in the PATH, and thus is not supported on Windows systems. Under Windows systems, you can use the *custom* editor instead.

vi This editor works exactly like vim, but uses the standard *vi* command instead of *vim*.

custom You can specify any external editor by choosing this item. The full command line used to call the editor can be specified in the preference *Editor* → *Custom editor command*.

none No external editor is used, and the contextual menus simply won't appear.

In the cases that require an Emacs server, the project file currently used in GPS will be set appropriately the first time Emacs is spawned. This means that if you load a new project in GPS, or modify the paths of the current project, you should kill any running Emacs, so that a new one is spawned by GPS with the appropriate project.

Alternatively, you can reload explicitly the project from Emacs itself by using the menu *Project* → *Load* in emacs (if the ada-mode was correctly installed).

The preference *Editor* → *Always use external editor* lets you chose to always use an external editor every time you double-click on a file, instead of opening GPS' own editor.

3.10 Using the Clipboard

This section concerns X-Window users who are used to cutting and pasting with the middle mouse button. In the GPS text editor, as in many recent X applications, the *GPS clipboard* is set by explicit cut/copy/paste actions, either through menu items or keyboard shortcuts, and the *primary clipboard* (i.e. the 'middle button' clipboard) is set by the current selection.

Therefore, copy/paste between GPS and other X applications using the *primary clipboard* will still work, provided that there is some text currently selected. The *GPS clipboard*, when set, will override the *primary clipboard*.

By default, GPS overrides the X mechanism. To prevent this, add the following line: *OVER-RIDE_MIDDLE_CLICK_PASTE* = *no* to your `traces.cfg` file (typically in `~/ .gps/`). Note, however, that the X mechanism pastes all attributes of text, including coloring and editability, which can be confusing.

See <http://standards.freedesktop.org/clipboards-spec/clipboards-latest.txt> for more information.

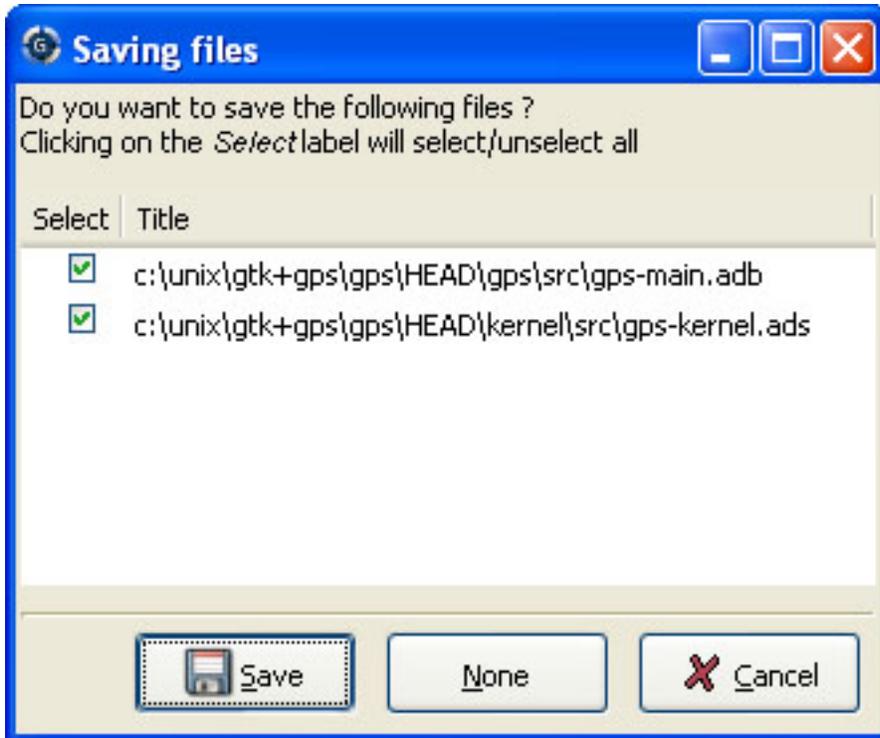
3.11 Saving Files

After you have finished modifying your files, you need to save them. The basic method to do that is to select the menu *File* → *Save*, which saves the currently selected file.

You can also use the menu *File* → *Save As...* if you want to save the file with another name, or in another directory.

If you have multiple files to save, another possibility is to use the menu *File* → *Save More* → *All*. This will open a dialog listing all the currently modified editors that need saving. You can then select individually which one should be saved, and click on *Save* to do the actual saving.

When calling external commands, such as compiling a file, if the *Editor* → *Autosave delay* preference is set to 0, this same dialog is also used, to make sure that e.g. the compiler will take into account your local changes. If the preference is enabled, the saving is performed automatically.



You can conveniently select or unselect all the files at once by clicking on the title of the first column (labeled *Select*). This will toggle the selection status of the first line, and have the same status for all other editors.

If you press *Cancel* instead of *Save*, no saving will take place, and the action that displayed this dialog is also canceled. Such actions can be for instance starting a compilation command, a VCS operation, or quitting GPS with unsaved files.

SOURCE NAVIGATION

4.1 Support for Cross-References

GPS provides cross-reference navigation for program entities, such as types, procedures, functions, variables, ..., defined in your application. The cross-reference support in GPS relies on the compiler generated xref information, which means that you need to compile your project first before being able to navigate. Similarly when your sources have been modified, you need to rebuild and recompute xref information so that your changes are taken into account.

Here are language specific information about source navigation:

Ada The GNAT compiler is used to generate the cross-references information needed by GPS by default, unless you are using the *-gnatD* or *-gnatx* switches, in which case no cross reference information will be available.

If you need to navigate through sources that do not compile (e.g after modifications, or while porting an application), GNAT can still generate partial cross-reference information if you specify the *-gnatQ* compilation option. Along with the *-k* option of *gnatmake*, it is then possible to generate as much relevant information as possible for your non compilable sources.

There are a few special cases where GPS cannot find the external file (called `ALI file`) that contains the cross-reference information. Most likely, this is either because you haven't compiled your sources yet, or because the source code has changed since the `ALI file` was generated.

It could also be that you haven't included in the project the object directories that contain the `ALI files`.

In addition, one special case cannot be handled automatically. This is for separate units, whose file names have been crunched through the `gnatkr` command.

C/C++ The GCC C and C++ compilers that come with GNAT need to be used to generate the cross-references information needed by GPS, via the *-fdump-xref* switch. This means that you need to first add the *-fdump-xref* switch to your project's switches for C and C++ sources, and compile your application before you browse through the cross-references or view various graphs in GPS. If sources have been modified, you should recompile the modified files.

4.1.1 Ada xrefs heuristics

GPS is able to provide some basic navigation support for Ada, C and C++ sources in the absence of information coming from the compiler. It uses a built-in parser, parsing the files at startup and everytime they are modified. This provides basic navigation in simple cases.

In this mode, GPS is able to navigate to an entity body from the declaration, and to an entity declaration from the body. In case of other references, GPS will navigate to the declaration on simple cases, namely if the heuristics provide an information without ambiguity. In particular, it may not work with overloaded entities.

These heuristics are not used in global reference searching operations or call graphs.

Note that this parser is also used to provide the Ada outline view, code completion and entity view.

4.1.2 The xref database

GPS parses the cross-reference information generated by the compiler (the `.ali` and `.gli` files) into an sqlite database. This database can become quite big, and should preferably be on a fast local disk.

By default, GPS will place this database in the object directory of the root project that is currently loaded. You can override this choice by adding an attribute `Xref_Database` in the `IDE` package of your project file. This attribute can be either an absolute path, or a path relative to the location of the project file itself

We recommend that this path be specific to each user, and to each project this user might be working on, as in the following examples:

```
-- assume this is in /home/user1/work/default.gpr
project Default is
  for Object_Dir use "obj";

  package IDE is
    for Xref_Database use "xref_database.db";
    -- This would be /home/user1/work/xref_database.db

    for Xref_Database use Project'Object_Dir & "/xref_database.db";
    -- This would be /home/user1/work/obj/xref_database.db
    -- This is the default when this attribute is not specified

    for Xref_Database use external("HOME") & "/prj1/database.db";
    -- This would be /home/user1/prj1/database.db
  end IDE;
end Default;
```

One of the drawbacks in altering the default location is that **gprclean** will not be able to remove this database when you clean your project.

On the other hand, it might speed up your system if you can put the database on a fast local disk.

4.2 The Navigate Menu

Navigate → **Find or Replace...** Open the find and replace dialog. *Searching and Replacing.*

Navigate → **Find Next** Find next occurrence of the current search. *Searching and Replacing.*

Navigate → **Find Previous** Find previous occurrence of the current search. *Searching and Replacing.*

Navigate → **Find All References** Find all the references to the current entity in the project. The search is based on the semantic information extracted from the sources, this is not a simple text search. The result of the search is displayed in the location window, see *The Locations view*.

Navigate → **Goto declaration** Go to the declaration/spec of the current entity. The current entity is determined by the word located around the cursor. This item is also accessible through the editor's contextual menu directly. This capability requires the availability of cross-reference information. *Support for Cross-References.*

Navigate → **Goto body** Go to the body/implementation of the current entity. If the current entity is the declaration of an Ada subprogram imported from C it goes to the location where the C function is defined. This item is also accessible through the editor's contextual menu directly. This capability requires the availability of cross-reference information. *Support for Cross-References.*

Navigate → **Goto matching delimiter** Go to the delimiter matching the one right before (for a closing delimiter) or right after (for an opening delimiter) the cursor if any.

Navigate → **Goto line** Open a dialog where you can type a line number, in order to jump to a specific location in the current source editor. This feature is also available by clicking on the location at the bottom of editors.

Navigate → **Goto entity** Moves the focus to the *The omni-search* window. You can the enter the name (or part of the name) for any entity defined in your project, and clicking on one of the results will take you to its declaration.

Navigate → **Goto file spec<->body** Open the corresponding spec file if the current edited file is a body file, or body file otherwise. This item is also accessible through the editor's contextual menu

This capability requires support for cross-references. This item is also accessible through the editor's contextual menu

Navigate → **Start of statement** Move the cursor position to the start of the current statement, move to the start of the enclosing statement if the cursor position is already at the start of the statement.

Navigate → **End of statement** Move the current cursor position to the end of the statement, move to the end of the enclosing statement if the cursor position is already at the end of the statement.

Navigate → **Previous subprogram** Move the current cursor position to the start of the previous procedure, function, task, protected record or entry.

Navigate → **Next subprogram** Move the current cursor position to the start of the next procedure, function, task, protected record or entry.

Navigate → **Previous tag** Go to previous tag/location. *The Locations view*.

Navigate → **Next tag** Go to next tag/location. *The Locations view*.

Navigate → **Back** Everytime you use one of the navigation feature in GPS, GPS will first store the current location in a history, and then move the focus to another part of the editor or to another editor. This menu allows you to navigate backward in the history, and thus go to the location you were previously viewing.

Navigate → **Forward** Moves forward in the history of locations.

4.3 Contextual Menus for Source Navigation

This contextual menu is available from any source editor. If you right click on an entity, or first select text, the contextual menu will apply to this selection or entity.

Goto declaration of *entity* Go to the declaration/spec of *entity*. The current entity is determined by the word located around the cursor or by the current selection if any. This capability requires support for cross-references.

Goto declarations of *entity* This contextual menu appears when you are clicking on a subprogram call that is a dispatching call. In such a case, there is no possibility for GPS to know what subprogram will actually be called at run time, since that depends on dynamic information. It therefore gives you a list of all entities in the tagged type hierarchy, and lets you choose which of the declarations you want to jump to. See also the `methods.py` plug-in (enabled by default) which, given an object, lists all its primitive operations in a contextual menu so that you can easily jump to them. See also the contextual menu *References* → *Find References To...* which allows you to find all calls to a subprogram or to one of its overriding subprograms.

Goto full declaration of *entity* This contextual menu appears for a private or limited private types. Go to the full declaration/spec of *entity*. The current entity is determined by the word located around the cursor or by the current selection if any. This capability requires support for cross-references.

Goto type declaration of *entity* Go to the type declaration of *entity*. The current entity is determined by the word located around the cursor or by the current selection if any. This capability requires support for cross-references.

Display type hierarchy for **entity** This contextual menu appears for derived or access types. Output the type hierarchy for *entity* into the location view. The current entity is determined by the word located around the cursor or by the current selection if any. This capability requires support for cross-references.

Goto body of **entity** Go to the body/implementation of *entity*. If *entity* is the declaration of an Ada subprogram imported from C it goes to the the location where the C function is defined. This capability requires support for cross-references.

Goto bodies of **entity** This is similar to *Goto declarations of*, but applies to the bodies of the entities.

Goto file spec/body Open the corresponding spec file if the current edited file is a body file, or body file otherwise. This option is only available for the Ada language.

****Entity** calls** Display a list of all subprograms called by *entity* in a tree view. This is generally more convenient than using the corresponding *Browsers/* submenu if you expect lots of references, *The Call trees view and Callgraph browser*.

****Entity** is called by** Display a list of all subprograms calling *entity* in a tree view. This is generally more convenient than using the corresponding *Browsers/* submenu if you expect lots of references, *The Call trees view and Callgraph browser*.

References → Find all references *Find all references to entity* in all the files in the project.

References → Find all references... This menu is similar to the one above, except it is possible to select more precisely what kind of reference should be displayed. It is also possible to indicate the scope of the search, and whether the context (or caller) at each reference should be displayed.

This dialog has an option *Include overriding and overridden operations*, which, when activated, will include references to overridden or overriding entities of the one you selected.

This is particularly useful when you are wondering whether you can easily modify the profile of a primitive operation or method, since you can then see what other entities will also be impacted. If you select only the *declaration* check box, you will see the list of all related primitive operations.

This dialog also allows you to find out which entities are imported from a given file/unit. Click on any entity from that file (for instance on the *with* line for Ada code), then select the *All entities imported from same file* toggle button. This will display in the location window the list of all entities imported from the same file as the entity selected.

In addition, if you have selected the *Show context* option, you will get a list of all the exact references to these entities within the file. Otherwise, you just get a pointer to the declaration of the imported entities.

References → Find all local references to **entity** *Find all references to entity* in the current file (or in the current top level unit for Ada sources).

References → Variables used in **entity** Find all variables (local or global) used in *entity* and list each first reference in the locations window.

References → Non Local variables used in **entity** Find all non-local variables used in the entity.

References → Methods of **entity** This submenu is only visible if you have activated the plug-in `methods.py` (which is the case by default), and when you click on a tagged type or an instance of a tagged type. This menu lists all the primitive operations or methods of that type, and you can therefore easily jump to the declaration of any of these operations.

Browsers → **Entity calls** Open or raise the call graph browser on the specified entity and display all the subprograms called by *entity*. *Callgraph browser*.

Browsers → **Entity calls (recursively)** Open or raise the call graph browser on the specified entity and display all the subprograms called by *entity*, transitively for all subprograms. Since this can take a long time to compute and generate a very large graph, an intermediate dialog is displayed to limit the number of subprograms to display (1000 by default). *Callgraph browser*.

***Entity* is called by** Open or raise the call graph browser on the specified entity and display all the subprograms calling *entity*. *Callgraph browser*.

Expanded code Present for Ada files only. This menu generates a `.dg` file using your gnat compiler (using the `:index:-gnatGL` switch) and displays the expanded code. This can be useful when investigating low-level issues and tracing precisely how the source code is transformed by the GNAT front-end.

Expanded code → **Show subprogram** Display expanded code for the current subprogram in the current editor.

Expanded code → **Show file** Display expanded code for the current file in the current editor.

Expanded code → **Show in separate editor** Display expanded code for the current file in a new editor.

Expanded code → **Clear** Remove expanded code from the current editor.

For Ada files only, this entry will generate, and will open this file at the location corresponding to the current source line.

Open *filename* When you click on a filename (for instance a C' `#include`, or an error message in a log file), this menu gives you a way to open the corresponding file. If the file name was followed by `":` and a line number, the corresponding line is activated.

4.4 Navigating with hyperlinks

When the `Control` key is pressed and you start moving the mouse, entities in the editors under the mouse cursor become hyperlinks and the mouse cursor aspect changes.

Left-clicking on a reference to an entity will open a source editor on the declaration of this entity, and left-clicking on an entity declaration will open an editor on the implementation of this entity.

Left-clicking on the Ada declaration of a subprogram imported from C will open a source editor on the definition of the corresponding C entity. This capability requires support for cross-references.

Clicking with the middle button on either a reference to an entity or the declaration of an entity will jump directly to the implementation or type completion) of this entity.

Note that for efficiency, GPS may create hyperlinks for some entities which have no associated cross reference. In this case, clicking will have no effect, even though an hyperlink may have been displayed.

This behavior is controlled by the *General* → *Hyper links* preference.

4.5 Highlighting dispatching calls

Dispatching calls in Ada and C++ source code are highlighted by default in GPS via the `dispatching.py` plug-in.

Based on the cross-reference information, this plug-in will highlight (with a special color that you can configure in the preferences dialog) all calls that are dispatching (or calls to virtual methods in C++). A dispatching call, in Ada, is a subprogram call where the actual subprogram that is called is not known until run time, and is chosen based on the tag of the object (so this of course only exists when you are using object-oriented programming).

To disable this highlighting (which might sometimes be slow if you are using big sources, even though the highlighting itself is done in the background), you can go to the *Tools* → *Plug-ins* menu, and disable the `dispatching.py` plug-in.

PROJECT HANDLING

The section on the project view (*The Project view*) has already given a brief overview of what the projects are, and the information they contain.

This chapter provides more in-depth information, and describes how such projects can be created and maintained.

5.1 Description of the Projects

5.1.1 Project files and GNAT tools

This section describes what the projects are, and what information they contain.

The most important thing to note is that the projects used by GPS are the same as the ones used by GNAT. These are text files (using the extension `.gpr`) which can be edited either manually, with any text editor, or through the more advanced GPS interface.

The exact syntax of the project files is fully described in the GNAT User's Guide (`gnat_ug.html`) and GNAT Reference Manual (`gnat_rm.html`). This is recommended reading if you want to use some of the more advanced capabilities of project files which are not yet supported by the graphical interface.

GPS can load any project file, even those that you have been edited manually. Furthermore, you can manually edit project files created by GPS.

Typically you will not need to edit project files manually, since several graphical tools such as the project wizard (*The Project Wizard*) and the properties editor (*The Project Properties Editor*) are provided.

GPS doesn't preserve the layout nor comments of manually created projects after you have edited them in GPS. For instance, multiple case statements in the project will be coalesced into a single case statement. This normalization is required for GPS to be able to preserve the previous semantic of the project in addition to the new settings.

All command-line GNAT tools are project aware, meaning that the notion of project goes well beyond GPS' user interface. Most capabilities of project files can be accessed without using GPS itself, making project files very attractive.

GPS uses the same mechanisms to locate project files as GNAT itself:

- absolute paths
- relative paths. These paths, when used in a with line as described below, are relative to the location of the project that does the with.
- `ADA_PROJECT_PATH`. If this environment variable is set, it contains a colon-separated (or semicolon under Windows) list of directories in which the project files are searched.
- `GPR_PROJECT_PATH`. If this environment variable is set, it contains a colon-separated (or semicolon under Windows) list of directories in which the project files are searched.

- predefined project path. The compiler itself defines a predefined project path, in which standard libraries can be installed, like XML/Ada for instance.

5.1.2 Contents of project files

Project files contain all the information that describe the organization of your source files, object files and executables.

A project file can contain comments, which have the same format as in Ada, that is they start by “--” and extend to the end of the line. You can add comments when you edit the project file manually. GPS will attempt to preserve them when you save the project through the menu, but this will not always be possible. It helps if the comments are put at the end of the line, as in:

```
project Default is
  for Source_Dirs use (); -- No source in this project
end Default;
```

Generally, one project file will not be enough to describe a complex organization. In this case, you will create and use a project hierarchy, with a root project importing other sub projects. Each of the projects and sub projects is responsible for its own set of sources (compiling them with the appropriate switches, put the resulting files in the right directories, ...).

Each project contains the following information (see the GNAT user’s guide for the full list)

- **List of imported projects:** .. index:: project; imported project

When you are compiling sources from this project, the builder will first make sure that all the imported projects have been correctly recompiled and are up-to-date. This way, dependencies between source files are properly handled.

If one of the source files of project A depends on some source files from project B, then B must be imported by A. If this isn’t the case, the compiler will complain that some of the source files cannot be found.

One important rule is that each source file name must be unique in the project hierarchy (i.e. a file cannot be under control of two different projects). This ensures that the same file will be found no matter what project is managing the source file that uses

- **List of source directories:** .. index:: project; source directory

All the sources managed by a project are found in one or more source directories. Each project can have multiple source directories, and a given source directory might be shared by multiple projects.

- **Object directory:** .. index:: project; object directory

When the sources of the project are compiled, the resulting object files are put into this object directory. There exist exactly one object directory for each project. If you need to split the object files among multiple object directories, you need to create multiple projects importing one another as appropriate.

When sources from imported sub-projects are recompiled, the resulting object files are put in the sub project’s own object directory, and will never pollute the parent’s object directory.

- **Exec directory:** .. index:: project; exec directory

When a set of object files is linked into an executable, this executable is put in the exec directory of the project file. If this attribute is unspecified, the object directory is used.

- **List of source files:** .. index:: project; source files

The project is responsible for managing a set of source files. These files can be written in any programming languages. Currently, the graphical interface supports Ada, C and C++.

The default to find this set of source files is to take all the files in the source directories that follow the naming scheme (see below) for each language. In addition if you edit the project file manually, it is possible to provide an explicit list of source files.

This attribute cannot be modified graphically yet.

- **List of main units:** .. index:: project; main units

The main units of a project (or main files in some languages) are the units that contain the main subprogram of the application, and that can be used to link the rest of the application.

The name of the file is generally related to the name of the executable.

A given project file hierarchy can be used to compile and link several executables. GPS will automatically update the Compile, Run and Debug menu with the list of executables, based on this list.

- **Naming schemes:** .. index:: project; naming schemes

The naming scheme refers to the way files are named for each languages of the project. This is used by GPS to choose the language support to use when a source file is opened. This is also used to know what tools should be used to compile or otherwise work with a source file.

- **Embedded targets and cross environments:** .. index:: project; cross environment

GPS supports cross environment software development: GPS itself can run on a given host, such as GNU/Linux, while compilations, runs and debugging occur on a different remote host, such as Sun/Solaris.

GPS also supports embedded targets (VxWorks, ...) by specifying alternate names for the build and debug tools.

The project file contains the information required to log on the remote host.

- **Tools:** Project files provide a simple way to specify the compiler and debugger commands to use.

- **Switches:** .. index:: project; switches

Each tool that is used by GPS (compiler, pretty-printer, debugger, ...) has its own set of switches. Moreover, these switches may depend on the specific file being processed, and the programming language it is written in.

5.2 Supported Languages

Another information stored in the project is the list of languages that this project knows about. GPS support any number of language, with any name you choose. However, advanced support is only provided by default for some languages (Ada, C and C++), and you can specify other properties of the languages through customization files (*Adding support for new languages*).

By default, the graphical interface will only give you a choice of languages among the ones that are known to GPS at that point, either through the default GPS support or your customization files. But you can also edit the project files by hand to add support for any language.

Languages are a very important part of the project definition. For each language, you should specify a naming scheme that allows GPS to associate files with that language. You would for instance specify that all `.adb` files are Ada, all `.txt` files are standard text files, and so on.

Only the files that have a known language associated with them are displayed in the *Project View*, or available for easy selection through the *File* → *Open From Project* menu. Similarly, only these files are shown in the Version Control System interface.

It is therefore important to properly setup your project to make these files available conveniently in GPS, although of course you can still open any file through the *File* → *Open* menu.

If your project includes some README files, or other text files, you should add “txt” as a language (or any other name you want), and make sure that these files are associated with that language in the *Project* → *Edit project properties*.

By default, GPS provides support for a number of languages. In most cases, this support takes the form of syntax highlighting in the editor, and possibly the Outline View. Other languages have advanced cross-references available.

All the supported languages can be added to the project, but you can also add your own languages as you need (either by editing the project files by hand, or by creating XML files to add GPS support for these languages, which will then show in the project properties editor graphically).

5.3 Scenarios and Configuration Variables

The behavior of projects can be further tailored by the use of scenarios.

All the attributes of a project, except its list of imported projects, can be chosen based on the value of external variables, whose value is generally coming from the host computer environment, or directly set in GPS. The interface to manipulate these scenarios is the scenario view, which can be displayed by selecting the menu *Tools* → *Views* → *Scenario* (*The Scenario View*). It can be convenient to drag this window with your mouse, and drop it above the project view, so that you can see both at the same time.

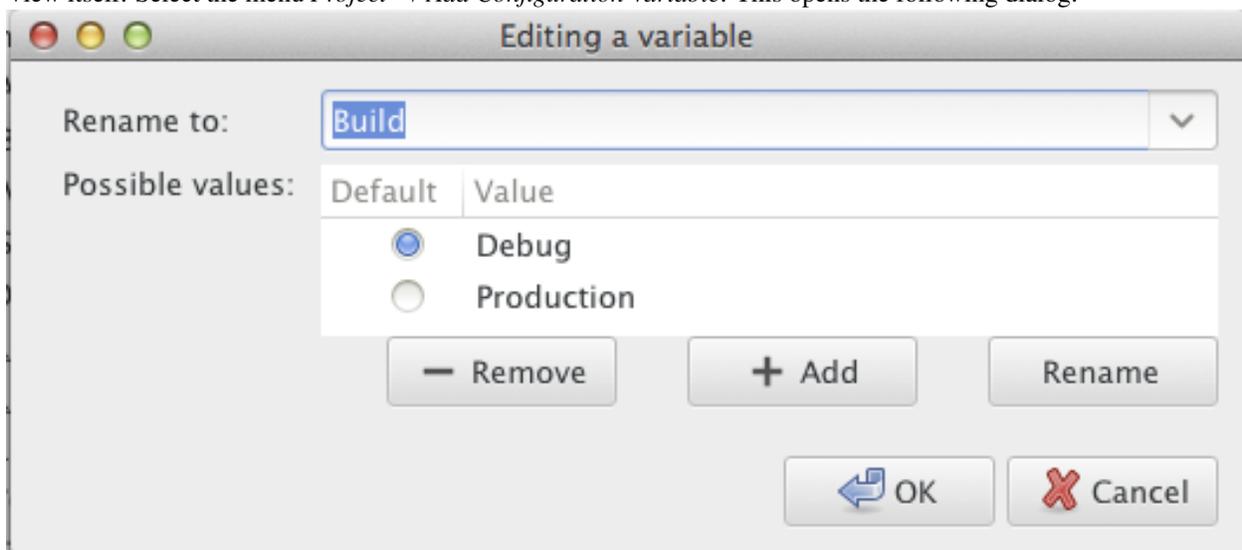
This area allows you to select new values for the scenario variables defined in your project, and thus change dynamically the view GPS has of your project and your source files.

This facility can for instance be used to compile all the sources either in debug mode (so that the executables can be run in the debugger), or in optimized mode (to reduce the space and increase the speed when delivering the software). In this configuration scenario, all the attributes (source directories, tools, ...) remain the same, except for the compilation switches. It would be more difficult to maintain a completely separate hierarchy of project, and it is much more efficient to create a new configuration variable and edit the switches for the appropriate scenario (*The Project Properties Editor*).

There is one limitation in what GPS can do with scenario variables: although gnatmake and gprbuild have no problem dealing with scenario variables whose default value is not a static string (for instance a concatenation, or the value of another scenario variable), GPS will not be able to edit such a project graphically. Such projects will load fine in GPS though.

5.3.1 Creating new scenario variables

Creating a new scenario variable is done through the contextual menu (right-click) in the Project View or the Scenario View itself. Select the menu *Project* → *Add Configuration Variable*. This opens the following dialog:



There are two main areas in this dialog: in the top line, you specify the name of the variable. This name is used for two purposes:

- It is displayed in the *Scenario* view
- This is the name of the environment variable from which the initial value is read. When GPS is started, all configuration variables are initialized from the host computer environment, although you can of course change its value later on inside GPS. Note that selecting a new value for the scenario variable does not change the actual value of the environment variable, which is only used to get the default initial value of the scenario variable.

When you spawn external tools like gnatmake for instance, you can also specify the value they will use for the scenario variable by using a command line switch, typically *-X*.

If you click on the arrow on the right of this name area, GPS will display the list of all the environment variables that are currently defined. However, you don't need to pick the name of an existing variable, neither must the variable exist when GPS is started.

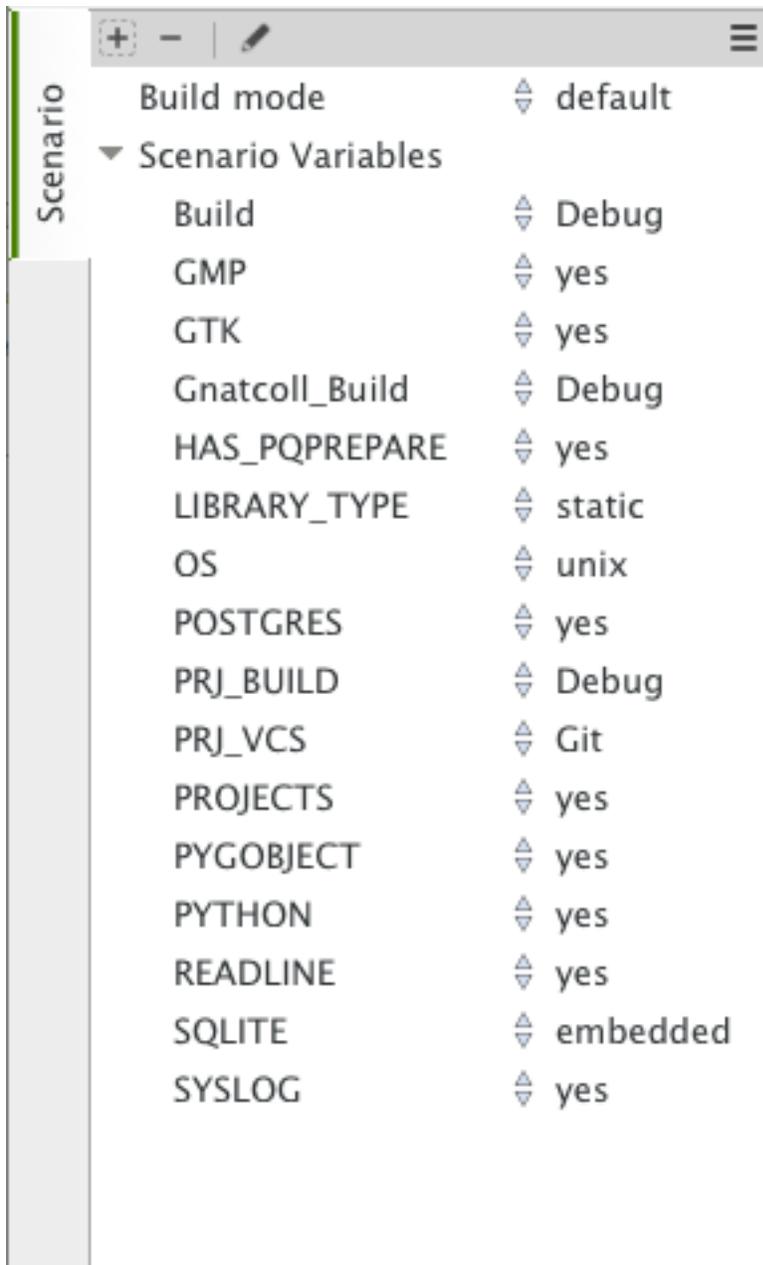
The second part of this dialog is the list of authorized value for this variable. Any other value will generate an error reported by GPS, and the project won't be loaded as a result.

One of these values is the default value (the one whose button in the Default column is selected). This means that if the environment variable doesn't exist when GPS is started, GPS will behave as if it did exist with this default value.

The list of possible values can be edited by right-clicking on the name of the variable, and selecting one of *Edit properties* or *Delete variable*.

5.3.2 Editing existing scenario variables

If at least one configuration variable is defined in your project, the *Scenario* view will contain something similar to:



You can easily change the current value of any of these variables by clicking on the value. This will display a pop-up window with the list of possible values, from which you select the one you wish to use.

As soon as a new value is selected, GPS will recompute the project view (in case source directories, object directories or list of source files have changed). A number of things will also be updated (like the list of executables in the *Compile*, *Run* and *Debug* menus).

Currently, GPS will not recompute the contents of the various browsers (call graph, dependencies, ...) for this updated project. This would be too expensive to do every time the value changes, and therefore you need to explicitly request an update.

You can change the list of possible values for a configuration variable at any time by clicking on the *edit* button in the local toolbar. This will pop up the same dialog that is used to create new variables. This dialog also allows you to change the name of the scenario variable. This name is the same as the environment variable that is used to set the initial value of the scenario variable.

Removing a variable is done by clicking the *remove* the button in the local toolbar, after selecting the variable. GPS will then display a confirmation dialog.

If you confirm that you want to delete the variable, GPS will simply remove the variable, and from now on act as if the variable always had the value it had when it was deleted.

5.4 Extending Projects

5.4.1 Description of extending projects

The project files were designed to support big projects, with several hundreds or thousands of source files. In such contexts, one developer will generally work on a subset of the sources. It is also not rare for such a project to take several hours to fully compile. Most developers therefore do not need to have the full copy of the project compiled on their own machine or personal disk space.

However, it is still useful to be able to access other source files of the application, for instance to find out whether a subprogram can be changed and where it is currently called.

Such a setup can be achieved through extending projects. These are special types of projects that inherit most of their attributes and source files from another project, and can have, in their source directories, some source files that hide/replace those inherited from the original project.

When compiling such projects, the compiler will put the newly created project files in the extending project's directory, and will leave the original untouched. As a result, the original project can be shared read-only among several developers (for instance, it is usual for this original project to be the result of a nightly build of the application).

5.4.2 Creating extending projects

This project wizard allows you to easily create extending projects. You should select an empty directory (which will be created automatically if needed), as well as a list of source files you want to work on initially. New files can also be added later.

As a result, GPS will copy the selected source files to the new directory (if you so decided), and create a number of project files there. It will then load a new project, which has the same properties as the previous one, except that some files are found transparently in the new directory, and object files resulting from the compilation are create into that directory as opposed to the object directory of the original project.

5.4.3 Adding files to extending projects

Once you have loaded an extending project in GPS, things work mostly transparently. If you open a file through the *File* → *Open From Project* dialog, the files found in the local directory of your extending project will be picked up first.

The build actions will create object files in the extending project's directory, leaving the original project untouched.

It might happen that you want to start working on a source file that you had not added in the extending project when it was created. You can of course edit the file found in the original project, provided you have write access to it. However, it is generally better to edit it in the context of the extending project, so that the original project can be shared among developers.

This can be done by clicking on the file in the *Project* view, then selecting the menu *Add To Extending Project*. This will popup a dialog asking whether you want GPS to copy the file to the extending project's directory for you. GPS might also create some new project files in that directory if necessary, and automatically reload the project as needed. From then on, if you use the menu *File* → *Open From Project*, GPS will first see the file from the extending project.

Note that open editors will still be editing the same file they were before, so you should open the new file if needed.

5.5 Disabling Project Edition Features

The project files should generally be considered as part of the sources, and thus be put under control of a version control system. As such, you might want to prevent accidental editing of the project files, either by you or some other person using the same GPS installation.

The main thing to do to prevent such accidental edition is to change the write permissions on the project files themselves. On Unix systems, you could also change the owner of the file. When GPS cannot write a project file, it will report an error to the user.

However, the above doesn't prevent a user from trying to do some modifications at the GUI level, since the error message only occurs when trying to save the project (this is by design, so that temporary modification can be done in memory).

You can disable all the project editing related menus in GPS by adding special startup switches. The recommended way is to create a small batch script that spawns GPS with these switches. You should use the following command line:

```
gps --traceoff=MODULE.PROJECT_VIEWER --traceoff=MODULE.PROJECT_PROPERTIES
```

What these do is prevent the loading of the two GPS modules that are responsible for project edition. However, this also has an impact on the python functions that are exported by GPS, and thus could break some plug-ins. Another solution which might apply is simply to hide the corresponding project-editing menus and contextual menus. This could be done by creating a small python plugin for GPS (*Customizing through XML and Python files*, which contains the following code:

```
import GPS
GPS.Menu.get('/Project/Edit Project Properties').hide()
GPS.Contextual('Edit project properties').hide()
GPS.Contextual('Save project').hide()
GPS.Contextual('Add configuration variable').hide()
```

5.6 The Project Menu

The menu bar item *Project* contains several commands that generally act on the whole project hierarchy. If you only want to act on a single project, use the contextual menu in the project view.

Some of these menus apply to the currently selected project. This notion depends on what window is currently active in GPS: if it is the project view, the selected project is either the selected node (if it is a project), or its parent project (for a file, directory, ...). If the currently active window is an editor, the selected project is the one that contains the file.

In all cases, if there is no currently selected project, the menu will apply to the root project of the hierarchy.

These commands are:

Project → **New** This menu will open the project wizard (*The Project Wizard*), so that you can create new project. On exit, the wizard asks whether the newly created project should be loaded. If you select *Yes*, the new project will replace the currently loaded project hierarchy.

You will get asked what information you would like to create the project from. In particular, you can create a set of project files from existing Ada sources.

Project → **New from template** This menu will open the project template wizard, allowing you to create a new project using one of the project templates defined in GPS. *Adding project templates*.

Project → **Open** This menu opens a file selection dialog, so that any existing project can be loaded into GPS. The newly loaded project replaces the currently loaded project hierarchy. GPS works on a single project hierarchy at a time.

Project → **Recent** This menu can be used to easily switch between the last projects that were loaded in GPS.

Project → **Edit Project Properties** This menu applies to the currently selected project, and will open the project properties dialog for this project.

Project → **Save All** This will save all the modified projects in the hierarchy.

Project → **Edit File Switches** This menu applies to the currently selected project. This will open a new window in GPS, listing all the source files for this project, along with the switches that will be used to compile them, *The Switches Editor*.

Project → **Reload project** Reload the project from the disk, to take into account modifications done outside of GPS. In particular, it will take into account new files added externally to the source directories. This isn't needed for modifications made through GPS.

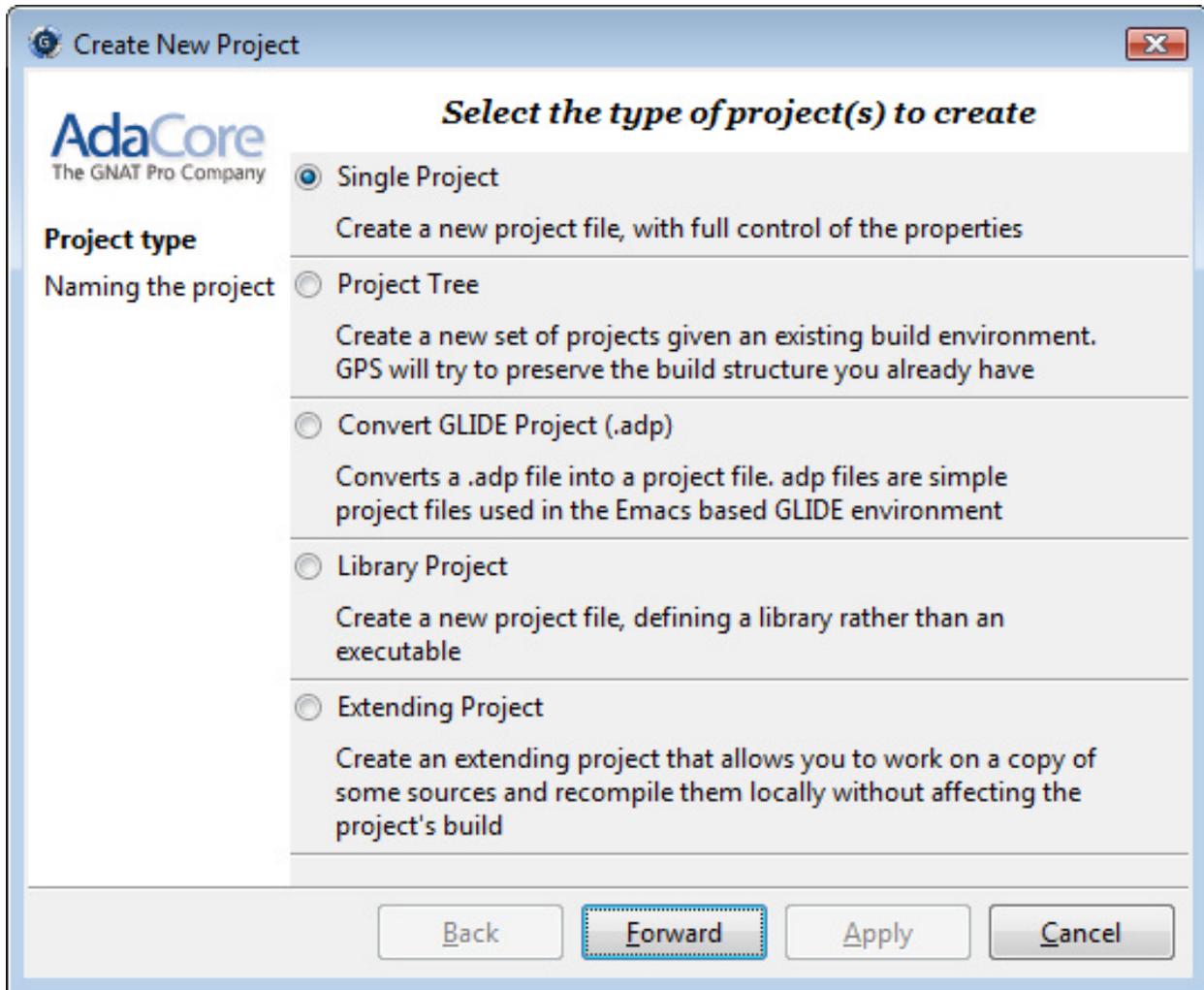
Project → **Project View** Open (or raise if it is already open) the project view on the left side of the GPS window.

5.7 The Project Wizard

The project wizard allows you to create in a few steps a new project file. It has a number of pages, each dedicated to editing a specific set of attributes for the project.

The typical way to access this wizard is through the *Project* → *New...* menu.

The project wizard is also launched when a new dependency is created between two projects, through the contextual menu in the project view.



The wizard gives access to the following list of pages:

- Project type
- Project Naming
- Languages Selection
- Version Control System Selection
- Source Directories Selection
- Build Directory
- Main Units
- Library
- Naming Scheme
- Switches

5.7.1 Project Type

Several types of project wizards are provided in GPS. Depending on the information you have or your current setup, you will choose one or the other.

- Single Project

This is likely the wizard you will use most often. It creates a project file from scratch, and asks you for the location of source directories, the object directory, ...; The rest of this chapter describes this wizard in more details

- Project Tree

This wizard will attempt to create a set of one or more project files to represent your current build environment. It will analyze what your sources are, where the corresponding object files are, and will try and find some possible setup for the project files (remember that a given `.gpr` project file can be associated with a single object directory).

This wizard might not work in all cases, but is worth a try to get you started if you already have an existing set of sources

- Convert GLIDE Project (.adp)

This wizard will help you convert a `.adp` project file that is used by the GLIDE environment. The same restrictions apply as above, except that the list of source directories, object directories and tool switches are read directly from that file.

- Library Project .. index:: project; library

This specialized wizard is similar to the Single Project wizard, except it adds one extra page, the Library page. The output of the compilation of this project is a library (shared or static), as opposed to an executable in the case of *Single Project*.

- Extending Project .. index:: project; extending

This specialized wizard allows you to easily create extending projects (*Extending Projects*).

5.7.2 Project Naming

This is the first page displayed by any of the wizard.

You must enter the name and location of the project to create. This name must be a valid Ada identifier (i.e. start with a letter, optionally followed by a series of digits, letters or underscores). Spaces are not allowed. Likewise, reserved Ada keywords must be avoided. If the name is invalid, GPS will display an error message when you press the *Forward* button.

Child projects can be created from this dialog. These are project whose name is of the form *Parent.Child*. GPS will automatically generate the dependency to the parent project so as to make the child project valid.

In this page, you should also select what languages the source files in this project are written in. Currently supported languages are *Ada*, *C* and *C++*. Multiple languages can be used for a single project.

The last part of this page is used to indicate how the path should be stored in the generated project file. Most of the time, this setting will have no impact on your work. However, if you wish to edit the project files by hand, or be able to duplicate a project hierarchy to another location on your disk, it might be useful to indicate that paths should be stored as relative paths (they will be relative to the location of the project file).

5.7.3 Languages Selection

This page is used to select the programming languages used for the sources of this project. By default, only *Ada* is selected. New languages can be added to this list by using XML files, see the section on customizing GPS (*Adding support for new languages*).

Additionally, this page allows you to select the toolchain used when working on your project. There you can select one of the pre-defined toolchains or scan your system for installed toolchains. You can also manually define some of the tools in the toolchain such as the debugger to use, the gnat driver to use or the gnatls tool to use.

If you need to select a toolchain for a cross environment, you should have a look at *Working in a Cross Environment* for more info on this subject.

5.7.4 VCS Selection

The second page in the project wizard allows you to select which Version Control system is to be used for the source files of this project.

GPS doesn't attempt to automatically guess what it should use, so you must specify it if you want the VCS operations to be available to you.

The two actions *Log checker* and *File checker* are the name and location of programs to be run just prior an actual commit of the files in the Version Control System. These should be used for instance if you wish to enforce style checks before a file is actually made available to other developers in your team.

If left blank, no program will be run.

5.7.5 Source Directories Selection

This page lists and edits the list of source directories for the project. Any number of source directory can be used (the default is to use the directory which contains the project file, as specified in the first page of the wizard).

If you do not specify any source directory, no source file will be associated with the project, since GPS wouldn't know where to look for them.

To add source directories to the project, select a directory in the top frame, and click on the down arrow. This will add the directory to the bottom frame, which contains the current list of source directories.

You can also add a directory and all its subdirectories recursively by using the contextual menu in the top frame. This contextual menu also provides an entry to create new directories, if needed.

To remove source directories from the project, select the directory in the bottom frame, and click on the up arrow, or use the contextual menu.

All the files in these directories that match one of the language supported by the project are automatically associated with that project.

The relative sizes of the top and bottom frame can be changed by clicking on the separation line between the two frames and dragging the line up or down.

5.7.6 Build Directory

The object directory is the location where the files resulting from the compilation of sources (e.g. `.o` files) are placed. One object directory is associated for each project.

The exec directory is the location where the executables are put. By default, this is the same directory as the object directory.

5.7.7 Main Units

The main units of a project are the files that should be compiled and linked to obtain executables.

Typically, for C applications, these are the files that contain the *main()* function. For Ada applications, these are the files that contain the main subprogram each partition in the project.

These files are treated specially by GPS. Some sub-menus of *Build* and *Debug* will have predefined entries for the main units, which makes it more convenient to compile and link your executables.

To add main units click on the *Add* button. This opens a file selection dialog. No check is currently done that the selected file belongs to the project, but GPS will complain later if it doesn't.

When compiled, each main unit will generate an executable, whose name is visible in the second column in this page. If you are using a recent enough version of GNAT (3.16 or more recent), you can change the name of this executable by clicking in the second column and changing the name interactively.

5.7.8 Library

This page allows you to configure your project so that the output of its compilation is a library (shared or static), as opposed to an executable or a simple set of object files. This library can then be linked with other executables (and will be automatically if the project is imported by another one).

You need to define the attributes in the top box to transform your project into a library project. See the tooltips that appear when you leave your mouse on top of the label to the left of each field.

If you define any of the attributes in the Standalone Library box, you will compile a standalone library. This is a library that takes care of its elaboration by itself, instead of relying on its caller to elaborate it as is standard in Ada. You also have more control over what files make up the public interface to the library, and what files are private to the library and invisible from the outside.

5.7.9 GNATname

This page allows you to add Ada units stored in files with irregular or arbitrary naming conventions into your project. To take advantage of this ability you need to specify file name patterns. GPS will use these patterns to search for Ada units in each of source directories specified in *Source Directories Selection* page. Then GPS utilises gnatname tool to generate the required pragmas for a set of files. Files with arbitrary naming convention are not compatible with naming scheme customization, so next page will be skipped.

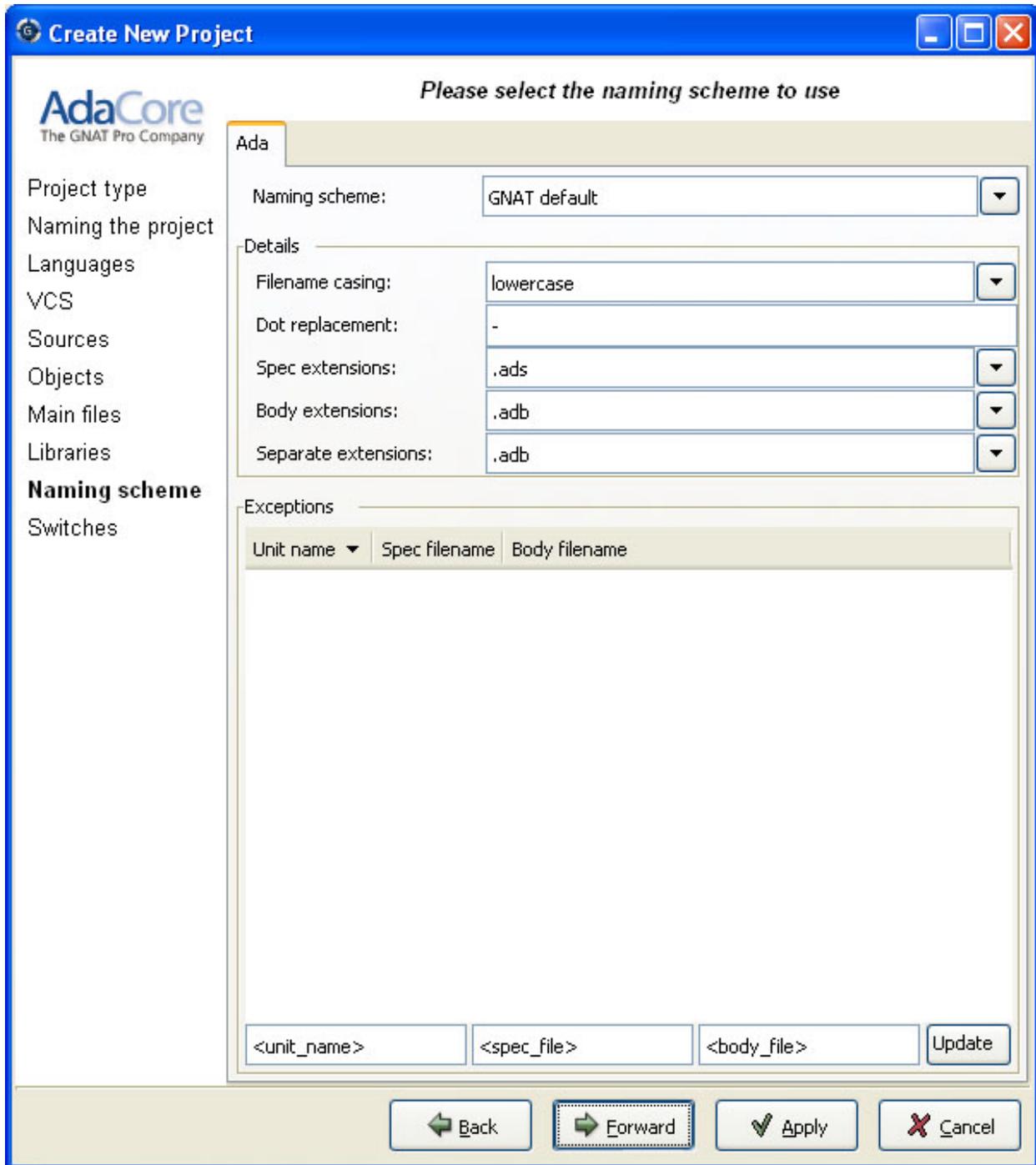
5.7.10 Naming Scheme

A naming scheme indicates the file naming conventions used in the different languages supported by a given project. For example, all `.adb` files are Ada files, all `.c` files are C files.

GPS is very flexible in this respect, and allows you to specify the default extension for the files in a given programming language. GPS makes a distinction between spec (or header) files, which generally contain no executable code, only declarations, and body files which contain the actual code. For languages other than Ada, this header file is used rather than the body file when you select *Go To Declaration* in the contextual menu of editors.

In a language like Ada, the distinction between spec and body is part of the definition of the language itself, and you should be sure to specify the appropriate extensions.

The default naming scheme for Ada is GNAT's naming scheme (`.ads` for specs and `.adb` for bodies). In addition, a number of predefined naming schemes for other compilers are available in the first combo box on the page. You can also create your own customized scheme by entering a free text in the text entries.



For all languages, GPS accepts exceptions to this standard naming scheme. For instance, this let you specify that in addition to using `.adb` for Ada body files, the file `foo.ada` should also be considered as an Ada file.

The list of exceptions is displayed in the bottom list of the naming scheme editor. To remove entries from this list, select the line you want to remove, and then press the `Del` key. The contents of the lines can be edited interactively, by double-clicking on the line and column you want to edit.

To add new entries to this list, use the fields at the bottom of the window, and press the update button.

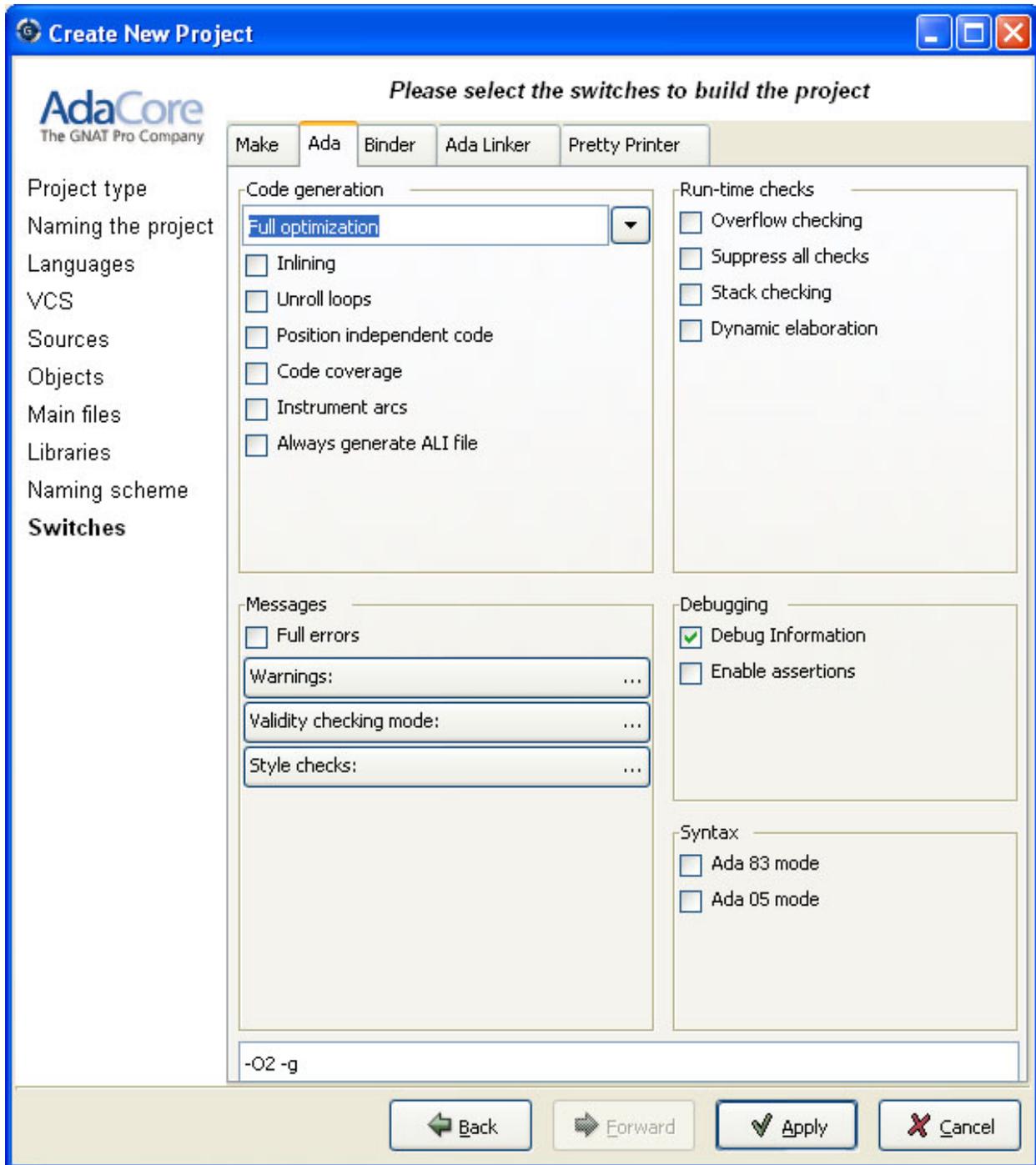
GNAT and GPS both support Ada source files that contain multiple Ada units (typically a single file would contain both the spec and the body of the unit for instance). This is not a recommend approach if you can avoid it, since that

might trigger unnecessary recompilation of your source files. Such source files are always handled as naming scheme exceptions, and you can specify those in the editor by adding *at 1*, *at 2*,... after the file name for either the spec, the body or both. The digit after *at* is the index (starting at 1) of the unit in the source file.

For instance, specifying *file.ada at 1* for the spec and *file.ada at 2* for the body of the unit “unit” indicates that the two components of the unit are in the same file, first the spec, followed by the body.

5.7.11 Switches

The last page of the project wizard is used to select the default switches to be used by the various tools that GPS calls (compiler, linker, binder, pretty printer, ...).



This page appears as a notebook, where each page is associated with a specific tool. All these pages have the same structure:

Graphical selection of switches The top part of each page contains a set of buttons, combo boxes, entry fields, ... which give fast and intuitive access to the most commonly used switches for that tool.

Textual selection of switches The bottom part is an editable entry field, where you can directly type the switches. This makes it easier to move from an older setup (e.g. Makefile, script) to GPS, by copy-pasting switches.

The two parts of the pages are kept synchronized at any time: clicking on a button will edit the entry field to show the new switch; adding a new switch by hand in the entry field will activate the corresponding button if there is one.

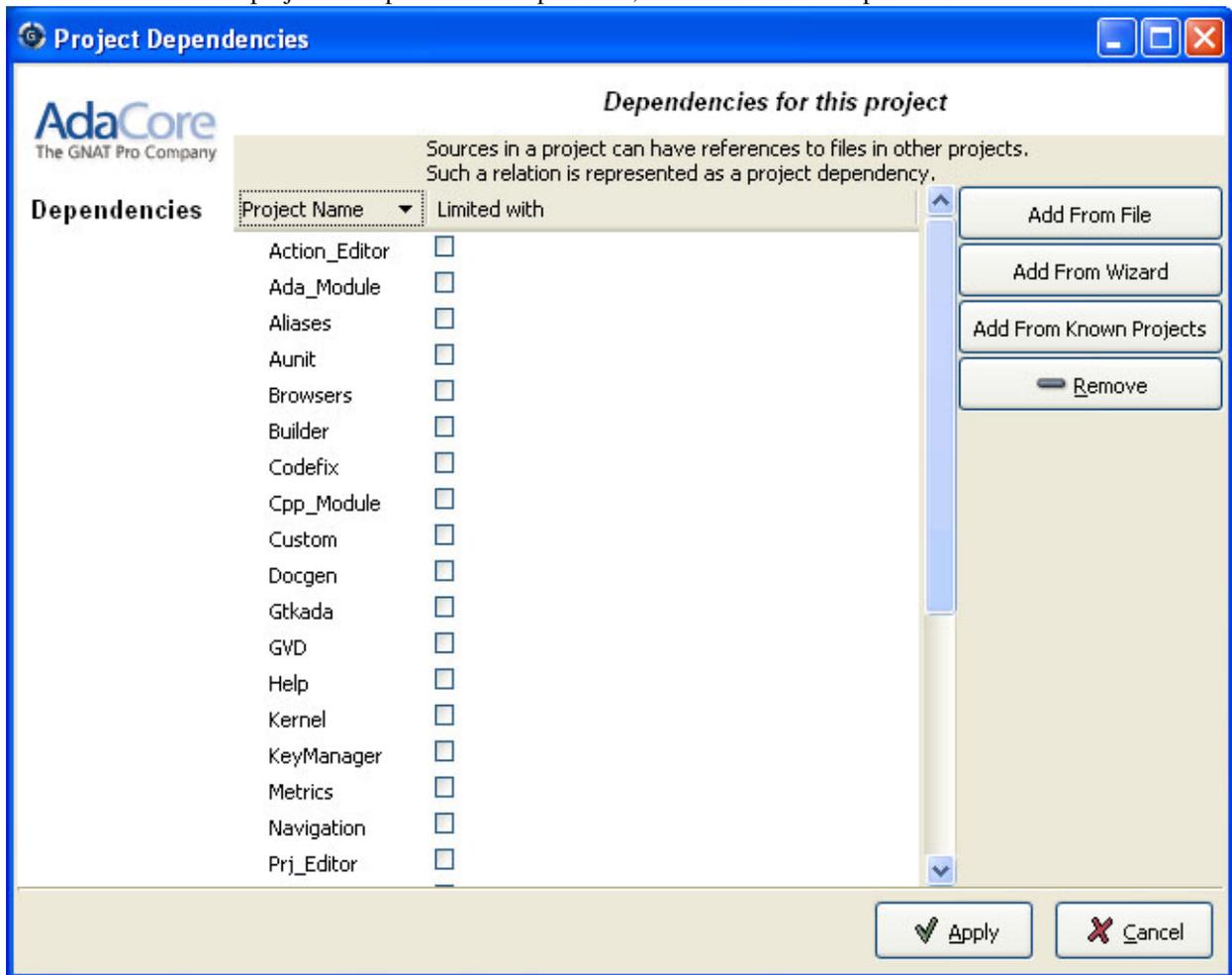
Any switch can be added to the entry field, even if there is no corresponding button. In this case, GPS will simply forward it to the tool when it is called, without trying to represent it graphically.

5.8 The Project Dependencies Editor

You can edit the dependencies between projects through the contextual menu *Project* → *Dependencies...* in the Project View.

This view makes it easy to indicate that your project depends on external libraries, or other modules in your source code. For instance, you can give access to the GtkAda graphical library in your project by adding a project dependency to `gtkada.gpr`, assuming GtkAda has been installed in your system.

The dependencies also determine in what order your application is built. When you compile a project, the builder will first make sure that the projects it depends on are up-to-date, and otherwise recompile them.



When you select that contextual menu, GPS will open a dialog that allows you to add or remove dependencies to your project. New dependencies are added by selecting a project file name from one of several sources:

- One of the loaded project from the current project tree
- One of the predefined projects

These are the projects that are found in one of the directories referenced in the `ADA_PROJECT_PATH` environment variable. Typically, these include third party libraries, such as GtkAda, win32ada, ...

- A new project created through the project wizard
- Any project file located on the disk

In all these cases, you will generally be able to choose whether this should be a simple dependency, or a limited dependency. The latter allows you to have mutually dependent projects (A depends on B, which in turns depends on A even indirectly), although you cannot reference the attribute of such a project in the current project (for instance to indicate that the compiler switches to use for A are the same as for B – you need to duplicate that information).

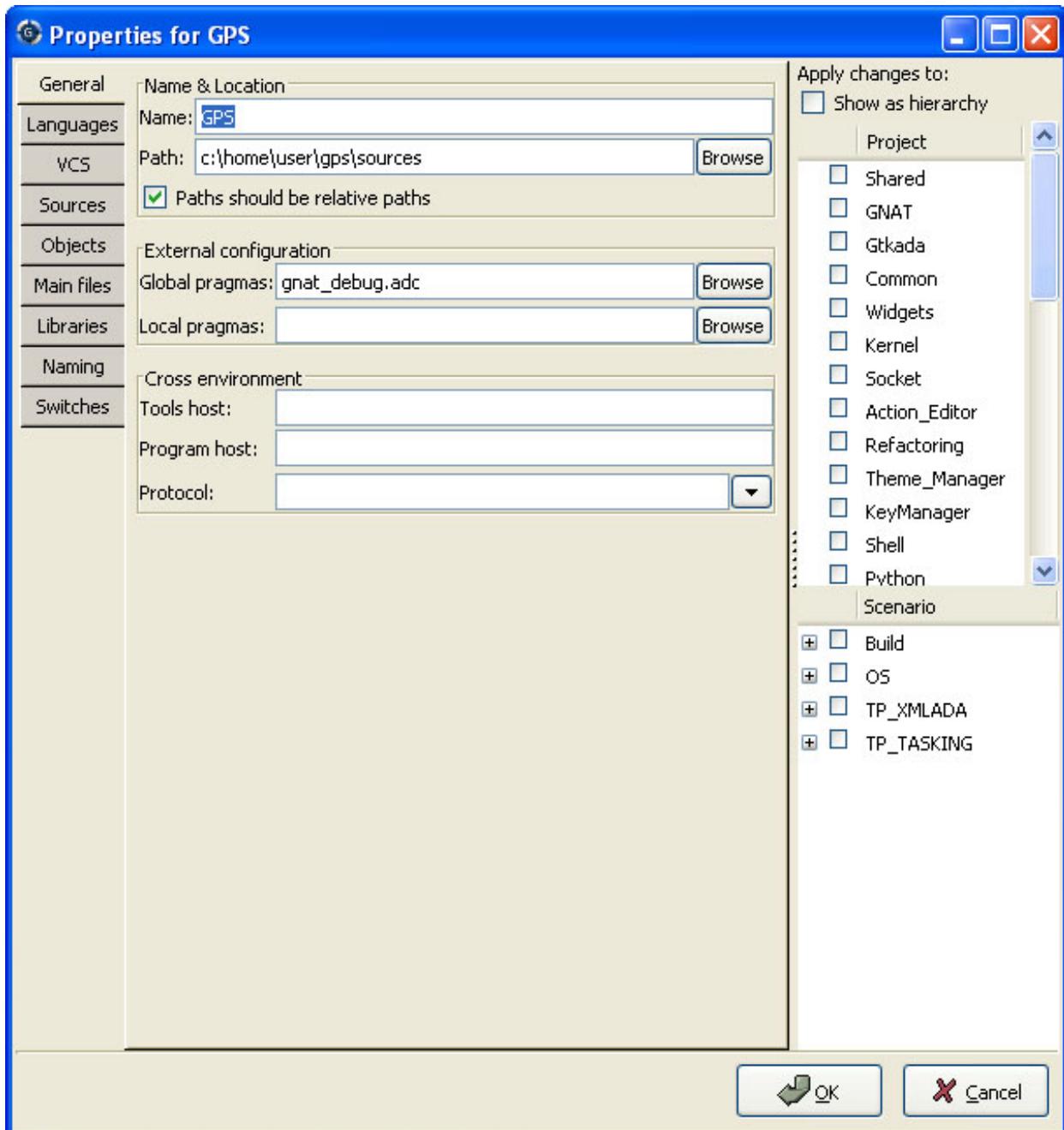
In some cases, GPS will force a limited dependency on you to avoid loops in the dependencies that would make the project tree illegal.

5.9 The Project Properties Editor

The project properties editor gives you access at any time to the properties of your project. It is accessible through the menu *Project* → *Edit Project Properties*, and through the contextual menu *Edit project properties* on any project item, e.g. from the Project View or the Project Browser.

If there was an error loading the project (invalid syntax, non-existing directories, ...), a warning dialog is displayed when you select the menu. This reminds you that the project might be only partially loaded, and editing it might result in the loss of data. In such cases, it is recommended that you edit the project file manually, which you can do directly from the pop-up dialog.

Fix the project file as you would for any text file, and then reload it manually (through the *Project* → *Open...* or *Project* → *Recent* menus).



The project properties editor is divided in three parts:

The attributes editor

The contents of this editor are very similar to that of the project wizard (*The Project Wizard*). In fact, all pages but the *General* page are exactly the same, and you should therefore read the description for these in the project wizard chapter.

See also *Working in a Cross Environment* for more info on the *Cross environment* attributes.

The project selector

This area, in the top-right corner of the properties editor, contains a list of all the projects in the hierarchy. The value in the attributes editor is applied to all the selected projects in this selector. You cannot unselect

the project for which you activated the contextual menu.

Clicking on the right title bar (*Project*) of this selector will sort the projects in ascending or descending order.

Clicking on the left title bar (untitled) will select or unselect all the projects.

This selector has two different possible presentations, chosen by the toggle button on top: you can either get a sorted list of all the projects, each one appearing only once. Or you can have the same project hierarchy as displayed in the project view.

The scenario selector

This area, in the bottom-right corner of the properties editor, lists all the scenario variables declared for the project hierarchy. By selecting some or all of their values, you can choose to which scenario the modifications in the attributes editor apply.

Clicking on the left title bar (untitled, on the left of the *Scenario* label) will select or unselect all values of all variables.

To select all values of a given variable, click on the corresponding check button.

5.10 The Switches Editor

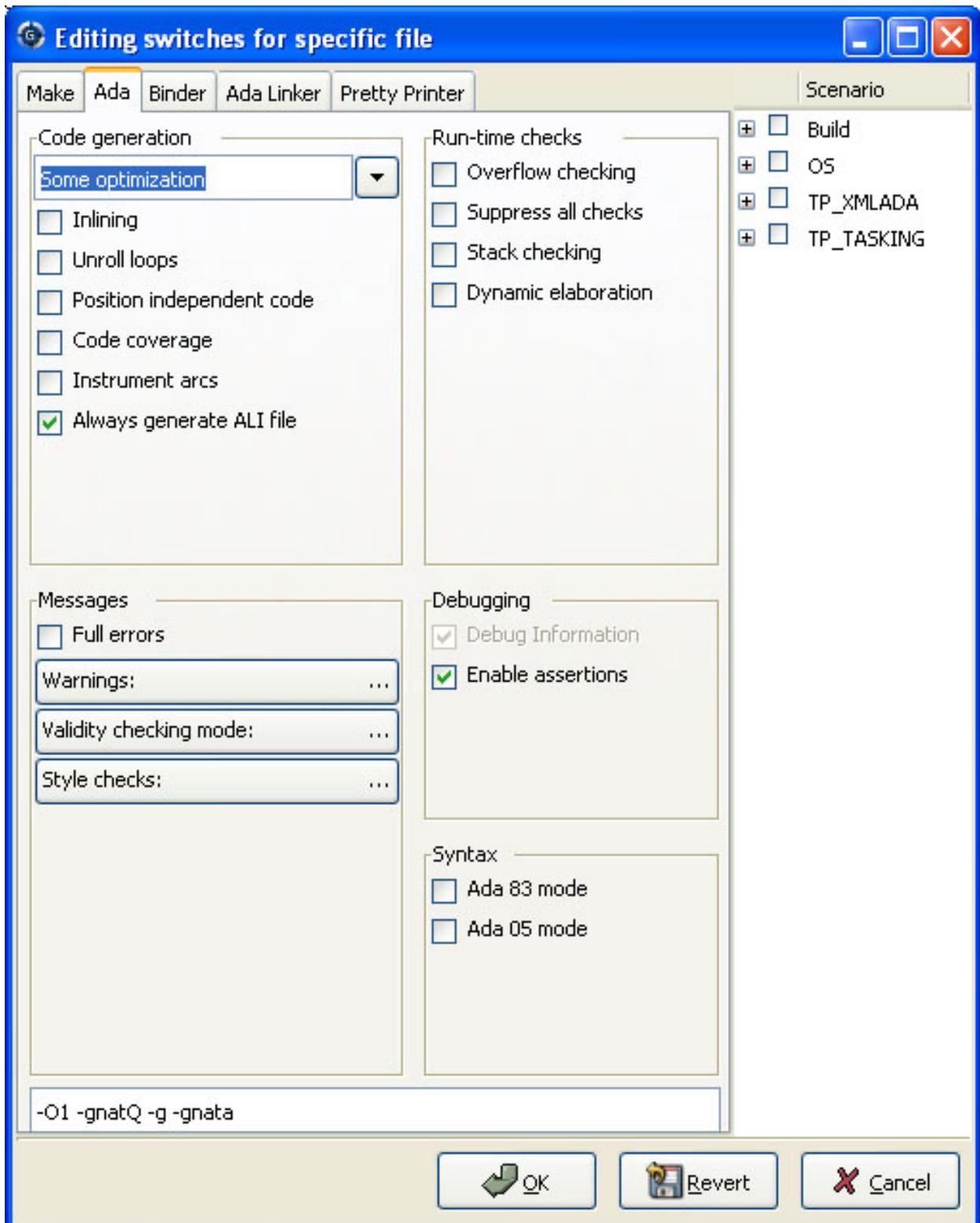
The switches editor, available through the menu *Project* → *Edit Switches*, lists all the source files associated with the selected project.

For each file, the compiler switches are listed. These switches are displayed in gray if they are the default switches defined at the project level (*The Project Properties Editor*). They are defined in black if they are specific to a given file.

Double-clicking in the switches column allows you to edit the switches for a specific file. It is possible to edit the switches for multiple files at the same time by selecting them before displaying the contextual menu (*Edit switches for all selected files*).

When you double-click in one of the columns that contain the switches, a new dialog is opened that allows you to edit the switches specific to the selected files.

This dialog has a button titled *Revert*. Clicking on this button will cancel any file-specific switch, and revert to the default switches defined at the project level.



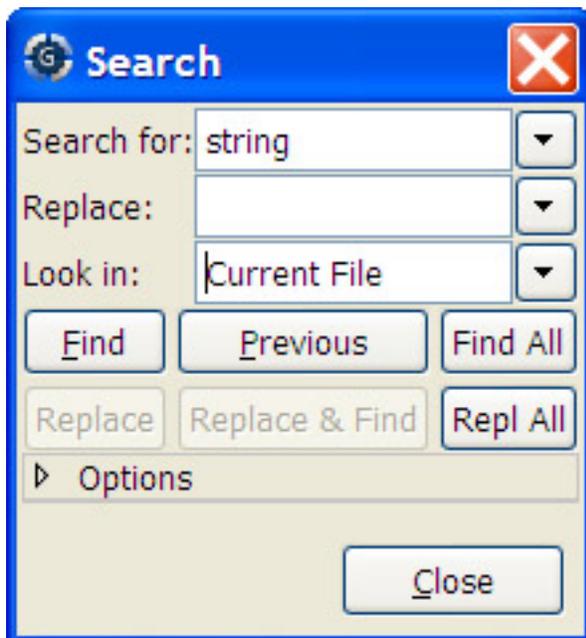
SEARCHING AND REPLACING

GPS provides extensive search capabilities among its different elements. For instance, it is possible to search in the currently edited source file, or in all the source files belonging to the project, even those that are not currently open. It is also possible to search in the project view (on the left side of the main GPS window), ...

All these search contexts are grouped into a single graphical window, that you can open either through the menu *Navigate* → *Find or Replace...*, or the shortcut `Ctrl-F`.

By default, the search window is floating, ie appears as a dialog on top of GPS. You can choose to put it inside the multiple document interface permanently for easier access. This can be done by selecting the menu *Window* → *Floating*, and then drag-and-dropping the search window in a new location if you wish (for instance above the Project View).

Selecting either of these two options will pop up a dialog on the screen, similar to the following:



On this screen shot, you can see three entry fields:

Search for This is the location where you type the string or pattern you are looking for. The search widget supports two modes, either fixed strings or regular expressions. You can commute between the two modes by either clicking on the *Options* button and selecting the appropriate check box, or by opening the combo box (click on the arrow on the right of the entry field).

In this combo box, a number of predefined patterns are provided. The top two ones are empty patterns, that automatically set up the appropriate fixed strings/regular expression mode. The other regular expressions are language-specific, and will match patterns like Ada type definition, C++ method declaration, ...

Replace with This field should contain the string that will replace the occurrences of the pattern defined above. The combo box provides a history of previously used replacement strings. If regular expression is used for search, special escapes *1, 2 .. 9* in this field refer to the corresponding matching sub-expressions and 0 refers whole matched string.

Look in This field defines the context in which the search should occur.

GPS will automatically select the most appropriate context when you open the search dialog, depending on which component currently has the focus. If several contexts are possible for one component (for example, the editor has *Current_File*, *Files from Project*, *Files...* and *Open Files*), then the last one you've been using will be selected. You can of course change the context to another one if needed.

Clicking on the arrow on the right will display the list of all possible contexts. This list includes:

Project View Search in the project view. An extra *Scope* box will be displayed where you can specify the scope of your search, which can be a set of: *Projects*, *Directories*, *Files*, *Entities*. The search in entities may take a long time, search each file is parsed during the search.

Open Files Search in all the files that are currently open in the source editor. The *Scope* entry is described in the *Files...* section below.

Files...

Search in a given set of files. An extra *Files* box will be displayed where you can specify the files by using standard shell (Unix or Windows) regular expression, e.g. **.ad?* for all files ending with *.ad* and any trailing character. The directory specified where the search starts, and the *Recursive search* button whether sub directories will be searched as well.

The *Scope* entry is used to restrict the search to a set of language constructs, e.g. to avoid matching on comments when you are only interested in actual code, or to only search strings and comments, and ignore the code.

Files From Projects

Search in all the files from the project, including files from project dependencies. The *Scope* entry is described in the *Files...* section above.

Files From Current Project

Search in all the files from the currently selected project, defaulting on the root project if there is no project currently selected. The currently selected project might be the one to which the source file belongs (if you are in an editor), or the selected project (if you are in the Project view) for instance. The *Scope* entry is described in the *Files...* section above.

Files From Runtime

Search in all specification files from GNAT runtime library. The *Scope* entry is described in the *Files...* section above.

Current File

Search in the current source editor. The *Scope* entry is described in the *Files...* section above.

Project Browser

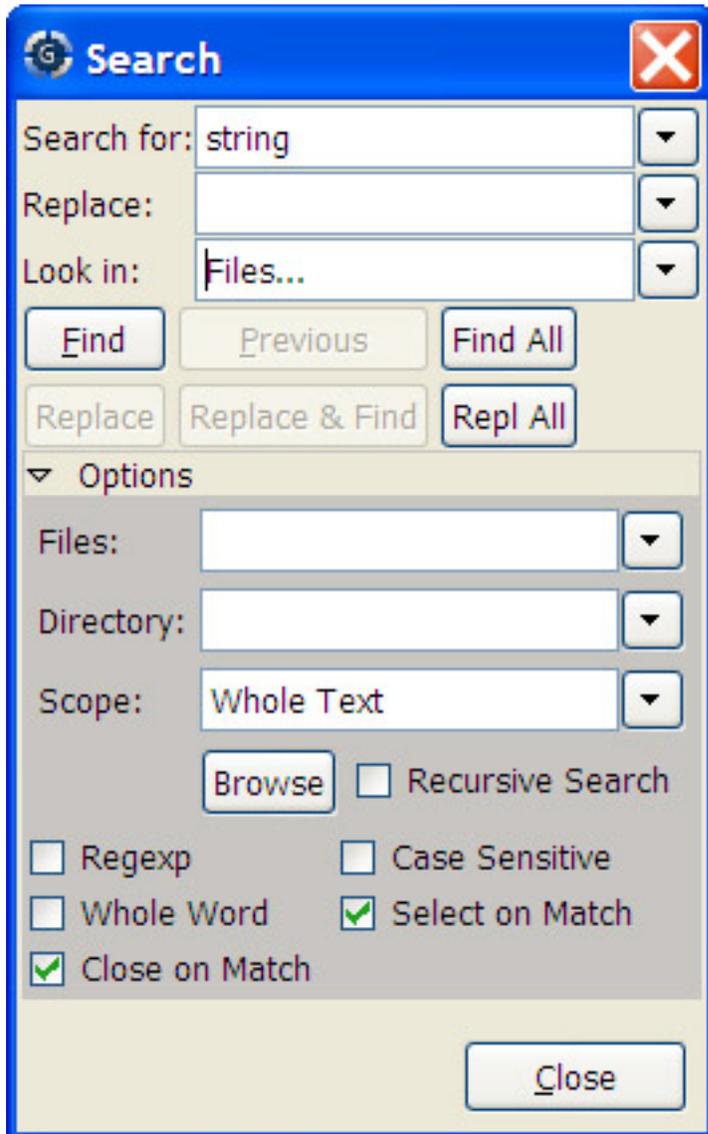
Search in the project browser (*The Project Browser*).

The default value for *Look In* is set through various means: by default, GPS will select a context that matches the currently selected window. For instance, if you are in an editor and open the search dialog, the context will be set to *Current File*. But if the project view is the active window, the context will be set to *Project* view. Optionally,

GPS can remember the last context that was set (see the preference *Search* → *Preserve Search Context*. If this is set, and an editor is selected, GPS will remember whether the last time you started a search from an editor you decided to search in *Current File* or *Files From Project* for instance.

Finally, you can create key shortcuts (through the *Edit* → *Key Shortcuts* menu, in the *Search* category) to open the search dialog and set the context to a specific value.

The second part of the window is a row of buttons, to start the search (or continue to the next occurrence), and to display the options.



There are five check boxes in this options box.

Regexp This button commutes between fixed string patterns and regular expressions. You can also commute between these two modes by selecting the arrow on the right of the *Search for:* field. The grammar followed by the regular expressions is similar to the Perl and Python regular expressions grammar, and is documented in the GNAT run time file `g-regpat.ads`. To open it from GPS, you can use the *open from project* dialog (*File* → *Open From Project...*) and type `g-regpat.ads`.

Whole Word If activated, this check box will force the search engine to ignore substrings. “sensitive” will no longer match “insensitive”.

Select on Match When this button is selected, the focus is given to the editor that contains the match, so that you can start editing the text immediately. If the button is not selected, the focus is left on the search window, so that you can press Enter to search for the next occurrence.

Close on Match This button only appears if the search window is floating. If this button is enabled, the search window will be automatically closed when an occurrence of the search string is found.

Case Sensitive Search By default, patterns are case insensitive (upper-case letters and lower-case letters are considered as equivalent). You can change this behavior by clicking on this check box.

Case Preserving Replace When this is checked, replacements preserve casing. Three casings are detected and preserved: all lower, all UPPER, and Mixed_Case where the first character of each word is capitalized. Note that when the replace pattern is not all lower case, replacement is never case-preserving, the original casing of the replace pattern is used.

Pressing the *Find / Previous* buttons performs an interactive search. It stops as soon as one occurrence of the pattern is found. Once a first occurrence has been found, the *Find* button is renamed to *Next*. You then have to press the *Next* button (or the equivalent shortcut `Ctrl-N`) to go to the next occurrence.

If you use the *Find all* button, the search widget will start searching for all occurrences right away, and put the results in a new window called *Locations*, *The Locations view*.

The *Replace* and *Replace & Find* buttons are grayed out as long as no occurrence of the pattern is found. In order to enable them, you have to start a search, e.g. by pressing the *Find* button. Pressing *Replace* will replace the current occurrence (and therefore the two buttons will be grayed out), and *Replace & Find* will replace the occurrence and then jump to the next one, if any. If you don't want to replace the current occurrence, you can jump directly to the next one by pressing *Next*.

The *Repl all* button will replace all the occurrences found. By default, a popup is displayed and ask for confirmation. It's possible to disable this popup by either checking the box "Do not ask this question again", or by going in the Search panel of the preferences pages, and unchecking "Confirmation for 'Replace all'". The confirmation popup can be reenabled through this checkbox.

As most GPS components, the search window is under control of the multiple document interface, and can thus be integrated into the main GPS window instead of being an external window.

To force this behavior, open the menu *Window* → *Search* in the list at the bottom of the menu, and then select either *Window* → *Floating* or *Window* → *Docked*.

If you save the desktop (*File* → *Save More* → *Desktop*), GPS will automatically reopen the search dialog in its new place when it is started next time.

COMPILATION/BUILD

This chapter describes how to compile files, build executables and run them. Most capabilities can be accessed through the *Build* menu item, or through the *Build* and *Run* contextual menu items, as described in the following section.

When compiler messages are detected by GPS, an entry is added in the *Locations View*, allowing you to easily navigate through the compiler messages (see *The Locations view*), or even to automatically correct some errors or warnings (see *Code Fixing*).

Compiler messages also appear as icons on the side of lines in the source editors. When the mouse pointer is left on these icons, a tooltip appears, listing the error messages found on this line. When GPS is capable of automatically correcting the errors, clicking on the icon will apply the fix to the source code. The icons on the side of editors are removed when the corresponding entries are removed from *The Locations view*.

7.1 The Build Menu

The build menu gives access to capabilities related to checking, parsing and compiling files, as well as creating and running executables. note that this menu is fully configurable via the *Targets* dialog, so what is documented in this manual are the default menus (see *Build* → *Settings* → *Targets* below).

Build → **Check syntax** Check the syntax of the current source file. Display an error message in the *Messages* window if no file is currently selected.

Build → **Check semantic** Check the semantic of the current source file. Display an error message in the *Messages* window if no file is currently selected.

Build → **Compile file** Compile the current file.

By default, will display an intermediate dialog where you can add extra switches, or simply press `Enter` to get the standard (or previous) switches. Display an error message in the *Messages* window if no file is selected.

If errors or warnings occur during the compilation, the corresponding locations will appear in the *Locations View*. If the corresponding Preference is set, the source lines will be highlighted in the editors (see *The Preferences Dialog*). To remove the highlighting on these lines, remove the files from the *Locations View* using either the contextual menu (*Remove category*) or by closing the *Locations View*.

Build → **Project** → **<main>** The menu will list of all mains defined in your project hierarchy. Each menu item will build the selected main.

Build → **Project** → **Build All** Build and link all main units defined in your project. If no main unit is specified in your project, build all files defined in your project and subprojects recursively. For a library project file, compile sources and recreate the library when needed.

Build → **Project** → **Compile All Sources** Compile all source files defined in the top level project.

Build → **Project** → **Build <current file>** Consider the currently selected file as a main file, and build it.

Build → **Project** → **Custom build** Display a text entry where you can enter any external command. This menu is very useful when you already have existing build scripts, make files, ... and want to invoke them from GPS. If the *SHELL* environment variable is defined (to e.g. */bin/sh*), then the syntax used to execute the command is the one for this shell. Otherwise, the command will be spawned directly by GPS without any shell interpretation.

Build → **Clean** → **Clean all** Remove all object files and other compilation artifacts associated to all projects related to the current one. It allows to restart a complete build from scratch.

Build → **Clean** → **Clean root** Remove all object files and other compilation artifacts associated to the root project. It does not clean objects from other related projects.

Build → **Makefile** If you have the *make* utility in your PATH, and have a file called `Makefile` in the same directory as your project file is, or if you've set the *makefile* property in the *Make* section of the project properties (see [The Project Properties Editor](#)), this menu will be displayed, giving access to all the targets defined in your makefile.

Build → **Ant** If you have the *ant* utility in your PATH, and have a file called `build.xml` in the same directory as your project file is, or if you've set the *antfile* property in the *Ant* section of the project properties (see [The Project Properties Editor](#)), this menu will be displayed, giving access to all the targets defined in your ant file.

Build → **Run** → **<main>** For each main source file defined in your top level project, an entry is listed to run the executable associated with this main file. Running an application will first open a dialog where you can specify command line arguments to your application, if needed. You can also specify whether the application should be run within GPS (the default), or using an external terminal.

When running an application from GPS, a new execution window is added in the bottom area where input and output of the application is handled. This window is never closed automatically, even when the application terminates, so that you can still have access to the application's output. If you explicitly close an execution window while an application is still running, a dialog window will be displayed to confirm whether the application should be terminated.

When using an external terminal, GPS launches an external terminal utility that will take care of the execution and input/output of your application. This external utility can be configured in the preferences dialog ([External Commands](#) → [Execute command](#)).

The GPS execution windows have several limitations compared to external terminals. In particular, they do not handle signals like `ctrl-z` and `control-c`. In general, if you are running an interactive application, we strongly encourage you to run in an external terminal.

Similarly, the *Run* contextual menu accessible from a project entity contains the same entries.

Build → **Run** → **Custom...** Similar to the entry above, except that you can run any arbitrary executable. If the *SHELL* environment variable is defined (to e.g. */bin/sh*), then the syntax used to execute the command is the one for this shell. Otherwise, the command will be spawned directly by GPS without any shell interpretation.

Build → **Settings** → **Targets** This opens the Target Configuration Dialog. [The Target Configuration Dialog](#).

Build → **Settings** → **Toolchains** Open a dialog allowing the configuration of GPS for working with two compilation toolchains. This is particularly useful when compiling a project with an old compiler, while wanting up-to-date functionalities from the associated tools (`gnatmetric`, `gnatcheck` and so on). [Working with two compilers](#).

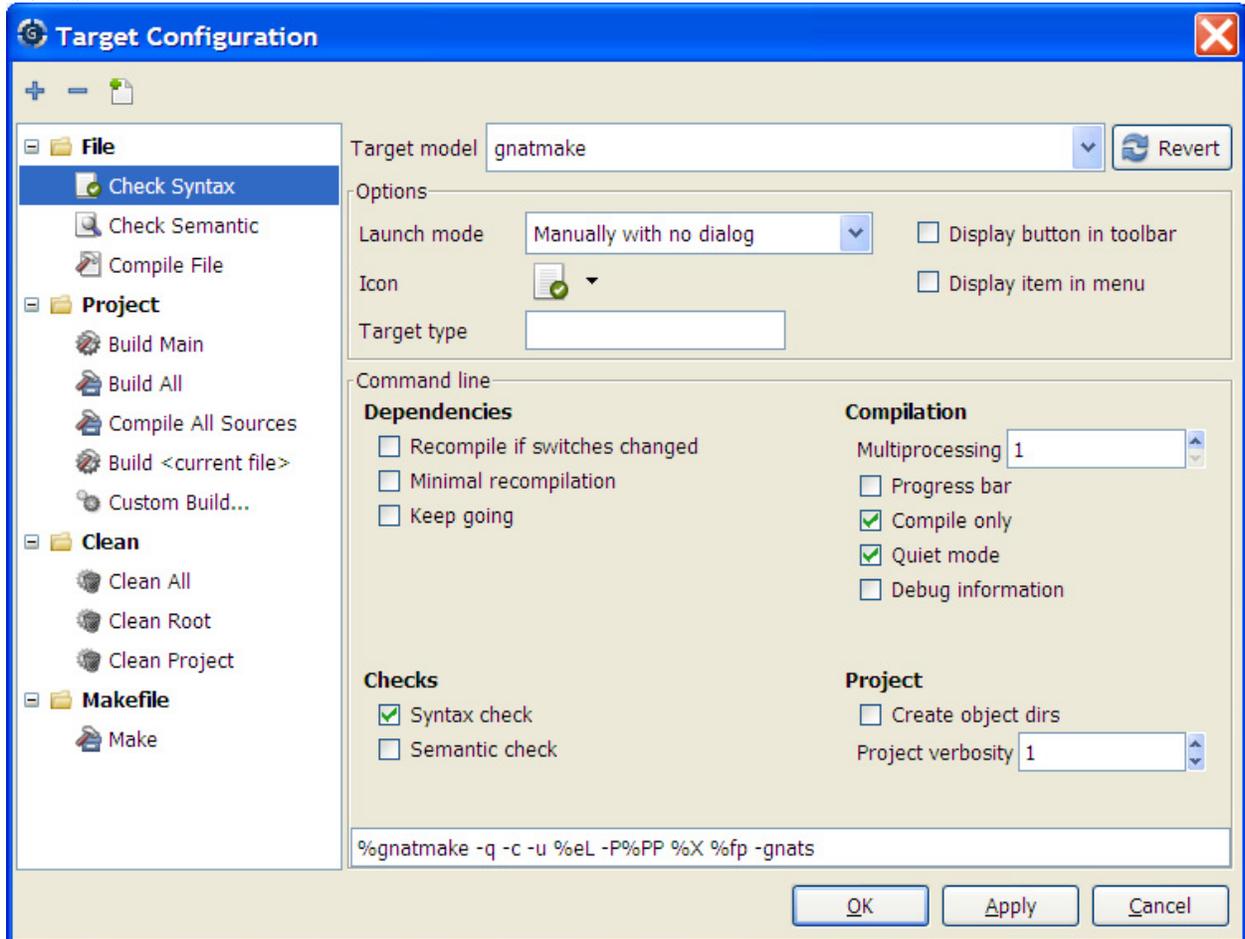
Tools → **Interrupt** This menu can be used to interrupt the last compilation or run command. Once you have interrupted that last operation, you can interrupt the previous one by selecting the same menu again.

Tools → **Views** → **Tasks** However, the easiest way to interrupt a specific operation, no matter if it was started last or not, is to use the *Task Manager*, through the **Tools** → **Views** → **Tasks** menu. It will show one line per running process, and right-clicking on any of these lines gives the possibility to interrupt that process.

If your application is build through a Makefile, you should probably load the `makefile.py` startup script (see the menu **Tools** → **Plug-ins**).

7.2 The Target Configuration Dialog

GPS provides an interface for launching operations like building projects, compiling individual files, performing syntax or semantic checks, and so on. All these operations have in common that they involve launching an external command, and parsing the output for error messages. In GPS, these operations are called “Targets”, and can be configured either through the Target Configuration dialog, or through XML configuration. *Customizing build Targets and Models.*



This dialog is divided in two areas: on the left, a tree listing Targets, and, in the main area, a panel for configuring the Target which is currently selected in the tree.

7.2.1 The Targets tree

The Tree contains a list of targets, organized by categories.

On top of the tree are three buttons:

- The Add button creates a new target.
- The Remove button removes the currently selected target. Note that only user-defined targets can be removed, the default targets created by GPS cannot be removed.
- The Clone button creates a new user-defined target which is identical to the currently selected target.

7.2.2 The configuration panel

On top of the configuration panel, one can select the Target model. The Model determines the graphical options available in the *Command line* frame.

The *Revert* button resets all target settings to their original value.

The *Options* frame contains a number of options that are available for all Targets.

- The Launch mode indicates the way the target is launched:
 - Manually: the target is launched when clicking on the corresponding icon in the toolbar, or when activating the corresponding menu item. In the latter case, a dialog is displayed, allowing last-minute modifications of the command line.
 - Manually with dialog: same as Manually, but the dialog is always displayed, even when clicking on the toolbar icon.
 - Manually with no dialog: same as Manually, but the dialog is never displayed, even when activating the menu item.
 - On file save: the Target is launched automatically by GPS when a file is saved. The dialog is never displayed.
 - In background: the Target is launched automatically in the background after each modification in the source editor. See *Background compilations* below.
- Icon: the icon to use for representing this target in the menus and in the toolbar. To use one of your icons, you must register a icons using the `<stock>` XML customization node. (*Adding stock icons*). Then, use “custom” choice and enter in the text field the ID of the icon.
- Target type: type of target described. If empty, or set to *Normal*, represents a simple target. If set to another value, represents multiple subtargets. For example, if set to *main*, each subtarget corresponds to a Main source as defined in the currently loaded project. Other custom values may be defined, and then handled via the `compute_build_targets` hook.

The *Display* frame indicates where the launcher for this target should be visible.

- in the toolbar: when active, a button is displayed in the main toolbar, allowing to quickly launch a Target.
- in the main menu: whether to display a menu item corresponding to the Target in the main GPS menu. By default, Targets in the “File” category are listed directly in the Build menu, and Targets in other categories are listed in a submenu corresponding to the name of the category.
- in contextual menus for projects: whether to display an item in the contextual menu for projects in the Project View
- in contextual menus for files: whether to display an item in the contextual menus for files, for instance in file items in the Project View or directly on source file editors.

The *Command line* contains a graphical interface for some configurable elements of the Target, which are specific to the Model of this Target.

The full command line is displayed at the bottom. Note that it may contain Macro Arguments. For instance if the command line contains the string “%PP”, GPS will expand this to the full path to the current project. For a full list of available Macros, see *Macro arguments*.

7.2.3 Background compilations

GPS is capable of launching compilation targets in the background. This means that GPS will launch the compiler on the current state of the file in the editor.

Error messages resulting from background compilations are not listed in the Locations view or the Messages window. The full messages are listed in the Background Build console, accessible from the menu *Tools* → *Consoles* → *Background Builds*. Error messages which contain a source location indication are shown as icons on the side of lines in editors, and the exact location is highlighted directly in the editor. On both of these places, tooltips show the contents of the error messages.

Messages from background compilations are removed automatically either when a new background compilation has finished, or when a non-background compilation is launched.

GPS will launch background compilations for all targets that have a *Launch mode* set to *In background*, after modifications occur in a source editor. Background compilation is useful mostly for targets such as *Compile File* or *Check Syntax*. For targets that work on Mains, the last main that was used in a non-background is considered, defaulting to the first main defined in the project hierarchy.

Background compilations are not launched while GPS is already listing results from non-background compilations, ie as long as there are entries in the Locations View showing entries in the *Builder results* category.

7.3 The Build Mode

GPS provides an easy way to build your project with different options, through the mode selection, located in the *Scenario* view (*Scenario view*).

When the mode is set to *default*, the build is done using the switches defined in the project. When the mode is set to another value, then specialized parameters are passed to the builder. For instance, the *gcov* Mode adds all the compilation parameters needed to instrument the produced objects and executables to work with the *gcov* tool.

In addition to changing the build parameters, the mode has the effect of changing the output directory for objects and executables. For instance, objects produced under the *debug* mode will be located in the *debug* subdirectories of the object directories defined by the project. This allows switching from one Mode to another without having to erase the objects pertaining to a different Mode.

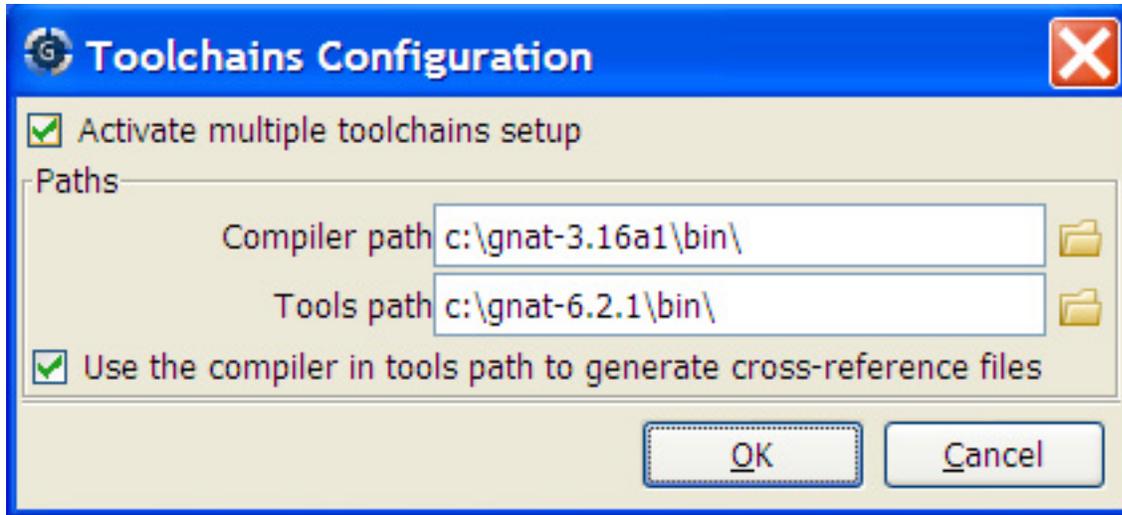
It is possible to define new modes using XML customization, see *Customizing build Targets and Models*.

Note that the Build Mode affects only builds done using recent versions of *gnatmake* and *gprbuild*. The Mode selection has no effect on builds done through Targets that launch other builders.

7.4 Working with two compilers

This functionality is intended for people whose projects need to be compiled with a specific (old) version of the GNAT toolchain, while still desiring to take full advantage of up-to-date associated tools for non-compilation actions, such as checking the code against a coding standard, getting better cross-reference browsing in GPS, computing metrics and so on.

GPS now allows you to handle this case. To configure GPS to make it handle two compiler toolchains, you need to use the *Build* → *Settings* → *Toolchains* menu. This will open a dialog where you can activate the multiple-toolchains mode.



In this dialog, two paths need to be configured: the compiler path and the tools path. The first one is used to actually compile the code, while the second one is used to run up-to-date tools to get more functionalities or accurate results.

Note that GPS will only enable the *OK* button when the two paths are set to different location, since otherwise it does not make sense to enable the multiple toolchains set up.

From this dialog, you can also activate an automated cross-reference generation. The cross-reference files are the *.ali* files generated by the GNAT compiler together with the compiled object. Those files are used by GPS for several functionalities, such as cross-reference browsing or documentation generation. Having those *.ali* files produced by a recent compiler helps having more accurate results with those functionalities, but might interact badly with an old compiler also reading those *.ali* files for compiling a project.

If the automated xref generation is activated, then GPS will generate those *.ali* files using the compiler found in the tools path, and place them in a directory distinct from the one used by the actual compiler. This allows GPS to take full benefit of up-to-date cross-reference files, while keeping the old toolchain happy as its *.ali* files remain untouched.

Note that the cross-reference files generation does not output anything in the “Messages” window, so as to not confuse the output of the regular build process. If needed, you can see the output of the cross-ref generation command by selecting the *Tools* → *Consoles* → *Auxiliary Builds* menu.

7.4.1 Interaction with the remote mode

The ability to work with two compilers has impacts on the remote mode configuration: paths defined here are local paths, so they have no meaning on the server side.

To handle the case of using a specific compiler version on the remote side while still wanting up-to-date tools, the following behavior is applied when both a remote compilation server is defined, and the multiple toolchains mode is activated:

- The compiler path is ignored when a remote build server is defined. All compilation actions are then performed normally on the build server.
- The tools path is however taken into account, and all related actions are performed on the local machine using this path.
- The cross-reference files are taken care of by the rsync mechanism so that they don't get overwritten during local and remote host synchronisations, as build and cross-reference generation actions occur at the same time, on the local machine and on the distant server.

DEBUGGING

GPS is also a graphical front-end for text-based debuggers such as GDB. A knowledge of the basics of the underlying debugger used by GPS will help understanding how GPS works and what kind of functionalities it provides.

Please refer to the debugger-specific documentation - e.g. the GNAT User's Guide (chapter *Running and Debugging Ada Programs*) or the GDB documentation for more details.

Debugging is tightly integrated with the other components of GPS. For example, it is possible to edit files and navigate through your sources while debugging.

To start a debug session, go to the menu *Debug* → *Initialize*, and choose either the name of your executable, if you have specified the name of your main program(s) in the project properties, or start an empty debug session using the *<no main file>* item. It is then possible to load any file to debug, by using the menu *Debug* → *Debug* → *Load File...*

Note that you first need to build your executable with debug information (*-g* switch), either explicitly as part of your project properties, or via the *Debug* build mode (see *The Build Mode* for more details).

Note that you can create multiple debuggers by using the *Debug* → *Initialize* menu several times: this will create a new debugger each time. All the debugger-related actions (e.g. stepping, running) are performed on the current debugger, which is represented by the current debugger console. To switch between debuggers, simply select its corresponding console.

After the debugger has been initialized, you have access to two new windows: the data window (in the top of the working area), and the debugger console (in a new page, after the *Messages* and *Shell* windows). All the menus under *Debugger* are now also accessible, and you also have access to additional contextual menus, in particular in the source editor where it is possible to easily display variables, set breakpoints, and get automatic display (via tooltips) of object values.

When you want to quit the debugger without quitting GPS, go to the menu *Debug* → *Terminate Current*, that will terminate your current debug session, or the menu *Debug* → *Terminate* that will terminate all your debug sessions at once.

8.1 The Debug Menu

The *Debug* entry in the menu bar provides operations that act at a global level. Key shortcuts are available for the most common operations, and are displayed in the menus themselves. Here is a detailed list of the menu items that can be found in the menu bar:

Debug* → *Run... Opens a dialog window allowing you to specify the arguments to pass to the program to be debugged, and whether this program should be stopped at the beginning of the main subprogram. If you confirm by clicking on the *OK* button, the program will be launched according to the arguments entered.

Debug* → *Step Execute the program until it reaches a different source line.

Debug → **Step Instruction** Execute the program for one machine instruction only.

Debug → **Next** Execute the program until it reaches the next source line, stepping over subroutine calls.

Debug → **Next Instruction** Execute the program until it reaches the next machine instruction, stepping over subroutine calls.

Debug → **Finish** Continue execution until selected stack frame returns.

Debug → **Continue** Continue execution of the program being debugged.

Debug → **Interrupt** Asynchronously interrupt the program being debugged. Note that depending on the state of the program, you may stop it in low-level system code that does not have debug information, or in some cases, not even a coherent state. Use of breakpoints is preferable to interrupting programs. Interrupting programs is nevertheless required in some situations, for example when the program appears to be in an infinite (or at least very time-consuming) loop.

Debug → **Terminate Current** Terminate the current debug session by terminating the underlying debugger (e.g. *gdb*) used to handle the low level debugging. You can control what happens to the windows through the *Debugger* → *Debugger Windows* preference.

Debug → **Terminate** Terminate all your debug sessions. Same as *Debug* → *Terminate Current* if there is only one debugger open.

8.1.1 Initialize

This menu contains one entry per main unit defined in your project, which will start a debug session and load the executable associated with the main unit selected and if relevant, all corresponding settings: a debug session will open the debug perspective and associated debug properties (e.g. saved breakpoints, and data display).

Debug → *Initialize* → <No Main File>

Will initialize the debugger with no executable. You can then use one of the other menu items like *Debug* → *Debug* → *Load File* or *Debug* → *Debug* → *Attach*.

8.1.2 Debug

Debug → *Debug* → *Connect to board*

Opens a simple dialog to connect to a remote board. This option is only relevant to cross debuggers.

Debug → **Debug** → **Load File...** Opens a file selection dialog that allows you to choose a program to debug. The program to debug is either an executable for native debugging, or a partially linked module for cross environments (e.g. VxWorks).

Debug → **Debug** → **Add Symbols** Add the symbols from a given file/module. This corresponds to the *gdb* command *add-symbol-file*. This menu is particularly useful under VxWorks targets, where the modules can be loaded independently of the debugger. For instance, if a module is independently loaded on the target (e.g. using windshell), it is absolutely required to use this functionality, otherwise the debugger won't work properly.

Debug → **Debug** → **Attach...** Instead of starting a program to debug, you can instead attach to an already running process. To do so, you need to specify the process id of the process you want to debug. The process might be busy in an infinite loop, or waiting for event processing. Note that as for *Core Files*, you need to specify an executable before attaching to a process.

Debug → **Debug** → **Detach** Detaches the currently debugged process from the underlying debugger. This means that the executable will continue to run independently. You can use the *Debug* → *Debug* → *Attach To Process* menu later to re-attach to this process.

Debug → **Debug** → **Debug Core File** This will open a file selection dialog that allows you to debug a core file instead of debugging a running process. Note that you must first specify an executable to debug before loading a core file.

Debug → **Debug** → **Kill** Kills the process being debugged.

8.1.3 Data

Note that most items in this menu need to access the underlying debugger when the process is stopped, not when it is running. This means that you first need to stop the process on a breakpoint or interrupt it, before using the following commands. Failing to do so will result in blank windows.

Debug → **Data** → **Data Window** Displays the Data window. If this window already exists, it is raised so that it becomes visible

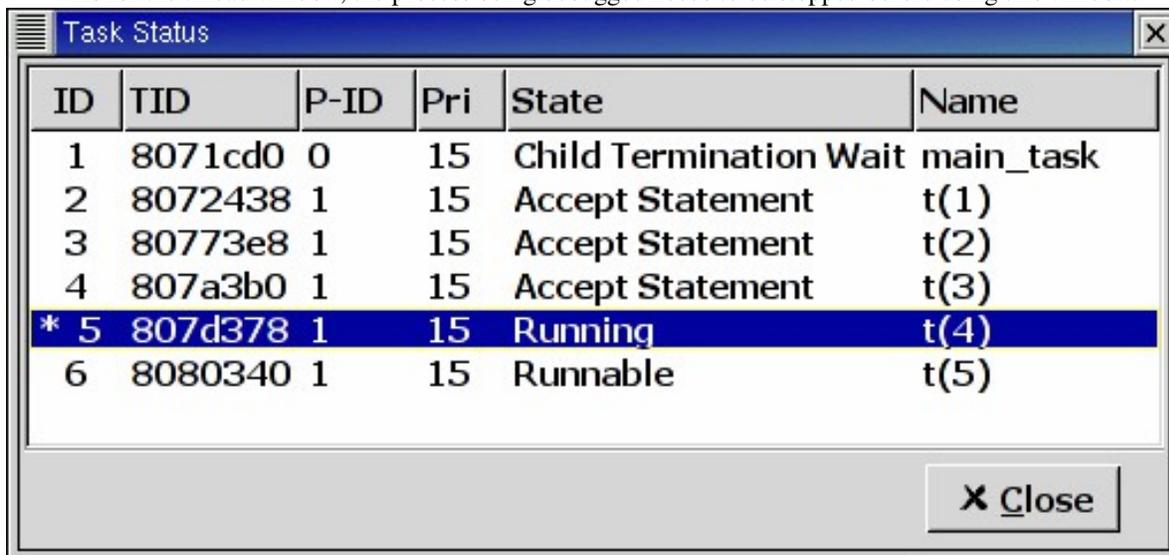
Debug → **Data** → **Call Stack** Displays the Call Stack window. See *The Call Stack Window* for more details.

Debug → **Data** → **Threads** Opens a new window containing the list of threads currently present in the executable as reported by the underlying debugger. For each thread, it will give information such as internal identifier, name and status. This information is language- and debugger-dependent. You should refer to the underlying debugger's documentation for more details. As indicated above, the process being debugged needs to be stopped before using this command, otherwise a blank list will be displayed.

When supported by the underlying debugger, clicking on a thread will change the context (variables, call stack, source file) displayed, allowing you to inspect the stack of the selected thread.

Debug → **Data** → **Tasks** For GDB only, this will open a new window containing the list of Ada tasks currently present in the executable. Similarly to the thread window, you can switch to a selected task context by clicking on it, if supported by GDB. See the GDB documentation for the list of items displayed for each task.

As for the thread window, the process being debugged needs to be stopped before using this window.



The screenshot shows a window titled "Task Status" with a table of task information. The table has columns for ID, TID, P-ID, Pri, State, and Name. The row for ID 5 is highlighted in blue.

ID	TID	P-ID	Pri	State	Name
1	8071cd0	0	15	Child Termination Wait	main_task
2	8072438	1	15	Accept Statement	t(1)
3	80773e8	1	15	Accept Statement	t(2)
4	807a3b0	1	15	Accept Statement	t(3)
* 5	807d378	1	15	Running	t(4)
6	8080340	1	15	Runnable	t(5)

Debug → **Data** → **Protection Domains** For VxWorks AE only, this will open a new window containing the list of available protection domains in the target. To change to a different protection domain, simply click on it. A @c{*} character indicates the current protection domain.

Debug → **Data** → **Assembly** Opens a new window displaying an assembly dump of the current code being executed. See *The Assembly Window* for more details.

Debug → **Data** → **Edit Breakpoints** Opens an advanced window to create and modify any kind of breakpoint, including watchpoints (see *The Breakpoint Editor*). For simple breakpoint creation, see the description of the source window.

Debug → **Data** → **Examine Memory** Opens a memory viewer/editor. See *The Memory Window* for more details.

Debug → **Data** → **Command History** Opens a dialog with the list of commands executed in the current session. You can select any number of items in this list and replay the selection automatically.

Debug → **Data** → **Display Local Variables** Opens an item in the *Data Window* containing all the local variables for the current frame.

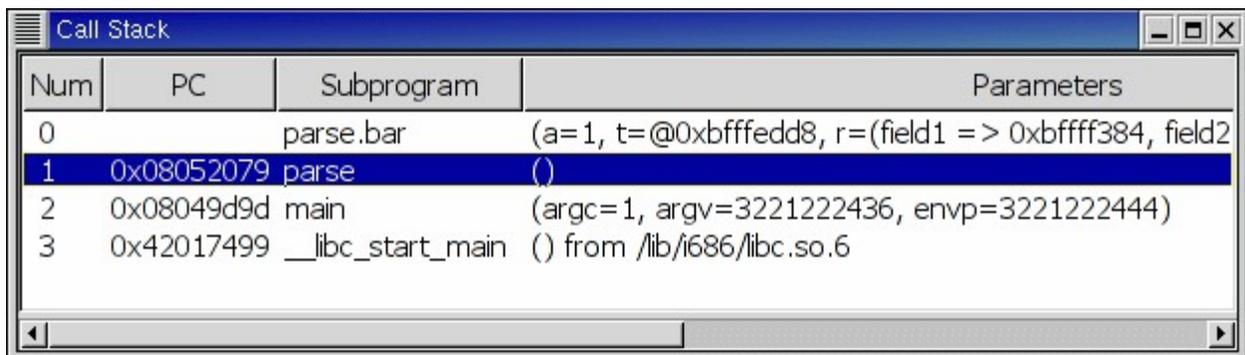
Debug → **Data** → **Display Argument** Opens an item in the *Data Window* containing the arguments for the current frame.

Debug → **Data** → **Display Registers** Opens an item in the *Data Window* containing the machine registers for the current frame.

Debug → **Data** → **Display Any Expression...** Opens a small dialog letting you specify an arbitrary expression in the *Data Window*. This expression can be a variable name, or a more complex expression, following the syntax of the underlying debugger. See the documentation of e.g gdb for more details on the syntax. The check button *Expression is a subprogram call* should be enabled if the expression is actually a debugger command (e.g *p/x var*) or a procedure call in the program being debugged (e.g *call my_proc*).

Debug → **Data** → **Recompute** Recomputes and refreshes all the items displayed in the *Data Window*.

8.2 The Call Stack Window



Num	PC	Subprogram	Parameters
0		parse.bar	(a=1, t=@0xbffffedd8, r=(field1 => 0xbffff384, field2
1	0x08052079	parse	()
2	0x08049d9d	main	(argc=1, argv=3221222436, envp=3221222444)
3	0x42017499	__libc_start_main	() from /lib/i686/libc.so.6

The call stack window gives a list of frames corresponding to the current execution stack for the current thread/task.

The bottom frame corresponds to the outermost frame where the thread is currently stopped. This frame corresponds to the first function executed by the current thread (e.g main if the main thread is in C). You can click on any frame to switch to the caller's context, this will update the display in the source window. See also the up and down buttons in the tool bar to go up and down one frame in the call stack.

The contextual menu (right mouse button) allows you to choose which information you want to display in the call stack window (via check buttons):

- *Frame number*: the debugger frame number (usually starts at 0 or 1)
- *Program Counter*: the low level address corresponding to the function's entry point.
- *Subprogram Name*: the name of the subprogram in a given frame
- *Parameters*: the parameters of the subprogram
- *File Location*: the filename and line number information.

By default, only the subprogram name is displayed. You can hide the call stack window by closing it, as for other windows, and show it again using the menu *Debug* → *Data* → *Call Stack*.

8.3 The Data Window

8.3.1 Description

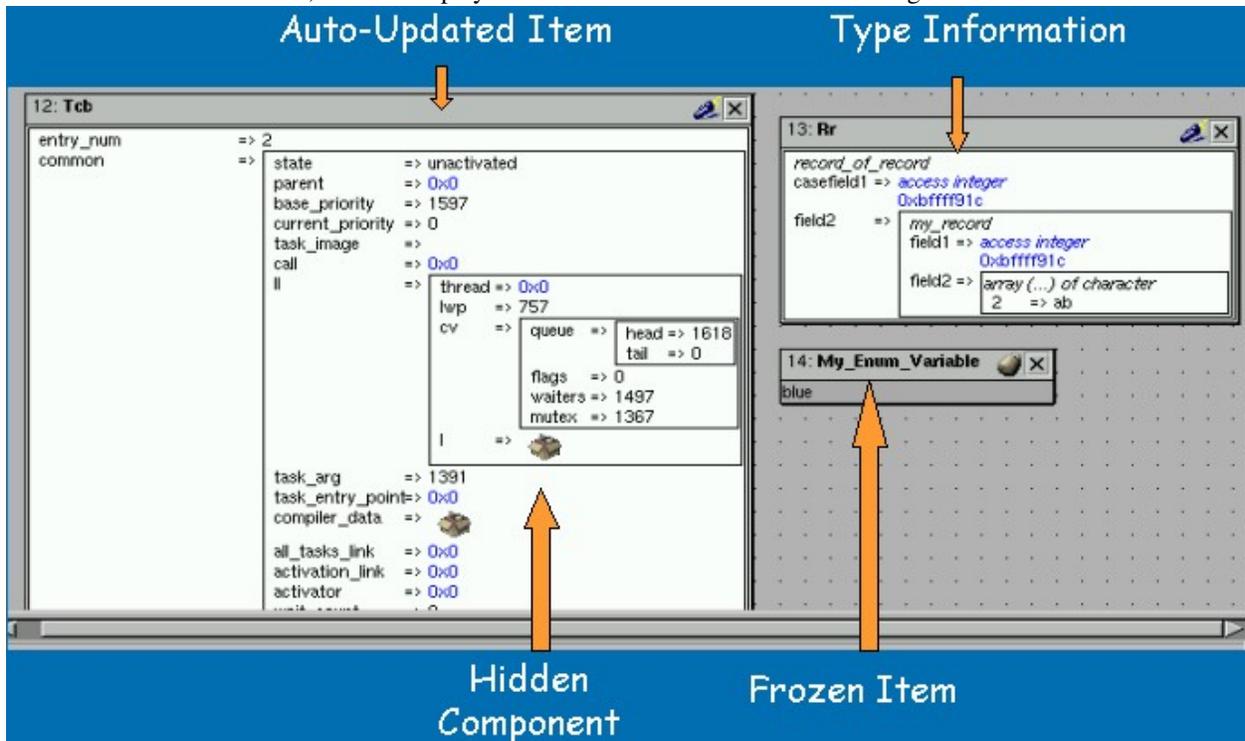
The Data Window is the area in which various information about the debugged process can be displayed. This includes the value of selected variables, the current contents of the registers, the local variables, ...

This window is open by default when you start the debugger. You can also force its display through the menu *Debug* → *Data* → *Data Window*.

The contents of the data window is preserved by default whenever you close it. Thus, if you reopen the data window either during the same debugger session, or automatically when you start a debugger on the same executable, it will display the same items again. This behavior is controlled by the *Debugger* → *Preserve State on Exit* preference.

The data window contains all the graphic boxes that can be accessed using the *Debug* → *Data* → *Display** menu items, or the data window *Display Expression...* contextual menu, or the source window *Display* contextual menu items, or finally the *graph* command in the debugger console.

For each of these commands, a box is displayed in the data window with the following information:



- A title bar containing:
 - The number of this expression: this is a positive number starting from 1 and incremented for each new box displayed. It represents the internal identifier of the box.
 - The name of the expression: this is the expression or variable specified when creating the box.
 - An icon representing either a flash light, or a lock.

This is a clickable icon that will change the state of the box from automatically updated (the flash light icon) to frozen (the lock icon). When frozen, the value is grayed, and will not change until you change the state. When updated, the value of the box will be recomputed each time an execution command is sent to the debugger (e.g step, next).

- An icon representing an ‘X’. You can click on this icon to close/delete any box.

- A main area.

The main area will display the data value hierarchically in a language-sensitive manner. The canvas knows about data structures of various languages (e.g C, Ada, C++) and will organize them accordingly. For example, each field of a record/struct/class, or each item of an array will be displayed separately. For each subcomponent, a thin box is displayed to distinguish it from the other components.

A contextual menu, that takes into account the current component selected by the mouse, gives access to the following capabilities:

Close *component* Closes the selected item.

Hide all *component* Hides all subcomponents of the selected item. To select a particular field or item in a record/array, move your mouse over the name of this component, not over the box containing the values for this item.

Show all *component* Shows all subcomponents of the selected item.

Clone *component* Clones the selected component into a new, independent item.

View memory at address of *component* Brings up the memory view dialog and explore memory at the address of the component.

Set value of *component* Sets the value of a selected component. This will open an entry box where you can enter the new value of a variable/component. Note that GDB does not perform any type or range checking on the value entered.

Update Value Refreshes the value displayed in the selected item.

Show Value Shows only the value of the item.

Show Type Shows only the type of each field for the item.

Show Value+Type Shows both the value and the type of the item.

Auto refresh Enables or disables the automatic refreshing of the item upon program execution (e.g step, next).

The *Data Window* has a local menu bar which contains a number of useful buttons:

Align On Grid Enables or disables alignment of items on the grid.

Detect Aliases Enables or disables the automatic detection of shared data structures. Each time you display an item or dereference a pointer, all the items already displayed on the canvas are considered and their addresses are compared with the address of the new item to display. If they match, (for example if you tried to dereference a pointer to an object already displayed) instead of creating a new item a link will be displayed.

Zoom in Redisplays the items in the data window with a bigger font

Zoom out Displays the items in the data window with smaller fonts and pixmaps. This can be used when you have several items in the window and you can't see all of them at the same time (for instance if you are displaying a tree and want to clearly see its structure).

Zoom Allows you to choose the zoom level directly from a menu.

Clear When this item is selected, all the boxes currently displayed are removed.

8.3.2 Manipulating items

Moving items

All the items on the canvas have some common behavior and can be fully manipulated with the mouse. They can be moved freely anywhere on the canvas, simply by clicking on them and then dragging the mouse. Note that if you are trying to move an item outside of the visible area of the data window, the latter will be scrolled so as to make the new position visible.

Automatic scrolling is also provided if you move the mouse while dragging an item near the borders of the data window. As long as the mouse remains close to the border and the button is pressed on the item, the data window is scrolled and the item is moved. This provides an easy way to move an item a long distance from its initial position.

Colors

Most of the items are displayed using several colors, each conveying a special meaning. Here is the meaning assigned to all colors (note that the exact color can be changed through the preferences dialog; these are the default colors):



black

This is the default color used to print the value of variables or expressions.

blue This color is used for C pointers (or Ada access values), i.e. all the variables and fields that are memory addresses that denote some other value in memory.

You can easily dereference these (that is to say see the value pointed to) by double-clicking on the blue text itself.

red

This color is used for variables and fields whose value has changed since the data window was last displayed. For instance, if you display an array in the data window and then select the *Next* button in the tool bar, then the elements of the array whose value has just changed will appear in red.

As another example, if you choose to display the value of local variables in the data window (*Display->Display Local Variables*), then only the variables whose value has changed are highlighted, the others are left in black.

Icons

Several different icons can be used in the display of items. They also convey special meanings.

trash bin icon

This icon indicates that the debugger could not get the value of the variable or expression. There might be several reasons, for instance the variable is currently not in scope (and thus does not exist), or it might have been optimized away by the compiler. In all cases, the display will be updated as soon as the variable becomes visible again.

package icon

This icon indicates that part of a complex structure is currently hidden. Manipulating huge items in the data window (for instance if the variable is an array of hundreds of complex elements) might not be very helpful. As a result, you can shrink part of the value to save some screen space and make it easier to visualize the interesting parts of these variables.

Double-clicking on this icon will expand the hidden part, and clicking on any sub-rectangle in the display of the variable will hide that part and replace it with that icon.

See also the description of the contextual menu to automatically show or hide all the contents of an item. Note also that one alternative to hiding subcomponents is to clone them in a separate item (see the contextual menu again).

8.4 The Breakpoint Editor

Location Variable Exception

Source location
 File: Line:
 Subprogram Name

 Address

 Regular expression

Temporary breakpoint

+ Add

Breakpoints
 Click in the 'Enb' column to change the status

Num	Enb	Type	Disp	File/Variable	Line	Exception	Subprogram
1		break	keep	a-except.adb	871	all	
3		break	keep	gps.adb	50		gps

Remove View Advanced Close

The breakpoint editor can be accessed from the menu *Debug* → *Data* → *Edit Breakpoints*. It allows manipulation of different kinds of breakpoints: at a source location, on a subprogram, at an executable address, on memory access (watchpoints), and on Ada exceptions.

You can double-click on any breakpoint in the list to open the corresponding source editor at the right location. Alternatively, you can select the breakpoint and then click on the *View* button.

The top area provides an interface to create the different kinds of breakpoints, while the bottom area lists existing breakpoints and their characteristics.

It is possible to access advanced breakpoint characteristics for a given breakpoint. First, select a breakpoint in the list. Then, click on the *Advanced* button, which will display a new dialog window. You can specify commands to run automatically after a breakpoint is hit, or specify how many times a selected breakpoint will be ignored. If running VxWorks AE, you can also change the Scope and Action settings for breakpoints.

The image shows a dialog box for configuring a breakpoint. It is divided into three main sections:

- Condition:** A text field with the label "Break only when following condition is true:" and a dropdown arrow on the right.
- Ignore:** A text field with the label "Enter the number of times to skip before stopping:" and a spinner control on the right. The value "0" is entered in the field.
- Commands:** A text field with the label "Enter commands to execute when program stops:" and a spinner control on the right.

Below the text fields are two buttons: "Record" and "Stop recording". At the bottom of the dialog are two larger buttons: "Apply" (with a checkmark icon) and "Close" (with an 'X' icon).

8.4.1 Scope/Action Settings for VxWorks AE

In VxWorks AE breakpoints have two extra properties:

- Scope: which task(s) can hit a given breakpoint. Possible Scope values are:
 - task: the breakpoint can only be hit by the task that was active when the breakpoint was set. If the breakpoint is set before the program is run, the breakpoint will affect the environment task
 - pd: .. index:: protection domain
any task in the current protection domain can hit that breakpoint
 - any:
any task in any protection domain can hit that breakpoint. This setting is only allowed for tasks in the Kernel domain.
- Action: when a task hits a breakpoints, which tasks are stopped:
 - task: stop only the task that hit the breakpoint.
 - pd: stop all tasks in the current protection domain
 - all: stop all breakable tasks in the system

These two properties can be set/changed through the advanced breakpoints characteristics by clicking on the *Advanced* button. There are two ways of setting these properties:

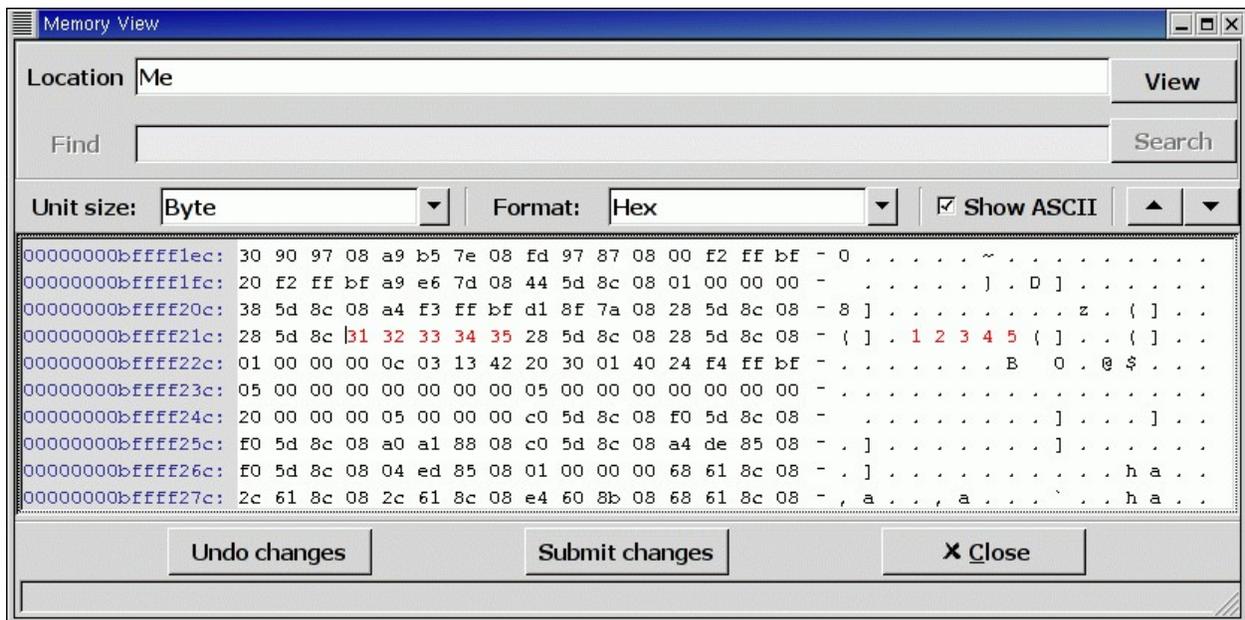
- Per breakpoint settings:
 - after setting a breakpoint (the default Scope/Action values will be task/task), select the *Scope/Action* tab in the *Advanced* settings. To change these settings on a given breakpoint, select it from the breakpoints list, select the desired values of Scope and Action and click on the *Update* button.

- Default session settings:

select the *Scope/Action* tab in the *Advanced* settings. Select the desired Scope and Action settings, check the *Set as session defaults* check box below and click the *Close* button. From now on, every new breakpoint will have the selected values for Scope and Action.

If you have enabled the preference *Debugger → Preserve state on exit*, GPS will automatically save the currently set breakpoints, and restore them the next time you debug the same executable. This allows you to immediately start debugging your application again, without resetting the breakpoints every time.

8.5 The Memory Window



The memory window allows you to display the contents of memory by specifying either an address, or a variable name.

To display memory contents, enter the address using the C hexadecimal notation: 0xabcd, or the name of a variable, e.g foo, in the *Location* text entry. In the latter case, its address is computed automatically. Then either press *Enter* or click on the *View* button. This will display the memory with the corresponding addresses in the bottom text area.

You can also specify the unit size (*Byte*, *Halfword* or *Word*), the format (*Hexadecimal*, *Decimal*, *Octal* or *ASCII*), and you can display the corresponding ASCII value at the same time.

The up and down arrows as well as the *Page up* and *Page down* keys in the memory text area allows you to walk through the memory in order of ascending/descending addresses respectively.

Finally, you can modify a memory area by simply clicking on the location you want to modify, and by entering the new values. Modified values will appear in a different color (red by default) and will only be taken into account (i.e written to the target) when you click on the *Submit changes* button. Clicking on the *Undo changes* or going up/down in the memory will undo your editing.

Clicking on *Close* will close the memory window, canceling your last pending changes, if any.

8.6 Using the Source Editor when Debugging

When debugging, the left area of each source editor provides the following information:

Lines with code

In this area, blue dots are present next to lines for which the debugger has debug information, in other words, lines that have been compiled with debug information and for which the compiler has generated some code. Currently, there is no check when you try to set a breakpoint on a non dotted line: this will simply send the breakpoint command to the underlying debugger, and usually (e.g in the case of gdb) result in setting a breakpoint at the closest location that matches the file and line that you specified.

Current line executed This is a green arrow showing the line about to be executed.

Lines with breakpoints For lines where breakpoints have been set, a red mark is displayed on top of the blue dot for the line. You can add and delete breakpoints by clicking on this area (the first click will set a breakpoint, the second click will remove it).

```

80   type Integer_Array2 is array (1 .. 2, 1 .. 3) of Integer;
81   U : Integer_Array2 := ((2, 3, 4), (5, 6, 7));
82   ((2, 3, 4), (5, 6, 7))
83   type Array_3d is array (3 .. 4, 1 .. 2, 6 .. 7) of Integer;
84   A3d : Array_3d := (others => (others => (1, 2)));
85
86   type Enum_Array is array (My_Enum'Range) of My_Enum;
87   Enum_Array_Variable : Enum_Array := (Blue => Red,
88                                       Red   => Green,
89                                       Green => Blue);
90
91   type Negative_Array is array (-50 .. -46) of Character;
92   Negative_Array_Variable : Negative_Array := ('A', 'B', 'C', 'D', 'E');
93
94   type Array_Of_Array is array (1 .. 2) of First_Index_Integer_Array;
95   Aa : Array_Of_Array := (1 => (24 => 3, 25 => 4, 26 => 5),
96                           2 => (6, 7, 8));
97

```

The second area in the source window is a text window on the right that displays the source files, with syntax highlighting. If you leave the cursor over a variable, a tooltip will appear showing the value of this variable. Automatic tooltips can be disabled in the preferences menu.

See *Preferences Dialog*.

When the debugger is active, the contextual menu of the source window contains a sub menu called *Debug* providing the following entries.

Note that these entries are dynamic: they will apply to the entity found under the cursor when the menu is displayed (depending on the current language). In addition, if a selection has been made in the source window the text of the selection will be used instead. This allows you to display more complex expressions easily (for example by adding some comments to your code with the complex expressions you want to be able to display in the debugger).

Debug → *Print *selection** Prints the selection (or by default the name under the cursor) in the debugger console.

Debug → *Display *selection** Displays the selection (or by default the name under the cursor) in the data window. The value will be automatically refreshed each time the process state changes (e.g after a step or a next command). To freeze the display in the canvas, you can either click on the corresponding icon in the data window, or use the contextual menu for the specific item (see *The Data Window* for more information).

Debug → *Print *selection*.all* Dereferences the selection (or by default the name under the cursor) and prints the value in the debugger console.

Display *selection*.all Dereferences the selection (or by default the name under the cursor) and displays the value in the data window.

View memory at address of **selection** Brings up the memory view dialog and explores memory at the address of the selection.

Set Breakpoint on Line **xx** Sets a breakpoint on the line under the cursor, in the current file.

Set Breakpoint on **selection** Sets a breakpoint at the beginning of the subprogram named *selection*

Continue Until Line **xx** Continues execution (the program must have been started previously) until it reaches the specified line.

Show Current Location Jumps to the current line of execution. This is particularly useful after navigating through your source code.

8.7 The Assembly Window

It is sometimes convenient to look at the assembly code for the subprogram or source line you are currently debugging.

You can open the assembly window by using the menu *Debug* → *Data* → *Assembly*.

```

• procedure Sdc is
•   File : File_Type;
begin
•   Msg ("Welcome to sdc. Go ahead type your commands ...");
•
•   if Argument_Count = 1 then
•       begin
•           Open (File, In_File, Argument (1));
•       exception
•           when Use_Error | Name_Error =>
•               Error_Msg ("Could not open input file, exiting.");
•           .....

```

CVS:1.16 Writable Unmodified 13:1

```

Assembly View
0x804c0b5 <_ada_sdc+249>:    call    0x804ce28 <ada__command_line>
0x804c0ba <_ada_sdc+254>:    add     $0x18,%esp
0x804c0bd <_ada_sdc+257>:    cmp     $0x1,%eax
0x804c0c0 <_ada_sdc+260>:    jne    0x804c278 <_ada_sdc+700>
0x804c0c6 <_ada_sdc+266>:    call   0x805d3b4 <system__secondary>
0x804c0cb <_ada_sdc+271>:    mov    %eax,0xffffffff0(%ebp)
0x804c0ce <_ada_sdc+274>:    call   0x805e7f4 <system__soft_link>
0x804c0d3 <_ada_sdc+279>:    mov    %eax,0xfffffd4(%ebp)
0x804c0d9 <_ada_sdc+285>:    lea   0xfffffc8(%ebp),%esi
0x804c0dc <_ada_sdc+288>:    mov    %esi,0xffffdfd0(%ebp)
0x804c0e2 <_ada_sdc+294>:    mov    %ebp,(%esi)
0x804c0e4 <_ada_sdc+296>:    movl  $0x804c0f0,0x4(%esi)
0x804c0eb <_ada_sdc+303>:    mov    %esp,0x8(%esi)
0x804c0ee <_ada_sdc+306>:    jmp   0x804c12c <_ada_sdc+368>

```

The current assembly instruction is highlighted with a green arrow on its left. The instructions corresponding to the current source line are highlighted in red by default. This allows you to easily see where the program counter will point to, once you have pressed the *Next* button on the tool bar.

Moving to the next assembly instruction is done through the *Nexti* (next instruction) button in the tool bar. If you choose “Stepi” instead (step instruction), this will also jump to the subprogram being called.

For efficiency reasons, only a small part of the assembly code around the current instruction is displayed. You can specify in the *Preferences Dialog* how many instructions are displayed by default. Also, you can easily display the instructions immediately preceding or following the currently displayed instructions by pressing one of the Page up or Page down keys, or by using the contextual menu in the assembly window.

A convenient complement when debugging at the assembly level is the ability of displaying the contents of machine registers. When the debugger supports it (as gdb does), you can select the *Debug → Data → Display Registers* menu to get an item in the canvas that will show the current contents of each machine register, and that will be updated every time one of them changes.

You might also choose to look at a single register. With gdb, select the *Debug → Data → Display Any Expression*, entering something like:

```
output /x $eax
```

in the field, and selecting the toggle button *Expression is a subprogram call*. This will create a new canvas item that will be refreshed every time the value of the register (in this case `eax`) changes.

8.8 The Debugger Console

This is the text window located at the bottom of the main window. In this console, you have direct access to the underlying debugger, and can send commands (you need to refer to the underlying debugger's documentation, but usually typing *help* will give you an overview of the commands available).

If the underlying debugger allows it, pressing Tab in this window will provide completion for the command that is being typed (or for its arguments).

There are also additional commands defined to provide a simple text interface to some graphical features.

Here is the complete list of such commands. The arguments between square brackets are optional and can be omitted.

graph (print|display) expression [dependent on display_num] [link_name name] [at x, y] [num num]

This command creates a new item in the canvas, that shows the value of *Expression*. *Expression* should be the name of a variable, or one of its fields, that is in the current scope for the debugger.

The command *graph print* will create a frozen item, that is not automatically refreshed when the debugger stops, whereas *graph display* displays an automatically refreshed item.

The new item is associated with a number, that is visible in its title bar. This number can be specified through the *num* keyword, and will be taken into account if no such item already exists. These numbers can be used to create links between the items, using the second argument to the command, *dependent on*. The link itself (i.e. the line) can be given a name that is automatically displayed, using the third argument.

graph (print|display) 'command'

This command is similar to the one above, except it should be used to display the result of a debugger command in the canvas.

For instance, if you want to display the value of a variable in hexadecimal rather than the default decimal with gdb, you should use a command like:

```
graph display `print /x my_variable`
```

This will evaluate the command between back-quotes every time the debugger stops, and display this in the canvas. The lines that have changed will be automatically highlighted (in red by default).

This command is the one used by default to display the value of registers for instance.

graph (enable|disable) display display_num [display_num ...]

This command will change the refresh status of items in the canvas. As explained above, items are associated with a number visible in their title bar.

Using the *graph enable* command will force the item to be automatically refreshed every time the debugger stops, whereas the *graph disable* command will freeze the item.

graph undisplay display_num

This command will remove an item from the canvas

8.9 Customizing the Debugger

GPS is a high-level interface to several debugger backends, in particular gdb. Each back end has its own strengths, but you can enhance the command line interface to these backends through GPS, using Python.

This section will provide a small such example. The idea is to provide the notion of “alias” in the debugger console. For example, this can be used so that you type “foo”, and this really executes a longer command, like displaying the value of a variable with a long name.

gdb already provides this feature through the *define* keywords, but we will in fact rewrite that feature in terms of python.

GPS provides an extensive Python API to interface with each of the running debugger. In particular, it provides the function “send”, which can be used to send a command to the debugger, and get its output, and the function “set_output”, which can be used when you implement your own functions.

It also provides, through *hook*, the capability to monitor the state of the debugger back-end. In particular, one such hook, *debugger_command_action_hook* is called when the user has typed a command in the debugger console, and before the command is executed. This can be used to add your own commands. The example below uses this hook.

Here is the code:

```
import GPS

aliases={}

def set_alias (name, command):
    """Set a new debugger alias. Typing this alias in a debugger window
    will then execute command"""
    global aliases
    aliases[name] = command

def execute_alias (debugger, name):
    return debugger.send (aliases[name], output=False)

def debugger_commands (hook, debugger, command):
    global aliases
    words = command.split()
    if words[0] == "alias":
        set_alias (words[1], " ".join (words [2:]))
        return True
    elif aliases.has_key (words [0]):
        debugger.set_output (execute_alias (debugger, words[0]))
        return True
    else:
        return False

GPS.Hook ("debugger_command_action_hook").add (debugger_commands)
```

The list of aliases is stored in the global variable *aliases*, which is modified by *set_alias*. Whenever the user executes an alias, the real command sent to the debugger is sent through *execute_alias*.

The real part of the work is done by *debugger_commands*. If the user is executing the *alias* command, it defines a new alias. Otherwise, if he typed the name of an alias, we really want to execute that alias. Else, we let the debugger back-end handle that command.

After you have copied this example in the `$HOME/.gps/plugin` directory, you can start a debugger as usual in GPS, and type the following in its console:

```
(gdb) alias foo print a_long_long_name
(gdb) foo
```

The first command defines the alias, the second line executes it.

This alias can also be used within the *graph display* command, so that the value of the variable is in fact displayed in the data window automatically, for instance:

```
(gdb) graph display `foo`
```

Other examples can be programmed. You could write complex python functions, which would for instance query the value of several variables, and pretty print the result. This complex python function can then be called from the debugger console, or automatically every time the debugger stops through the *graph display* command.

VERSION CONTROL SYSTEM

GPS offers the possibility for multiple developers to work on the same project, through the integration of version control systems (VCS). Each project can be associated to a VCS, through the *VCS* tab in the Project properties editor. *The Project Properties Editor*.

GPS does not come with any version control system: it uses underlying command-line systems such as Subversion or ClearCase to perform the low level operations, and provides a high level user interface on top of them. Be sure to have a properly installed version control system before enabling it under GPS.

The systems that are supported out of the box in GPS are:

Auto GPS can be setup to auto-detect the actual VCS to use for each project. This is done by selecting *Auto* in the *VCS* tab of the Project properties editor. *The Project Properties Editor*. This is also the default behavior when no VCS is specified in the project.

ClearCase The standard ClearCase interface, which is built-in and uses a generic GPS terminology for VCS operations.

Note that, at the moment, only Snapshot Views are supported in the ClearCase integration; Dynamic Views are not supported.

ClearCase Native Which is fully customizable and uses by default the terminology specific to ClearCase.

Note that, at the moment, only Snapshot Views are supported in the ClearCase integration; Dynamic Views are not supported.

CVS The Concurrent Version System.

GPS needs a corresponding *patch* command that usually comes with it.

Git Distributed fast source code management. Support for Git on GPS is partial. Basic commands are supported but the full power of Git (like working with the index) is only available on the command line.

GPS needs a corresponding *diff* command that usually comes with it.

Mercurial An experimental plugin for supporting Mercurial.

Subversion The Subversion version control system. Note that on Windows this version is intended to be used with Cygwin/Subversion and fully supports the Cygwin path names.

GPS needs a corresponding *patch* and *diff* command that usually comes with it.

Subversion Windows The Windows native Subversion version control system. The external Subversion commands are expected to be built for the Win32 subsystem. This version does not support Cygwin path names.

GPS needs a corresponding *patch* and *diff* command that usually comes with it.

The default VCS that GPS will use is “Auto” by default, and this can be configured through *The Preferences Dialog*.

It is also possible to add your own support for other version control systems, or modify one of the existing interfaces, see *Adding support for new Version Control Systems* for more information.

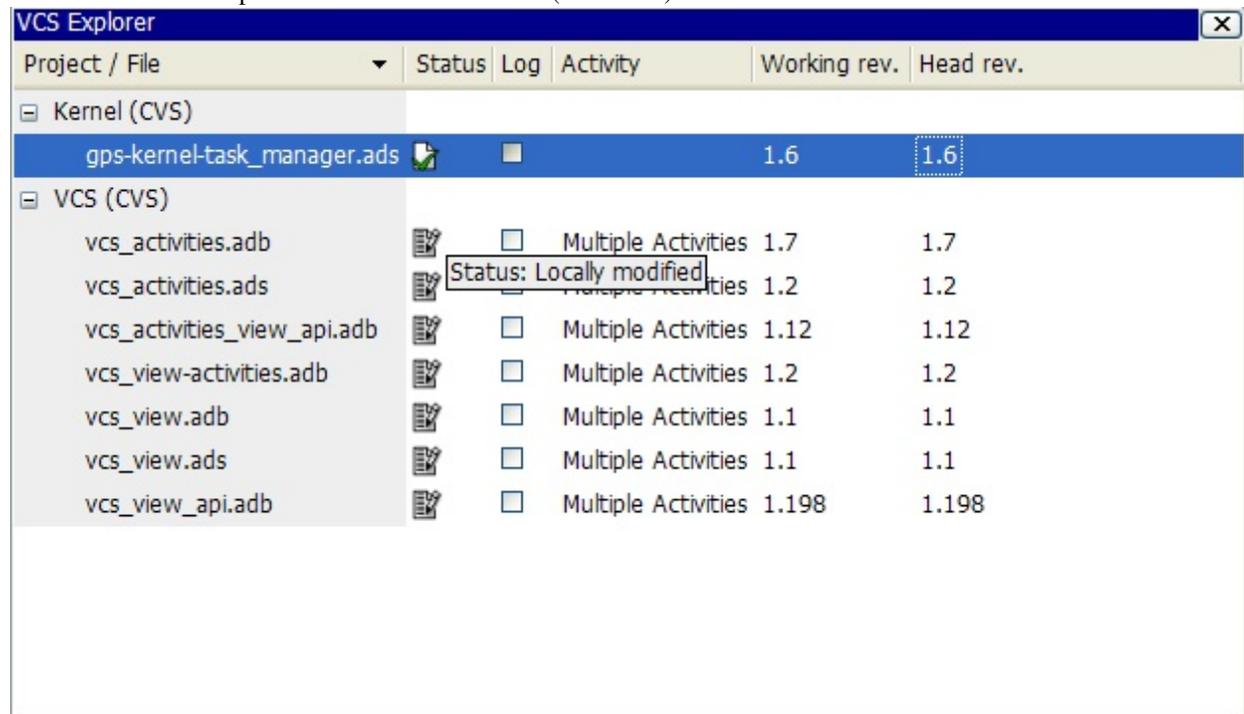
It is recommended that you first get familiar with the version control system that you intend to use in GPS first, since many concepts used in GPS assume basic knowledge of the underlying system.

Associating a VCS to a project enables the use of basic VCS features on the source files contained in the project. Those basic features typically include the checking in and out of files, the querying of file status, file revision history, comparison between various revisions, and so on.

Note: the set-up must make sure that the VCS commands can be launched without entering a password.

9.1 The VCS Explorer

The VCS Explorer provides an overview of source files and their status. A file edited in GPS will be automatically added on the VCS Explorer with a Modified status (see below).



The easiest way to bring up the VCS Explorer is through the menu *VCS->Explorer*. The Explorer can also be brought up using the contextual menu *Version Control->Query status* on files, directories and projects in the file and project views, and on file editors. *The Version Control Contextual Menu*.

The VCS Explorer contains the following columns:

Project / File This is a two levels tree, the first level contains the name of the project and the second the name of files inside the project. Next to the project name the VCS name, if any, is displayed. This is the only information available for a project. The columns described below are for the files only. This column can be sorted by clicking on the header.

Status Shows the status of the file. This column can be sorted by clicking on the header. The different possible status for files are the following:

Unknown 

The status is not yet determined or the VCS repository is not able to give this information (for example if it is unavailable, or locked).

Not registered 

The file is not known to the VCS repository.

Up-to-date 

The file corresponds to the latest version in the corresponding branch on the repository.

Added 

The file has been added remotely but is not yet updated in the local view.

Removed 

The file still exists locally but is known to have been removed from the VCS repository.

Modified 

The file has been modified by the user or has been explicitly opened for editing.

Needs merge 

The file has been modified locally and on the repository.

Needs update 

The file has been modified in the repository but not locally.

Contains merge conflicts 

The file contains conflicts from a previous update operation.

Log This column indicates whether a revision log exists for this file.

Activity The name of the activity the file belongs to. See *The VCS Activities* for more details.

Working rev. Indicates the version of the local file.

Head rev. Indicates the most recent version of the file in the repository.

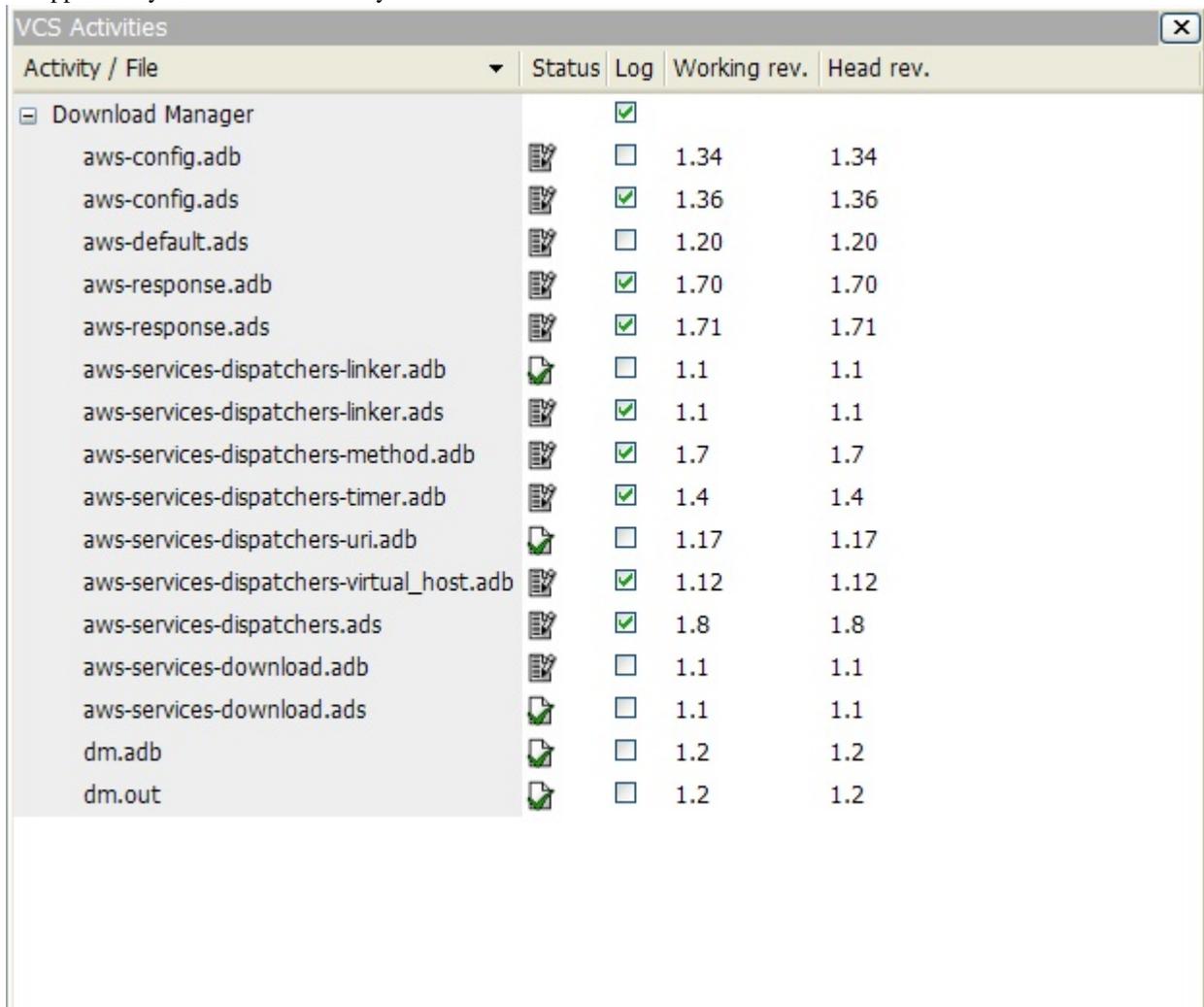
The VCS Explorer supports multiple selections. To select a single line, simply left-click on it. To select a range of lines, select the first line in the range, then hold down the `Shift` key and select the last line in the range. To add or remove single columns from the selection, hold down the `Control` key and left-click on the columns that you want to select/unselect. It is also possible to select files having the same status using the *Select files same status* menu entry. See *The Version Control Contextual Menu*.

The explorer also provides an *interactive search* capability allowing you to quickly look for a given file name. The default key to start an interactive search is `Ctrl-i`.

The VCS contextual menu can be brought up from the VCS explorer by left-clicking on a selection or on a single line. See *The Version Control Contextual Menu*.

9.2 The VCS Activities

The VCS Activities give the ability to group files to be committed together. The set of files can be committed atomically if supported by the version control system used.



Activity / File	Status	Log	Working rev.	Head rev.
Download Manager		<input checked="" type="checkbox"/>		
aws-config.adb		<input type="checkbox"/>	1.34	1.34
aws-config.ads		<input checked="" type="checkbox"/>	1.36	1.36
aws-default.ads		<input type="checkbox"/>	1.20	1.20
aws-response.adb		<input checked="" type="checkbox"/>	1.70	1.70
aws-response.ads		<input checked="" type="checkbox"/>	1.71	1.71
aws-services-dispatchers-linker.adb		<input type="checkbox"/>	1.1	1.1
aws-services-dispatchers-linker.ads		<input checked="" type="checkbox"/>	1.1	1.1
aws-services-dispatchers-method.adb		<input checked="" type="checkbox"/>	1.7	1.7
aws-services-dispatchers-timer.adb		<input checked="" type="checkbox"/>	1.4	1.4
aws-services-dispatchers-uri.adb		<input type="checkbox"/>	1.17	1.17
aws-services-dispatchers-virtual_host.adb		<input checked="" type="checkbox"/>	1.12	1.12
aws-services-dispatchers.ads		<input checked="" type="checkbox"/>	1.8	1.8
aws-services-download.adb		<input type="checkbox"/>	1.1	1.1
aws-services-download.ads		<input type="checkbox"/>	1.1	1.1
dm.adb		<input type="checkbox"/>	1.2	1.2
dm.out		<input type="checkbox"/>	1.2	1.2

The way to bring up the VCS Activities view is through the *VCS->Activities* menu.

The VCS Activities view contains the following columns:

Activity / File The name of the activity or files belonging to an activity. This column can be sorted by clicking on the header.

Status Shows the status of the file. This column can be sorted by clicking on the header. See *The VCS Explorer* for a full description.

Log This column indicates whether a revision log exists for this file.

Working rev. Indicates the version of the local file.

Head rev. Indicates the most recent version of the file in the repository.

The VCS Explorer supports multiple selections. To select a single line, simply left-click on it. To select a range of lines, select the first line in the range, then hold down the `Shift` key and select the last line in the range. To add or

remove single columns from the selection, hold down the `Control` key and left-click on the columns that you want to select/unselect.

There are different contextual menu entries depending on the position on the screen. On an empty area we have a simple contextual menu:

Create new activity Create a new activity. The name can be edited by double clicking on it.

On an activity line the contextual menu is:

Group commit This is a selectable menu entry. It is activated only if the VCS supports atomic commit and absolute filenames. See *The VCS node* for full details.

Create new activity Create a new activity. The name can be edited by double clicking on it.

Re-open activity / Close activity If the activity is closed it is possible to re-open it and if it is opened it is possible to close it manually.

Delete activity Remove the activity.

Commit activity Commit the activity. If group commit is activated then the commit log content is generated using a template file fully configurable. See *Files*. If group commit is not activated then the log content for each activity file is the file log catenated with the activity log. After this operation the file's log are removed but the activity log is kept as documentation.

Query status Query the status for all the source files contained in the activity.

Update Update all the source files contained in the activity.

Compare against head revision Show a visual comparison between the local activity files and the most recent version of those files in the repository.

Build patch file Create a patch file (in text format) for the activity. The patch file contains a header (the activity log and file's logs) and the diff of each file. The header format is fully configurable using a template file. See *Files*.

Edit revision log Edit the current revision log for activity. This log is shared with all the activity files.

Remove revision log Remove the current revision log for activity. This menu is present only if the activity revision log exists.

On a file line the contextual menu contains:

Create new activity Create a new activity. The name can be edited by double clicking on it.

Remove from activity Remove the selected file from the activity and delete the activity log.

Edit revision log Edit the current revision log for the selected file.

9.3 The VCS Menu

Basic VCS operations can be accessed through the VCS menu. Most of these functions act on the current selection, i.e. on the selected items in the VCS Explorer if it is present, or on the currently selected file editor, or on the currently selected item in the *Tools->Views->Files*. In most cases, the VCS contextual menu offers more control on VCS operations. *The Version Control Contextual Menu*.

Explorer Open or raise the VCS Explorer. *The VCS Explorer*.

Update all projects Update the source files in the current project, and all imported sub-projects, recursively.

Query status for all projects Query the status of all files in the project and all imported sub-projects.

Create tag... Create a tag or branch tag starting from a specific root directory. The name of the tag is a simple name.

Switch tag... Switch the local copy to a specific tag. The name of the tag depends on the external VCS used. For CVS this is the simple tag name, for Subversion the tag must conform to the default repository layout. For a branch tag this is `/branches/<tag_name>/<root_dir>`.

For a description of the other entries in the VCS menu, see *The Version Control Contextual Menu*

9.4 The Version Control Contextual Menu

This section describes the version control contextual menu displayed when you right-click on an entity (e.g. a file, a directory, a project) from various parts of GPS, including the project view, the source editor and the VCS Explorer.

Depending on the context, some of the items described in this section won't be shown, which means that they are not relevant to the current context.

Remove project Only displayed on a project line. This will remove the selected project from the VCS Explorer.

Expand all Expand all VCS Explorer project nodes.

Collapse all Collapse all VCS Explorer project nodes.

Clear View Clear the VCS Explorer.

Query status Query the status of the selected item. Brings up the VCS Explorer.

Update Update the currently selected item (file, directory or project).

Commit Submits the changes made to the file to the repository, and queries the status for the file once the change is made.

It is possible to tell GPS to check the file before the actual commit happens. This is done by specifying a *File checker* in the VCS tab of the project properties dialog. This *File checker* is in fact a script or executable that takes an absolute file name as argument, and displays any error message on the standard output. The VCS commit operation will actually occur only if nothing was written on the standard output.

It is also possible to check the change-log of a file before commit, by specifying a *Log checker* in the project properties dialog. This works on change-log files in the same way as the *File checker* works on source files.

Open Open the currently selected file for writing. On some VCS systems, this is a necessary operation, and on other systems it is not.

View entire revision history Show the revision logs for all previous revisions of this file.

View specific revision history Show the revision logs for one previous revision of this file.

Compare against head revision Show a visual comparison between the local file and the most recent version of that file in the repository.

Compare against other revision Show a visual comparison between the local file and one specific version of that file in the repository.

Compare two revisions Show a visual comparison between two specific revisions of the file in the repository.

Compare base against head Show a visual comparison between the corresponding version of the file in the repository and the most recent version of that file.

Compare against tag/branch Only available on a Revision View and over a tag/branch. Show a visual comparison between the corresponding version of the file in the repository and the version of that file in the tag/branch.

Annotate Display the annotations for the file, i.e. the information for each line of the file showing the revision corresponding to that file, and additional information depending on the VCS system.

When using CVS or Subversion, the annotations are clickable. Left-clicking on an annotation line will query and display the changelog associated to the specific revision for this line.

Remove Annotate Remove the annotations from the selected file.

Edit revision log Edit the current revision log for the selected file.

Edit global ChangeLog Edit the global ChangeLog entry for the selected file. *Working with global ChangeLog file.*

Remove revision log Clear the current revision associated to the selected file.

Add Add a file to the repository, using the current revision log for this file. If no revision log exists, activating this menu will create one. The file is committed in the repository.

Add/No commit Add a file to the repository, using the current revision log for this file. If no revision log exists, activating this menu will create one. The file is not committed in the repository.

Remove Remove a file from the repository, using the current revision log for this file. If no revision log exists, activating this menu will create one. The modification is committed in the repository.

Remove/No commit Remove a file from the repository, using the current revision log for this file. If no revision log exists, activating this menu will create one. The modification is not committed in the repository.

Revert Revert a locale file to the repository revision, discarding all local changes.

Resolved Mark files' merge conflicts as resolved. Some version control systems (like Subversion) will block any commit until this action is called.

Switch tag/branch Only available on a Revision View and over a tag/branch name. Will switch the tree starting from a selected root to this specific tag or branch.

Merge Only available on a Revision View and over a tag/branch name. Merge file changes made on this specific tag/branch.

View revision Only available on a Revision View and over a revision.

Commit as new Activity An action to prepare a group-commit in just one-click. This action will:

create an anonymous activity,

add all files selected into the VCS Explorer into the newly created anonymous activity,

open the activity log, Just fill the activity log and commit the anonymous activity.

Add to Activity A menu containing all the current activities. Selecting one will add the current file to this activity. This menu is present only if the file is not already part of an activity.

Remove from Activity Remove file from the given activity. This menu is present only if the file is already part of an activity.

Directory Only available when the current context contains directory information

Add/No commit Add the selected directory into the VCS.

Remove/No commit Remove the selected directory from the VCS.

Commit Commit the selected directory into the VCS. This action is available only if the VCS supports commit on directories, *The VCS node.*

Add to Activity Add the selected directory into the VCS. This action is available only if the VCS supports commit on directories, *The VCS node.*

Query status for directory Query status for the files contained in the selected directory.

Update directory Update the files in the selected directory.

Query status for directory recursively Query status for the files in the selected directory and all subdirectories recursively. Links and hidden directories are not included.

Update directory recursively Update the files in the selected directory and all subdirectories recursively. Links and hidden directories not included..

Project Only available when the current context contains project information

List all files in project Bring up the VCS Explorer with all the source files contained in the project.

Query status for project Query the status for all the source files contained in the project.

Update project Update all the source files in the project.

List all files in project and sub-projects Bring up the VCS Explorer with all the source files contained in the project and all imported sub-projects.

Query status for project and sub-projects Query the status for all the source files contained in the project and all imported sub-projects.

Update project and sub-projects Update all the source files in the project and all imported sub-projects.

Select files same status Select the files having the same status as the current selected file.

Filters Only available from the VCS Explorer. This menu controls filtering of the items displayed in the list.

Show all status Do not filter out any file from the list in the VCS Explorer.

Hide all status Filter out all the files from the list in the VCS Explorer.

Show <status> When disabled, filter out the files with the given status from the VCS Explorer.

9.5 Working with global ChangeLog file

A global ChangeLog file contains revision logs for all files in a directory and is named ChangeLog. The format for such a file is:

```
**ISO-DATE *name <e-mail>***  
  
<HT>* **filename**[, **filename**]:  
<HT>revision history
```

where:

ISO-DATE A date with the ISO format YYYY-MM-DD

name A name, generally the developer name

<e-mail> The e-mail address of the developer surrounded with '<' and '>' characters.

HT Horizontal tabulation (or 8 spaces)

The *name* and *<e-mail>* items can be entered automatically by setting the *GPS_CHANGELOG_USER* environment variable. Note that there is two spaces between the *name* and the *<e-mail>*:

On sh shell:

```
export GPS_CHANGELOG_USER="John Doe <john.doe@home.com>"
```

On Windows shell:

```
set GPS_CHANGELOG_USER="John Doe <john.doe@home.com>"
```

Using the menu entry **Edit global ChangeLog** will open the file ChangeLog in the directory where the current selected file is and create the corresponding ChangeLog entry. This means that the ISO date and filename headers will be created if not yet present. You will have to enter your name and e-mail address.

This ChangeLog file serve as a repository for revision logs, when ready to check-in a file use the standard **Edit revision log** menu command. This will open the standard revision log buffer with the content filled from the global ChangeLog file.

9.6 The Revision View

The revision view is used to display a revision tree for a given file. Each node contains information for a specific revision of the file.

Revision	Author	Date / Log
1.203	charlet	2006/02/07 09:58:29
1.202	obry	2006/02/07 09:52:48
1.200	obry	2006/01/04 13:28:58 (remote-branchpoint remote-branch)
		(On_Menu_Edit_Log): Use Open_File_Editor Group parameter to properly open the files in the right MDI. We do not split the MDI here. Minor reformatting. -- Take advantage of the Open_File_Editor grouping support when opening a log files. This avoid unwanted splitting of the MDI. This behavior is now in line with the Commit (via revision log) action. Fix for F104-007.
	remote-branchpoint	
	remote-branch	
1.197	obry	2005/12/03 20:06:02
1.195	setton	2005/11/23 17:09:12
1.194	briot	2005/11/21 15:03:30
1.193	lambourg	2005/11/09 16:33:10 (gps-3_1-branchpoint gps-3_1-branch)

the revision number This corresponds to the external VCS revision number.

author The author of this revision.

date / log For root nodes this column contains the check-in date and eventually the list of tags and branches associated with this revision. For children nodes this contains the log for the corresponding revision.

TOOLS

10.1 The Tools Menu

The *Tools* menu gives access to additional tools. Some items are currently disabled, meaning that these are planned tools not yet available.

The list of active items includes:

Views

Bookmarks *The Bookmarks view.*

Call Trees Open a tree view of function callers and callees. See also

Callgraph browser.

Clipboard *The Clipboard view.*

Coverage Report *Coverage Report.*

Files Open a file system explorer on the left area.

The Files View.

File Switches *File Switches.*

Outline Open a view of the current source editor.

The Outline view.

Messages Open the Messages window

The Messages window.

Project *The Project view.*

Remote *Setup a remote project.*

Scenario *Scenarios and Configuration Variables.*

Tasks *The Task Manager.*

VCS Activities *The VCS Activities.*

VCS Explorer *The VCS Explorer.*

Windows Open a view containing all currently opened files.

The Windows view.

Browsers

Call Graph *Callgraph browser.*

Dependency *The Dependency Browser.*

Elaboration Cycles *The Elaboration Circularities browser.*

Entity *The Entity Browser.*

Coding Standard *Coding Standard.*

Compare *Visual Comparison.*

Consoles

GPS Shell Open a shell console at the bottom area of GPS. Note that this not an OS shell console, but a GPS shell console, where you can type GPS specific commands such as *help*.

The Shell and Python Windows.

Python Open a python console to access the python interpreter. *The Shell and Python Windows.*

OS Shell Open an OS (Windows or Unix) console, using the environment variables *SHELL* and *COMSPEC* to determine which shell to use. *The Shell and Python Windows.*

On Unix, this terminal behaves a lot like a standard Unix terminal. In particular, you need to make sure that your shell will output all the information. In some cases, the configuration of your shell (*.bashrc* if you are running *bash* for instance) will deactivate the *echo* of what you type to the terminal. Since GPS is not outputting anything on its own, just showing what the shell is outputting, you need to somehow ensure that your shell always echos what you type. This is done by running the command:

```
stty echo
```

in such cases. In general, this can be safely done in your *.bashrc*

Auxiliary Builds Open the console containing auxiliary builds output. For now, only cross-reference automated generation output is redirected to this console. *Working with two compilers.*

Coverage *Code Coverage.*

Documentation *Documentation Generation.*

GNATtest *Working With Unit Tests.*

Stack Analysis *Stack Analysis.*

Macro *Recording and replaying macros.*

Metrics *Metrics.*

Plug-ins *The Plug-ins Editor.*

Interrupt Interrupt the last task launched (e.g. compilation, vcs query, ...).

10.2 Coding Standard

The Coding Standard menu allows you to edit your coding standard file, as can be understood by *gnatcheck*, as well as run it against your code, to verify its compliance with this coding standard.

Note that you can also use the contextual menu to check the conformance of a particular project or source file against a Coding Standard.

The Coding standard editor is triggered by the menu Tools->Coding Standard->Edit Rules File. The editor allows you to select an existing coding standard file, or create a new one. The editor adapts itself to the version of gnatcheck you are using on your local machine.

The currently used rules are summarized in the bottom of the editor. Once all rules are defined, you can check the box 'Open rules file after exit' to manually verify the created file.

Once the Coding Standard file is created, you can define it as the default coding standard file for a project by going to the project editor, selecting the 'Switches' tab, and using this file in the 'Gnatcheck' section.

10.3 Visual Comparison

The visual comparison, available either from the VCS menus or from the Tools menu, provide a way to display graphically differences between two or three files, or two different versions of the same file.

The 2-file comparison tool is based on the standard text command *diff*, available on all Unix systems. Under Windows, a default implementation is provided with GPS, called *gnudiff.exe*. You may want to provide an alternate implementation by e.g. installing a set of Unix tools such as cygwin (<http://www.cygwin.com>).

The 3-file comparison tool is based on the text command *diff3*, available on all Unix systems. Under Windows, this tool is not shipped with GPS. It is available as part of cygwin, for example.

When querying a visual comparison in GPS, in Side_By_Side mode, the user area will show, side by side, editors for the files involved in the comparison. The reference file is placed by default on the left side. When in Unified mode, GPS will not open a new editor, but will show all the changes directly in the original editor. Note that Unified mode is relevant only when comparing two files: when comparing three files, the Side_By_Side mode is used.

Color highlighting will be added to the file editors:

gray This color is used for all the chunks on the reference (left) file. Only the modified (right) file is displayed with different colors.

yellow This color is used to display lines that have been modified compared to the reference file. When there are fine differences within one line, they are shown in a brighter yellow.

green Used to display lines added compared to the reference file; in other words, lines that are not present in the reference file.

red Used to display lines removed from the reference file; in other words, lines that are present only in the reference file.

These colors can be configured, *The Preferences Dialog*.

As with all highlighted lines in GPS, the visual differences highlights are visible in the Speed Column at the left of the editors.

Blank lines are also added in the editors, in places that correspond to existing lines in the other editors. The vertical and horizontal scrolling are synchronized between all editors involved in a visual comparison.

When a visual comparison is created, the Locations View is populated with the entries for each chunk of differences, and can be used to navigate between those.

Closing one of the editors involved in a visual comparison removes the highlighting, blank lines, and scrolling in the other editors.

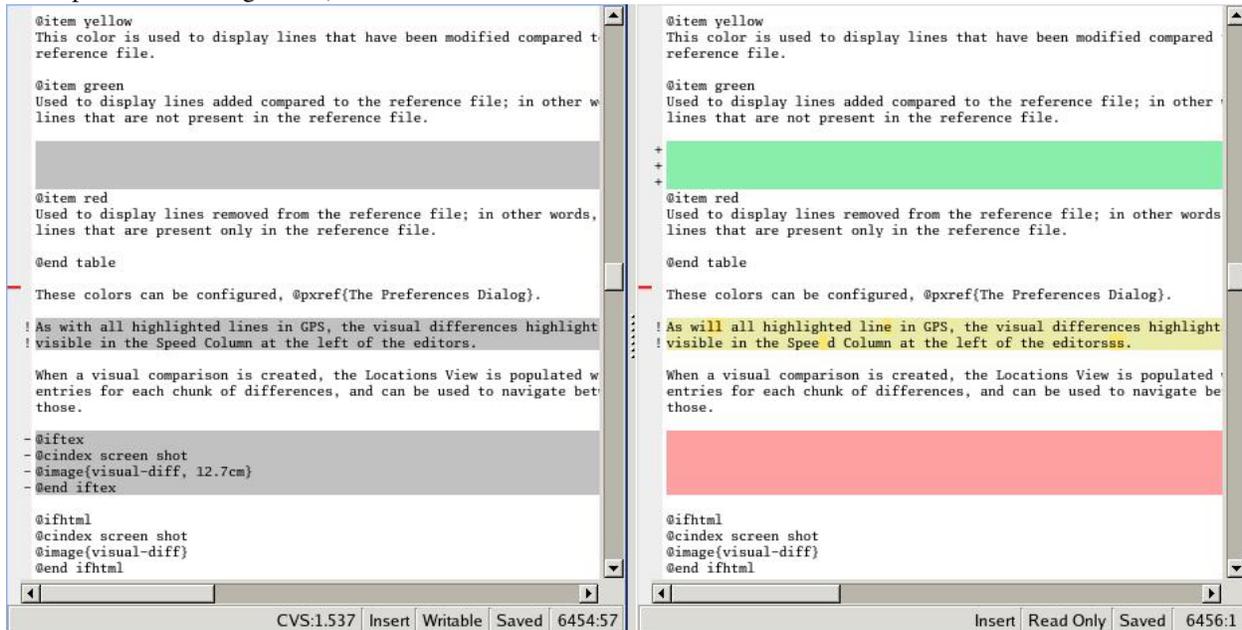
Editors involved in a visual comparison have a contextual menu *Visual diff* that contains the following entries:

Recompute Regenerates the visual comparison. This is useful, for example, when one of the editors has been modified by hand while it was involved in a visual comparison

Hide Removes the highlighting corresponding to the visual comparison from all editors involved

Close editors Closes all editors involved in this visual comparison

Use this editor as reference Change the reference to this editor. (This is only visible when displaying a visual comparison involving 3 files).



10.4 Code Fixing

GPS provides an interactive way to fix or improve your source code, based on messages (errors and warnings) generated by the GNAT compiler.

This capability is integrated with the *Locations View* (see *The Locations view*): when GPS can take advantage of a compiler message, an icon is added on the left side of the line.

For a simple fix, a wrench icon is displayed. If you click with the left button on this icon, the code will be fixed automatically, and you will see the change in the corresponding source editor. An example of a simple fix, is the addition of a missing semicolon.

You can also check what action will be performed by clicking on the right button which will display a contextual menu with a text explaining the action that will be performed. Similarly, if you display the contextual menu anywhere else on the message line, a sub menu called *Auto Fix* gives you access to the same information. In the previous example of a missing semicolon, the menu will contain an entry labelled *Add expected string ";"*. Two nested menu items let you choose to *Apply to this occurrence* or *Apply to all similar errors*. Latter choice will apply the same simple fix for all errors which are detected by the system as being the same kind. This is based on message parsing.

Once the code change has been performed, the tool icon is no longer displayed.

For more complex fixes, where more than one change is possible, a wrench icon with a blue *plus* sign is displayed. In this case, clicking on the icon will display the contextual menu directly, giving you access to the possible choices. For example, this will be the case when an ambiguity is reported by the compiler for resolving an entity.

Right clicking on a message with a fix will open a contextual menu with an entry "Auto Fix". Fixes that can be applied by clicking on the wrench are available through that menu as well. In addition, if one of the fixes is considered to be safe by GPS, additional entries will be provided to apply fixes on multiple messages:

Fix all simple style errors and warnings This entry is offered only when the selected message is a warning and a style error. Will fix all other warnings and style errors for which a unique simple fix is available.

Fix all simple errors Will fix all errors messages for which a unique simple fix is available

10.5 Documentation Generation

GPS provides a documentation generator which processes source files and generates annotated HTML files.

It is based on the source cross-reference information (e.g. generated by GNAT for Ada files). This means that you should ensure that cross-reference information has been generated before generating the documentation. It also relies on standard comments that it extracts from the source code. Note that unlike other similar tools, no macro needs to be put in your comments. The engine in charge of extracting them coupled with the cross-reference engine gives GPS all the flexibility needed to generate accurate documentation.

The documentation is put by default into a directory called `doc`, created under the object directory of the root project loaded in GPS. If no such object directory exists, then it is directly created in the same directory as the root project. This behavior can be modified by specifying the attribute `Documentation_Dir` in the package `IDE` of your root project:

```
project P is
  package IDE is
    for Documentation_Dir use "html";
  end IDE;
end P;
```

Once the documentation is generated, the main documentation file is loaded in your default browser.

The documentation generator uses a set of templates files to control the final rendering. This means that you can control precisely the rendering of the generated documentation. The templates used for generating the documentation can be found under `<install_dir>/share/gps/docgen2`. If you need a different layout as the proposed one, you can change directly those files.

In addition, user-defined structured comments can be used to improve the generated documentation. The structured comments use xml-like tags. To define your own set of tags, please refer to the GPS python extension documentation (from GPS Help menu, choose ‘Python extensions’).

The string values inside those tags are handled very roughly the same way as in regular xml: duplicated spaces and line returns are ignored. One exception is that the layout is preserved in the following cases:

The line starts with “- ” or ““* ” In this case, GPS makes sure that a proper line return precedes the line. This is to allow lists in comments

The line starts with a capital letter GPS then supposes that the preceding line return was intended, so it is kept

Some default tags have been already defined by GPS in `<install_dir>/share/gps/plugin-ins/docgen_base_tags.py`. The tags handled are:

summary Short summary of what a package or method is doing.

description Full description of what a package or method is doing.

parameter (attribute “name” is expected) Description of the parameter named “name”.

exception Description of possible exceptions raised by the method.

seealso Reference to another package, method, type, etc.

c_version For bindings, the version of the C file.

group For packages, this builds an index of all packages in the project grouped by categories.

code When the layout of the value of the node needs to be preserved. The text is displayed using a fixed font (monospace).

The following sample shows how those tags are translated:

```
-- <description>
--   This is the main description for this package. It can contain a complete
--   description with some xml characters as < or >.
-- </description>
-- <group>Group1</group>
-- <c_version>1.0.0</c_version>
package Pkg is

  procedure Test (Param : Integer);
  -- <summary>Test procedure with a single parameter</summary>
  -- <parameter name="Param">An Integer</parameter>
  -- <exception>No exception</exception>
  -- <seealso>Test2</seealso>

  procedure Test2 (Param1 : Integer; Param2 : Natural);
  -- <summary>Test procedure with two parameters</summary>
  -- <parameter name="Param1">An Integer</parameter>
  -- <parameter name="Param2">A Natural</parameter>
  -- <exception>System.Assertions.Assert_Failure if Param1 < 0</exception>
  -- <seealso>Test</seealso>

end Pkg;
```

Its documentation will be:

The screenshot shows the documentation for a package named **PACKAGE: PKG (SPEC / BODY)**. It includes a description, a binding from C File version 1.0.0, and two subprograms: **Test** and **Test2**. The **Test** subprogram has a summary, a parameter **Param**, and no exceptions. The **Test2** subprogram has a summary, parameters **Param1** and **Param2**, and an exception **System.Assertions.Assert_Failure if Param1 < 0**. The interface also features a left sidebar with navigation options and a right sidebar with an index.

The documentation generator can be invoked from the *Tools->Documentation* menu:

Generate project Generate documentation for all files from the loaded project.

Generate projects & subprojects Generate documentation for all files from the loaded project as well all its subprojects.

Generate current file Generate documentation for the file you are currently editing.

Generate for... This will open a File Selector Dialog (*The File Selector*) and documentation will be generated for the file you select.

In addition, when relevant (depending on the context), right-clicking with your mouse will show a *Documentation* contextual menu.

From a source file contextual menu, you have one option called *Generate for <filename>*, that will generate documentation for this file and if needed its corresponding body (*The Preferences Dialog*).

From a project contextual menu (*The Project view*), you will have the choice between generating documentation for all files from the selected project only or from the selected project recursively.

You will find the list of all documentation options in *The Preferences Dialog*.

Note that the documentation generator relies on the ALI files created by GNAT. Depending on the version of GNAT used, the following restrictions may or may not apply:

- A type named *type* may be generated in the type index.
- Parameters and objects of private generic types may be considered as types.

10.6 Working With Unit Tests

GPS relies on *gnattest* tool that creates unit-test stubs as well as a test driver infrastructure (harness). Harness can be generated for project hierarchy, single project or a package. Generation process can be launched from *Tools->GNATtest* menu or from contextual menu.

After generation of harness project GPS will switch to it, allowing you to implement tests, compile and run the harness. At any moment you can exit harness project and return to original project.

10.6.1 The GNATtest Menu

The *GNATtest* submenu is available from the *Tools* global menu and contains:

Generate unit test setup Generate harness for the root project.

Generate unit test setup recursive Generate harness for the root project and subprojects.

Show not implemented tests Find never modified tests and show them in Locations view. This menu is active in harness project only.

Exit from harness project Return from harness to original project.

10.6.2 The Contextual Menu

When relevant (depending on the context), right-clicking with your mouse will show GNATtest-related contextual menu entries.

Pointing to a source file containing the library package declaration, you have an option called *GNATtest/Generate unit test setup for <file>* that will generate the harness for this single package.

From a project contextual menu (*The Project view*), you have an option *GNATtest/Generate unit test setup for <project>* that will generate the harness for the project. An option *GNATtest/Generate unit test setup for <project> recursive* will generate harness for whole hierarchy of projects. If harness project already exists, an option “GNATtest/Open harness project” will switch GPS to harness project.

While harness project is opened it's easy to navigate from tested routine to test code and back. Pointing to name of tested routine provides options *GNATtest/Go to test case*, *GNATtest/Go to test setup* and *GNATtest/Go to test*

teardown. From contextual menu for source file of test case or setup/teardown, you have an option called *GNATtest/Go to <routine>* to go to tested routine.

10.6.3 Project Properties

Gnatstest's behaviour could be configured through project properties. GNATtest page in (*The Project Properties Editor*) gives you convenient access to these properties.

10.7 Metrics

GPS provides an interface with the GNAT software metrics generation tool *gnatmetric*.

The metrics can be computed for the one source file, for the current project, or for the current project and its imported subprojects

The metrics generator can be invoked either from the *Tools->Metrics* menu or from the contextual menu.

10.7.1 The Metrics Menu

The *Metrics* submenu is available from the *Tools* global menu and contains:

Compute metrics for current file Generate metrics for the current source file.

Compute metrics for current project Generate metrics for all files from the current project.

Compute metrics for current project and subprojects Generate metrics for all files from the current project and sub-projects.

10.7.2 The Contextual Menu

When relevant (depending on the context), right-clicking with your mouse will show metrics-related contextual menu entries.

From a source file contextual menu, you have an option called *Metrics for file* that will generate the metrics for the current file.

From a project contextual menu (*The Project view*), you have an option *Metrics for project* that will generate the metrics for all files in the project.

After having computed metrics, a new window in the left-side area is displayed showing the computed metrics as a hierarchical tree view. The metrics are arranged by files, and then by scopes inside the files in a nested fashion. Double-clicking on any of the files or scopes displayed will open the appropriate source location in the editor. Any errors encountered during metrics computation will be displayed in the Locations Window.

10.8 Code Coverage

GPS provides a tight integration with Gcov, the GNU code coverage utility.

Code coverage information can be computed from, loaded and visualized in GPS. This can be done file by file, for each files of the current project, project by project (in case of dependencies) or for the entire project hierarchy currently used in GPS.

Once computed then loaded, the coverage information is summarized in a graphical report (shaped as a tree-view with percentage bars for each item) and used to decorate source code through mechanisms such as line highlighting or coverage annotations.

All the coverage related operations are reachable via the *Tools->Coverage* menu.

In order to be loaded in GPS, the coverage information need to be computed before, using the *Tools->Coverage->Gcov->Compute coverage files* menu for instance.

At each attempt, GPS automatically tries to load the needed information and reports errors for missing or corrupted `.gcov` files.

To be able to produce coverage information from Gcov, your project must have been compiled with the `-fprofile-arcs` and `-ftest-coverage` switches, respectively “Instrument arcs” and “Code coverage” entries in *The Project Properties Editor*, and run once.

10.8.1 Coverage Menu

The *Tools->Coverage* menu has a number of entries, depending on the context:

Gcov->Compute coverage files Generates the `.gcov` files of current and properly compiled and run projects.

Gcov->Remove coverage files Deletes all the `.gcov` of current projects.

Show report Open a new window summarizing the coverage information currently loaded in GPS.

Load data for all projects Load or re-load the coverage information of every projects and subprojects.

Load data for project ‘XXX’ Load or re-load the coverage information of the project `XXX`.

Load data for :file:‘xxxxxxx.xxx’ Load or re-load the coverage information of the specified source file.

Clear coverage from memory Drop every coverage information loaded in GPS.

10.8.2 The Contextual Menu

When clicking on a project, file or subprogram entity (including the entities listed in the coverage report), you have access to a *Coverage* submenu.

This submenu contains the following entries, adapted to the entity selected. For instance, if you click on a file, you will have:

Show coverage information Append an annotation column to the left side of the current source editor. This column indicates which lines are covered and which aren’t. Unexecuted lines are also listed in the *The Locations view*.

Hide coverage information Withdraw from the current source editor a previously set coverage annotation column and clear *The Locations view* from the eventually listed uncovered lines.

Load data for :file:‘xxxxxxx.xxx’ Load or re-load the coverage information of the specified source file.

Remove data of :file:‘xxxxxxx.xxx’ Remove the coverage information of the specified source file from GPS memory.

Show Coverage report Open a new window summarizing the coverage information. (This entry appears only if the contextual menu has been created from outside of the Coverage Report.)

10.8.3 The Coverage Report

When coverage information is loaded, a graphical coverage report is displayed. This report contains a tree of Projects, Files and Subprograms with corresponding coverage information for each node in sided columns.

Report of Analysis 1		
Entities	Coverage	Coverage Percentage
▼ Sdc	142 lines (99 not covered), called 2 times	30 %
except.ads	4 lines (0 not covered)	100 %
input.adb	Gcov file corrupted	n/a
▶ input.ads	3 lines (2 not covered)	33 %
▶ instructions.adb	14 lines (14 not covered)	0 %
▶ instructions.ads	2 lines (2 not covered)	0 %
screen_output.adb	No Gcov file found	n/a
screen_output.ads	No Gcov file found	n/a
▶ sdc.adb	23 lines (11 not covered)	52 %
sdc.ads	No Gcov file found	n/a
▶ stack.adb	30 lines (24 not covered)	20 %
stack.ads	3 lines (0 not covered)	100 %
▼ tokens.adb	22 lines (15 not covered)	31 %
○ Next	17 lines (13 not covered), called 2 times	23 %
○ Process	5 lines (2 not covered), called 2 times	60 %
tokens.ads	5 lines (4 not covered)	20 %
▶ values-operations.adb	22 lines (22 not covered)	0 %
▶ values-operations.ads	2 lines (2 not covered)	0 %
▼ values.adb	9 lines (1 not covered)	88 %
○ Process	2 lines (0 not covered), called 2 times	100 %
○ Read	5 lines (1 not covered), called 2 times	80 %
○ To_String	2 lines (0 not covered), called 2 times	100 %
▶ values.ads	3 lines (2 not covered)	33 %

The contextual menus generated on this widget contain, in addition to the regular entries, some specific Coverage Report entries.

These entries allow you to expand or fold the tree, and also to display flat lists of files or subprograms instead of the tree. A flat list of file will look like:

Report of Analysis 1		
Entities	Coverage	Coverage Percentage
 except.ads	4 lines (0 not covered)	100 %
 input.adb	Gcov file corrupted	n/a
 input.ads	3 lines (2 not covered)	33 %
 instructions.adb	14 lines (14 not covered)	0 %
 instructions.ads	2 lines (2 not covered)	0 %
 screen_output.adb	No Gcov file found	n/a
 screen_output.ads	No Gcov file found	n/a
 sdc.adb	23 lines (11 not covered)	52 %
 sdc.ads	No Gcov file found	n/a
 stack.adb	30 lines (24 not covered)	20 %
 stack.ads	3 lines (0 not covered)	100 %
 tokens.adb	22 lines (15 not covered)	31 %
 tokens.ads	5 lines (4 not covered)	20 %
 values-operations.adb	22 lines (22 not covered)	0 %
 values-operations.ads	2 lines (2 not covered)	0 %
 values.adb	9 lines (1 not covered)	88 %
 values.ads	3 lines (2 not covered)	33 %

GPS and Gcov both support many different programming languages, and so code coverage features are available in GPS for many languages. But, note that subprogram coverage details are not available for every supported languages.

Note also that if you change the current main project in GPS, using the *Project->Open* menu for instance, you will also drop every loaded coverage information as they are related to the working project.

10.9 Stack Analysis

GPS provides an interface to *GNATstack*, the static stack analysis tool. This interface is enabled only if you have the *gnatstack* executable installed on your system and available on the path.

Stack usage information can be computed from, loaded and visualized in GPS for the entire project hierarchy used in GPS. Stack usage information for unknown and unbounded calls can be edited in GPS.

Once computed and loaded, the stack usage information is summarized in a report, and used to decorate source code through stack usage annotations. The largest stack usage path is filled into the *The Locations view*.

Stack usage information for undefined subprograms can be specified by adding a `.ci` file to the set of GNATStack switches in the *Switches* attribute of the *Stack* package of your root project, e.g:

```
project P is
  package Stack is
    for Switches use ("my.ci");
  end Stack;
end P;
```

You can also specify this information by using the *GNATStack* page of the *Switches* section in the *The Project Properties Editor*. Several files can be specified.

The Stack Usage Editor can be used to edit stack usage information for undefined subprograms.

10.9.1 The Stack Analysis Menu

All stack analysis related operations are reachable via the *Tools->Stack Analysis* menu:

Analyze stack usage Generates stack usage information for the root project.

Open undefined subprograms editor Opens undefined subprograms editor.

Load last stack usage Loads or re-loads last stack usage information for the root project.

Clear stack usage data Removes stack analysis data loaded in GPS and any associated information such as annotations in source editors.

10.9.2 The Contextual Menu

When clicking on a project, file or subprogram entity (including the entities listed in the coverage report), you have access to a *Stack Analysis* submenu.

This submenu contains the following entries, related to the entity selected:

Show stack usage Shows stack usage information for every subprogram of currently selected file.

Hide stack usage Hides stack usage information for every subprogram of currently selected file.

Call tree for xxx Opens Call Tree view for currently selected subprogram.

10.9.3 The Stack Usage Report

When the stack usage information is loaded, a report is displayed containing a summary of the stack analysis.

10.9.4 The Stack Usage Editor

The Stack Usage Editor allows to specify stack usage for undefined subprograms and use these values to refine results of future analysis.

Subprogram	Stack size
<__gnat_install_handler>	18
<_Unwind_Resume>	20

Subprogram	Stack size
<__gnat_break_start>	-2
<__gnat_finalize>	-2
<__gnat_initialize>	-2
<__gnat_rcheck_03>	-2
<__main>	-2
system.exception_table'Elab_Body	-2
system.secondary_stack'Elab_Body	-2
system.soft_links'Elab_Body	-2
system.standard_library.adafinal	-2

The Stack Usage Editor consists of two main areas. The notebook in the top area allows to select the file to edit. It displays the contents of the file and allows changing the stack usage of subprograms. The table in the bottom area displays all subprograms whose stack usage information is not specified so that they can be set.

Stack usage information for subprograms can be specified or changed by clicking in the stack usage column on the right of the subprogram's name. When a value is specified in the bottom area table, the subprogram is moved to the top table of the currently selected file. When a negative value is specified, the subprogram is moved to the bottom table.

All changes are saved when the stack usage editor window is closed.

WORKING IN A CROSS ENVIRONMENT

This chapter explains how to adapt your project and configure GPS when working in a cross environment.

11.1 Customizing your Projects

This section describes some possible ways to customize your projects when working in a cross environment. For more details on the project capabilities, see *Project Handling*.

When using the project editor to modify the project's properties, two areas are particularly relevant to cross environments: *Cross environment* part of the *General* page, and *Toolchains* part of the *Languages* page.

In the *Toolchains* section, you will typically either scan your system to display found toolchains, and select the one corresponding to your cross environment or use the Add button and manually select the desired cross environment.

If needed, you can also modify manually some of the tools defined in this toolchain in the *Details* part of the *Languages* page.

For example, assuming you have an Ada project, and using a powerpc VxWorks configuration. Hitting the scan button, you should see the toolchain *powerpc-wrs-vxworks* appearing in the *Toolchains* section. Selecting this toolchain will change the *Details* part, displaying the relevant tools (e.g. *Gnatls* to *powerpc-wrs-vxworks-gnatls* and *Debugger* to *powerpc-wrs-vxworks-gdb* ...).

The list of toolchains and their default values that can be selected when using the Add button can be modified via a custom xml file. See *Customizing and Extending GPS* and in particular *Toolchains customization* for further information.

If you are using an alternative run time, e.g. a *soft float* run time, you need to add the option *-RTS=soft-float* to the *Gnatls* property, e.g: *powerpc-wrs-vxworks-gnatls -RTS=soft-float*, and add this same option to the *Gnatmake* switches in the switch editor. See *Switches* for more details on the switch editor.

To modify your project to support configurations such as multiple targets, or multiple hosts, you can create scenario variables, and modify the setting of the Toolchains parameters based on the value of these variables. See *Scenarios and Configuration Variables* for more information on these variables.

For example, you may want to create a variable called *Target* to handle the different kind of targets handled in your project:

Target Native, Embedded

Target Native, PowerPC, M68K

Similarly, you may define a *Board* variable listing the different boards used in your environment and change the *Program host* and *Protocol* settings accordingly.

In some cases, it is useful to provide a different body file for a given package (e.g. to handle target specific differences). A possible approach in this case is to use a configuration variable (e.g. called *TARGET*), and specify a different naming scheme for this body file (in the project properties, *Naming* tab), based on the value of *TARGET*.

11.2 Debugger Issues

This section describes some debugger issues that are specific to cross environments. You will find more information on debugging by reading *Debugging*.

To connect automatically to the right remote debug agent when starting a debugging session (using the menu *Debug->Initialize*), be sure to specify the *Program host* and *Protocol* project properties, as described in the previous section.

For example, if you are using the *Tornado* environment, with a target server called *target_ppc*, set the *Protocol* to *wtx* and the *Program host* to *target_ppc*.

Once the debugger is initialized, you can also connect to a remote agent by using the menu *Debug->Debug->Connect to Board...* This will open a dialog where you can specify the target name (e.g. the name of your .. index:: board board or debug agent) and the communication protocol.

In order to load a new module on the target, you can select the menu *Debug->Debug->Load File...*

If a module has been loaded on the target and is not known to the current debug session, use the menu *Debug->Debug->Add Symbols...* to load the symbol tables in the current debugger.

Similarly, if you are running the underlying debugger (gdb) on a remote machine, you can specify the name of this machine by setting the *Tools host* field of the project properties.

USING GPS FOR REMOTE DEVELOPMENT

In a network environment, it is common for programmers to use a desktop computer that is not directly suitable for their development tasks. For example, each developer may have a desktop PC running Windows or GNU/Linux as their main entrypoint to the company network. They may do all their actual development work using project resources shared on networked servers. These remote servers may also be running an operating system that is different from the one on their desktop machine.

A typical way of operating in such an environment is to access the server through a remote windowing system such as X-Window. GPS does indeed work in such a context but it is not necessarily the most efficient organization. Running GPS remotely on a shared server will increase the workload of the server as well as the traffic on the network. When the network is slow or saturated, user interactions can become uncomfortably sluggish. This is unfortunate because the desktop used to access the network is often a powerful PC that remains idle most of the time. To address this situation, GPS offers the option to run natively on the desktop, with compilation, run and/or debug activities performed transparently on one or more remote servers.

12.1 Requirements

In order to compile, run or debug on a host remote from GPS, three conditions must be met:

- Have a remote connection to the host using ‘rsh’, ‘ssh’ or ‘telnet’. Note that GPS can now handle passwords for such connections.
- Have either a Network Filesystem (i.e. NFS, SMB or equivalent) sharing the project files between the host and the target, or have rsync installed on both client and server. Note that rsync can be found at <http://www.samba.org/rsync/> for unix, and comes as part of cygwin under Windows: <http://www.cygwin.com>.
- Subprojects must be ‘withed’ by the main project using relative paths, or the same absolute paths must exist on the machines involved.

The full remote development setup is performed in two broad steps:

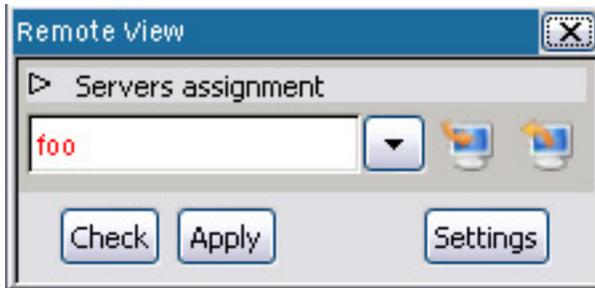
- Setup the remote servers configuration.
- Setup a remote project.

12.2 Setup the remote servers

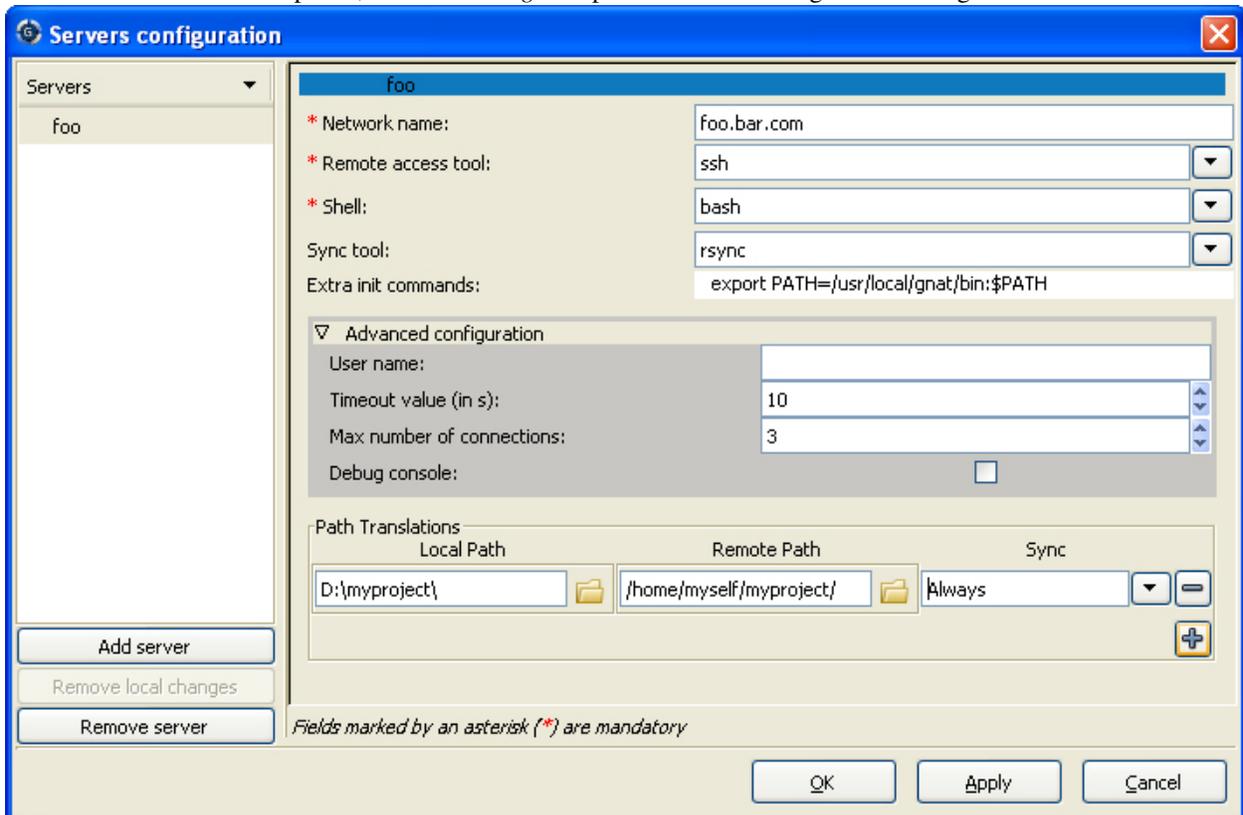
12.2.1 The remote configuration dialog

In order to configure remote servers, you need to open the remote configuration dialog. A predefined configuration can also be set when installing GPS, using xml files. *Defining a remote server*, and *Defining a remote path translation*, for more information.

The remote configuration dialog is opened via the remote view. You can open it using the menu *Tools->Views->Remote*.



Once the Remote View is opened, click on *Settings* to open the servers configuration dialog.



This dialog is composed of two parts:

- The left part of the dialog contains the list of configured servers, identified by their nickname. Three buttons allow you to create, reinitialize or delete a server.
- The right part of the dialog contains the selected server's configuration.

You need first to create a new server. For this, click on the button *Add Server* on the bottom left part of the dialog. Enter a nickname identifying the server you want to connect to (this is not necessarily the network name of this server). Note that this nickname identifies the server and therefore must be unique. This new server is then automatically selected, and the right part of the dialog shows its configuration, which is empty for the most part.

12.2.2 Connection settings

The first configuration part that needs to be filled concerns the way we will connect to this server:

You have to enter first all mandatory fields, identified by an asterisk:

- The network name is the name used to connect to this server via your network. It can be either an IP address, a host name of your local network, or a fully qualified network name.
- The remote access tool is the tool used to connect to this server. You select it using the drop down list. The following tools are supported natively by GPS: ssh, rsh, telnet and plink (Windows tool) in ssh, rsh or telnet mode. *Defining a remote connection tool*, if you need to add a specific tool. Note also that if one of those tools is not installed (e.g. is not in your path), then it won't appear in the tools list. Some tools incompatible with GPS will not be displayed either, such as the Microsoft telnet client.
- The shell tells GPS what shell runs on the remote server. The following unix shells are supported by GPS: sh, bash, csh and tcsh. Windows' shell is also supported (cmd.exe). *Limitations*, for cygwin's shell usage on windows: it is preferable to use cmd.exe as a remote shell on Windows servers.

Other fields might need to be taken into consideration, but they are not mandatory. They are, for the most part, accessible through the advanced configuration pane.

- The remote sync tool is used to synchronize remote and local filesystems, if these are not shared filesystems. For now, only rsync is supported.
- The Extra Init Commands field represents initialization commands sent to the server upon connection: when GPS connects to your remote machine, the chosen shell is launched, and your default initialization files are read (i.e. .bashrc file for the bash shell). Then GPS sends these extra init commands, allowing you for example to specify a compilation toolchain.
- (In Advanced configuration pane) The user name specifies the name used to connect to the server. If unspecified, the remote access tool will typically use your current login name. If not, and a user name is requested, GPS will prompt you for a user name.
- (In Advanced configuration pane) The timeout value is used to determine if a connection to a remote host is dead. All elementary operations performed on the remote host (i.e., operations that normally complete almost immediately) will use this timeout value. By default, this value is set to 10s. If you have a very slow network connection or a very overloaded server, set this timeout to a higher value.
- (In Advanced configuration pane) The maximum number of connections determines the maximum number of simultaneous connections GPS is allowed to have to this server. In fact, if you want to compile, debug and execute at the same time on the machine, GPS will need more than one connection to do this. The default value is 3.
- (In Advanced configuration pane) Depending on the kind of server and the remote access tool used, commands sent to the server may require a specific line terminator, i.e., either the LF character or CR/LF characters. Usually GPS can automatically detect what is needed (the 'auto' mode), but the choice can be forced to CR/LF (cr/lf handling set to 'on') or LF (cr/lf handling set to 'off').
- (In Advanced configuration pane) The Debug console allows you to easily debug a remote connection. If checked, it will open a console reporting all exchanges between GPS and the selected server.

12.2.3 Paths settings

The last configuration part defines the path translations between your local host and the remote server.

The remote paths definition will allow GPS to translate your locally loaded project (the project that resides in your local filesystem) to paths used on the remote server. This part also tells GPS how to keep those paths synchronized between the local machine and the remote server.

All your project's dependencies must then reside in a path that is defined here. Note that you can retrieve those paths by using `gnat list -v -Pyour_project`. In particular, the path to the GNAT run-time (`adainclude` directory) needs to be mapped so that code completion and source navigation work properly on run-time entities.

To add a new path, click on the + button, and enter the corresponding local and remote paths.

You can easily select the desired paths by clicking on the icon next to the path's entry. Remote browsing is allowed only when the connection configuration is set (*Connection settings*.) Clicking on *Apply* will apply your connection configuration and allow you to browse the remote host to select the remote paths.

Five kinds of path synchronization can be set for each defined path:

- *Never*: no synchronization is required from GPS, the paths are shared using an OS mechanism like NFS.
- *Manually*: synchronization is needed, but will only be performed manually using the remote view buttons.
- *Always*: Relevant to source and object paths of your project. They are kept synchronised by GPS before and after every remote action (such as performing a build or run).
- *Once to local/Once to remote*: Relevant to project's dependencies. They are synchronized once when a remote project is loaded or when a local project is set remote. They can still be manually synchronized using the Remote View (*The remote view*.)

The way those paths need to be configured depends on your network architecture.

- If your project is on a filesystem that is shared between your host and the remote host (using NFS or SMB filestems, for example), then only the roots of those filesystems need to be specified, using each server's native paths (on Windows, the paths will be expressed using `X:\my\mounted\directory\` while on unix, the paths will be expressed using `/mnt/path/`).
- If the project's files are synchronized using `rsync`, defining a too generic path translation will lead to very slow synchronization. In that case you should define the paths as specifically as possible, in order to speed up the synchronization process.

12.3 Setup a remote project

12.3.1 Remote operations

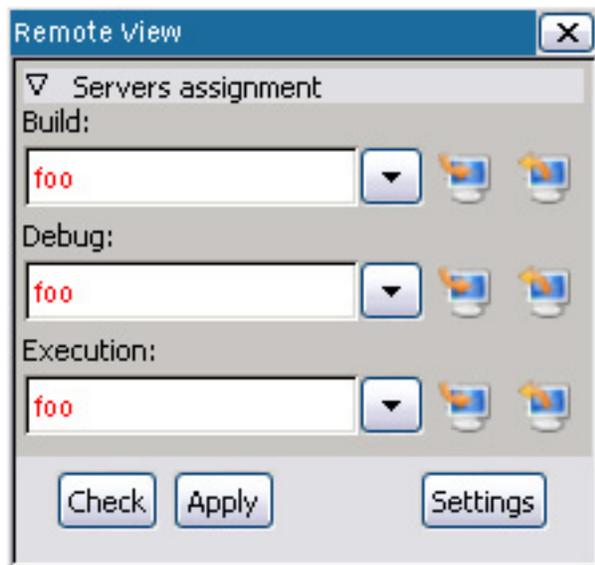
GPS defines four different remote operation categories: Build operations, Debug operations, Execution operations and Tools operations. All compiler related operations are performed on the `Build_Server`. The Tools server is somewhat special and will be explained later. The debugger is run on the `Debug_Server`, and the project's resulting programs are run on the `Execution_Server`. The `GPS_Server` (the local machine) is used for all other operations.

The Tools server is defined to handle all compiler related operations that do not depend on a specific compiler version. It is used in dual compilation mode, for example, to determine whether the action can be safely run using a very recent compiler toolchain (this is the tools server), or whether a specific older baseline compiler version must be used.

In case the remote mode is activated, and the dual compilation mode is not, all Tools server operations are executed on the build server. Otherwise, if the dual compilation mode is activated, then the tools server operations are always executed on the local machine.

12.3.2 The remote view

The Remote view (*Tools->Views->Remote*) allows you to assign servers to operation categories for the currently loaded project. You may assign each operation category a distinct server if the Servers assignment tab is fully expanded. Alternatively, you may assign all categories to a single server in one step if the Servers assignment tab is collapsed.



When a server is selected for a particular category, the change is not immediately effective. To indicate that fact, the server's name will appear in red. This approach allows you to check the configuration before applying it, by pressing the *Check* button. This action will test for correct remote hosts connection. It will also verify that the project path exists on the build server and that it has an equivalence on the local machine.

Clicking on the *Apply* button will perform the following actions:

- Read the default project paths on the Build machine and translate them into local paths.
- Synchronize from the build server those paths marked as *Sync Always* or *Once to local*.
- Load the translated local project.
- Assign the Build, Execution and Debug servers.

If one of the above operations fails, corresponding errors are reported in the *Messages* view and the previous project settings are retained.

Once a remote server is assigned, this remote configuration will be automatically loaded each time the project is loaded.

The two buttons on the right of each server can be used to manually perform a synchronization from the remote host to your local machine (left button) or from your local machine to the remote host (right button).

12.3.3 Loading a remote project

If the project you want to work with is already on a distant server, you can directly load it on your local GPS.

To do this, use the *Project->Open From Host* menu. Then select the server's nickname. This will show you its file tree. Navigate to your project and select it. The project will be loaded as described above, with all remote operations categories assigned to the selected server by default.

You can reload your project using the local files on your machine. The remote configuration will then be automatically reapplied.

12.4 Limitations

The GPS remote mode imposes a few limitations:

- Execution: you cannot use an external terminal to remotely execute your application. The *Use external terminal* checkbox of the run dialog will have no effect if the program is run remotely.
- Debugging: you cannot use a separate execution window. The *Use separate execution window* option is ignored for remote debugging sessions.
- Cygwin on remote host: the GNAT compilation toolchain does not understand cygwin's mounted directories. In order to use GPS with a remote Windows server using cygwin's bash, you need to use directories that are the same on Windows and cygwin (absolute paths). For example, a project having a C:\my_project will be accepted if cygwin's path is /my_project, but will not be accepted if /cygdrive/c/my_project is used.

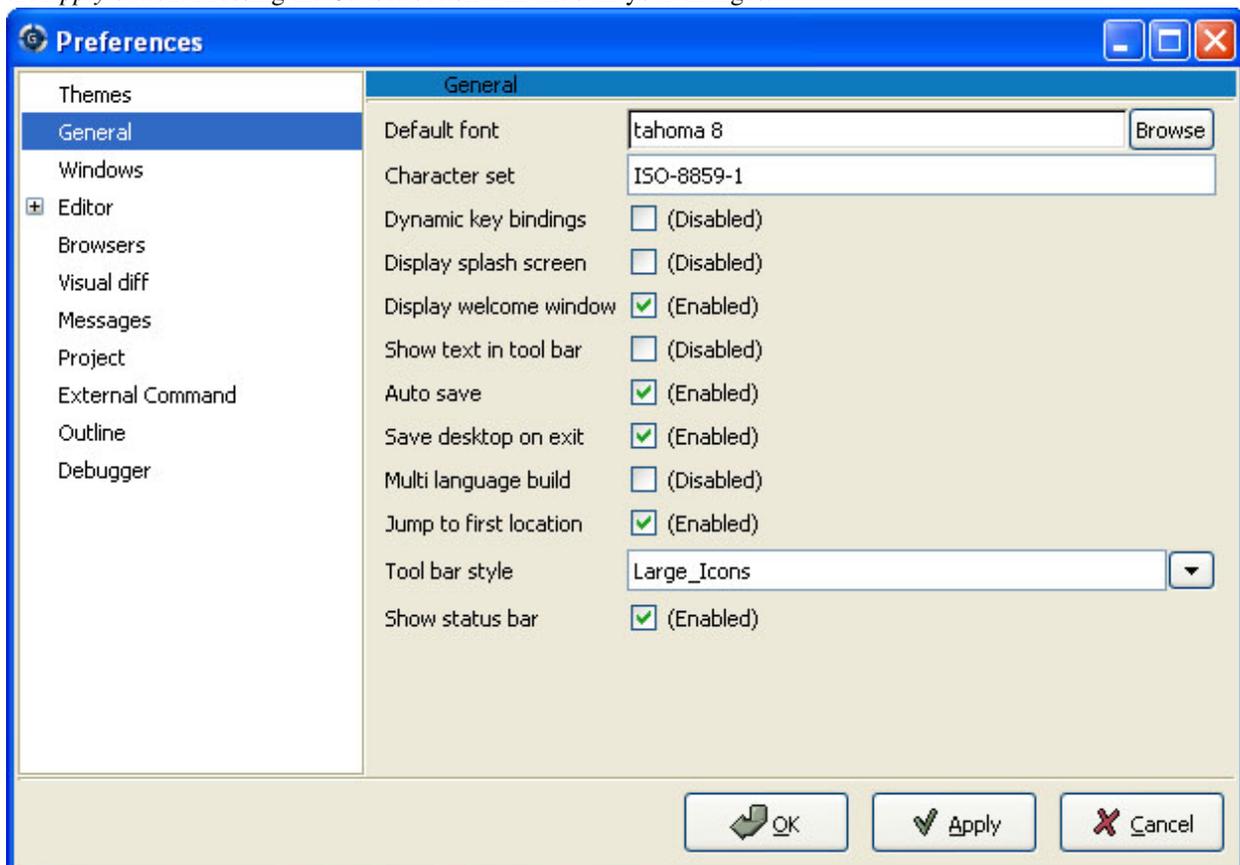
Note that even if you use cygwin's sshd on such a server, you can still access it using cmd.exe (*Connection settings*.)

CUSTOMIZING AND EXTENDING GPS

GPS provides several levels of customization, from simple preferences dialog to powerful scripting capability through the *python* language. This chapters describes each of these capabilities.

13.1 The Preferences Dialog

This dialog, available through the menu *Edit->Preferences*, allows you to modify the global preferences of GPS. To enable the new preferences, you simply need to confirm by pressing the *OK* button. To test your changes, you can use the *Apply* button. Pressing the *Cancel* button will undo all your changes.



Each preference is composed of a label displaying the name of the preference, and an editing area to modify its value. If you leave to mouse over the label, a tool tip will be displayed giving an on-line help on the preference.

The preferences dialog is composed of several areas, accessible through the tabs at the left of the dialog. Each page corresponds to a set of preferences.

- **Themes**

This page allows you to quickly change the current settings for GPS, including preferences, key bindings, menus...; See *GPS Themes* for more information on themes. It is only displayed when there are themes registered.

- **General**

Default font The default font used in GPS. The background color you select for this preference will set the background color for all consoles and most views (the ones that display their data as trees, mostly). To change the background color of editors, see the preference Edit/Fonts&Colors/Default.

Fixed view font The fixed (monospace) font used in views like the outline view, the bookmark view, ...; As much as possible, this font should use a fixed width for characters, for a better rendering

Character set Name of character set to use when reading or writing text files. GPS uses UTF-8 and Unicode internally, which can handle any character in any language. However, your system will generally not support Unicode natively, and thus the contents of the files should be translated from the file system encoding to unicode.

This preference indicates the file system encoding in use. It defaults to ISO-8859-1, which corresponds to western european characters.

Display splash screen Whether a splash screen should be displayed when starting GPS.

Display welcome window Whether GPS should display the welcome window for the selection of the project to use.

Show text in tool bar Whether the tool bar should show both text and icons, or only icons.

Auto save Whether unsaved files and projects should be saved automatically before calling external tools (e.g. before a build).

Save desktop on exit Whether the desktop (size and positions of all windows) should be saved when exiting. If you are working with a project created automatically by GPS, the desktop will not be saved.

Save editor in desktop Determines when source editors should be saved in the desktop: *Never*, *Always*, or when a source file is associated with the current project (*From_Project*).

Default builder The default builder to be used by GPS.

- *Auto* to use *gnatmake* for Ada-only projects and *gprbuild* otherwise (for multi-language and non Ada projects).
- *Gnatmake* to always use *gnatmake* for builds, even for projects that contain other sources. This will disable support for building non Ada projects.
- *Gprbuild* to always use *gprbuild* for builds, even for Ada only projects.

Hyper links Whether to display hyper links in the editors when the Control key is pressed. *Navigating with hyperlinks*.

Clipboard size This controls the size of the list where all the entries copied into the clipboard through *Edit->Copy* and *Edit->Cut* are saved. This list is navigated through the menu *Edit->Paste* and *Edit->Paste Previous*, as described earlier in this guide.

Show status bar Whether the status bar at the bottom of the GPS window should be displayed. This status bar contains one or more progress bars while GPS is executing long actions like a build or a search. These progress bars can be used to monitor the progress of those actions.

If you wish to save vertical screen space, you can hide this status bar. The progress bars will no longer be visible. Instead, you can display the Task Manager through the *Tools->Views->Tasks* menu, to get similar information. This manager can then be put on the right or left side of the GPS window, for instance just below the Project View.

Remove policy when fixing code Preferred way to fix code when parts have to be removed. *Always_Remove* means that the code will be removed by GPS. *Always_Comment* means that the code will always be commented out. *Propose_Both_Choices* will propose a menu with both choices.

Tip of the Day Whether GPS will display a *Tip of the Day* dialog at start up.

- **Windows**

This section specifies preferences that apply to the *Multiple Document Interface* described in *Multiple Document Interface*.

Opaque If True, items will be resized or moved opaquely when not maximized.

Destroy floats If False, closing the window associated with a floating item will put the item back in the main GPS window, but will not destroy it. If True, the item is destroyed.

All floating If True, then all the windows will be floating by default, i.e. be under the control of your system (Windows) or your window manager (Unix machines). This replaces the MDI.

Short titles for floats If True, all floating windows will have a short title. In particular, base file names will be used for editors instead of full names.

Background color Color to use for the background of the MDI.

Title bar color Color to use for the title bar of unselected items.

Selected title bar color Color to use for the title bar of selected items.

Show title bars

If Always, each window in GPS will have its own title bars, showing some particular information (like the name of the file edited for editors), and some buttons to iconify, maximize or close the window. This title bar is highlighted when the window is the one currently selected.

If Never, the title bar is not displayed, to save space on the screen. The tabs of the notebooks will then be highlighted.

If Central Only, then only the windows in the central area (ie the part that gets preserved when switching perspective, mostly editors) will have a title bar. All other windows will not show the title bar. This is often a good way to save space on the screen: the title bar is useful for editors since it gives the full name of the file as well as provide an easy handle for drag and drop operations, whereas the other views do not change position as much and it is better to save space on the screen by not displaying their title.

Notebook tabs policy

Indicates when the notebook tabs should be displayed. If set to “Never”, you will have to select the window in the Window menu, or through the keyboard. If set to “Automatic”, then the tabs will be shown when two or more windows are stacked.

Notebook tabs position

Indicates where the notebook tabs should be displayed by default. It is possible to select the position of tabs individually for each notebook by right-clicking in any of their tabs and choosing a new position in the contextual menu. This position will be saved as part of the desktop and restored the next time you restart GPS. However, if you change the value of this preference, all notebooks will reset the position of their tabs to match the new value of the preference.

- **Editor** .. index:: editor

General

Strip blanks Whether the editor should remove trailing blanks when saving a file.

Line terminator Choose between *Unix*, *Windows* and *Unchanged* line terminators when saving files. Choosing *Unchanged* will use the original line terminator when saving the file; *Unix* will use LF line terminators; *Windows* will use CRLF line terminators.

Display line numbers Whether the line numbers should be displayed in file editors.

Display subprogram names Whether the subprogram name should be displayed in the editor's status bar.

Tooltips Whether tool tips should be displayed automatically.

Tooltips timeout Time (in milliseconds) before displaying tooltips.

Highlight delimiters Determine whether the delimiter matching the character following the cursor should be highlighted. The list of delimiters includes: `{ } [] ()`

Autosave delay The period (in seconds) after which an editor is automatically saved, 0 if none.

Each modified file is saved under a file called `.#filename#`, which is removed on the next explicit save operation.

Right margin The right margin to highlight. 0 if none. This value is also used to implement the `Edit->Refill` command.

Block highlighting Whether the editor should highlight the current block. The current block depends on the programming language, and will include e.g. procedures, loops, if statements, ...

Block folding Whether the editor should provide the ability to fold/unfold blocks.

Speed Column Policy When the Speed Column should be shown on the side of the editors:

Never The Speed Column is never displayed.

Automatic The Speed Column is shown whenever lines are highlighted in the editor, for example to show the current execution point, or lines containing compilation errors, ...; It disappears when no lines are highlighted.

Always The Speed Column is always displayed.

Use Windows ACL This is a Windows specific preference which is disabled by default. When enabled GPS will use the ACL to change the file's write permission. Note that ACL can't be used on network drives.

External editor The default external editor to use.

Custom editor command Specify the command line for launching a custom editor. It is assumed that the command will create a new window/terminal as needed. If the editor itself does not provide this capability (such as vi or pico under Unix systems), you can use an external terminal command, e.g:

```
xterm -geo 80x50 -exe vi +%l %f
```

The following substitutions are provided:

`%l` line to display

`%c` column to display

%f full pathname of file to edit

%e extended lisp inline command

%p top level project file name

%% percent sign ('%')

Always use external editor True if all editions should be done with the external editor. This will deactivate completely the internal editor. False if the external editor needs to be explicitly called by the user.

Smart completion When enabled, GPS loads on startup all the information needed for the Smart completion to work.

Smart completion timeout The timeout, expressed in milliseconds, after which the Smart completion window appears automatically after entering a triggering character, such as '.'

Fonts & Colors

Default The default font, default foreground and default background colors used in the source editor.

Blocks Font variant and colors used to highlight blocks (subprograms, task, entries, ...) in declarations.

Types Font variant and colors used to highlight types in declarations.

Keywords Font variant and colors used to highlight keywords.

Comments Font variant and colors used to highlight comments. Setting the color to white will set a transparent color.

SPARK Annotations Font variant and colors used to highlight SPARK annotations within Ada comments (Starting with `#`). Setting the color to white will set a transparent color.

Ada/SPARK Aspects Font variant and colors used to highlight Ada 2012 and SPARK 2014 aspects. Setting the color to white will set a transparent color.

Strings Font variant and colors used to highlight strings. Setting the color to white will set a transparent color.

Numbers Font variant and colors used to highlight numbers. Setting the color to white will set a transparent color.

Current line color Color for highlighting the current line. Leave it to blank for no highlighting. Setting the color to white will set a transparent color.

Draw current line as a thin line Whether to use a thin line rather than full background highlighting on the current line.

Current block color Color for highlighting the current source block.

Delimiter highlighting color Color for highlighting delimiters.

Search results highlighting Color for highlighting the search results in the text of source editors.

Ada

Auto indentation How the editor should indent Ada sources. None means no indentation; Simple means that indentation from the previous line is used for the next line; Extended means that a language specific parser is used for indenting sources.

Use tabulations Whether the editor should use tabulations when indenting. Note that this preference does not modify the `Tab` key which will still insert `Tab` characters. Consider also the `/Edit/Insert Tab With Spaces` key shortcut which can be mapped (to e.g. `Tab`) via *The Key Manager Dialog*. Finally, another alternative is to reconfigure the default key binding for the

automatic indentation action: by default, it is mapped to `Tab` and can be changed to `Tab` by modifying the `/Edit/Format Selection` action from *The Key Manager Dialog*.

Default indentation The number of spaces for the default Ada indentation.

Continuation lines The number of extra spaces for continuation lines.

Declaration lines The number of extra spaces for multiple line declarations. For example, using a value of 4, here is how the following code would be indented:

```
variable1,  
    variable2,  
    variable3 : Integer;
```

Conditional continuation lines The number of extra spaces used to indent multiple lines conditionals within parentheses.

For example, when this preference is set to 1 (the default), continuation lines are indented based on the previous parenthesis plus one space:

```
if (Condition1  
    and then Condition2)  
then
```

When this preference is set to 3, this gives:

```
if (Condition1  
    and then Condition2)  
then
```

Record indentation The number of extra spaces for record definitions, when the *record* keyword is on its own line.

For example, when this preference is set to 3 (the default), the following sample will be indented as:

```
type T is  
    record  
        F : Integer;  
    end record;
```

When this preference is set to 1, this gives:

```
type T is  
    record  
        F : Integer;  
    end record;
```

Case indentation Whether GPS should indent case statements with an extra level, as used in the Ada Reference Manual, e.g:

```
case Value is  
    when others =>  
        null;  
end case;
```

If this preference is set to *Non_Rm_Style*, this would be indented as:

```

case Value is
when others =>
    null;
end case;

```

By default (*Automatic*), GPS will choose to indent with an extra level or not based on the first *when* construct: if the first *when* is indented by an extra level, the whole case statement will be indented following the RM style.

Casing policy The way the editor will handle the case settings below. *Disabled* no auto-casing will be done; *End_Of_Line* auto-casing will be done when hitting Enter key; *End_Of_Word* auto-casing will be done word-by-word while typing; *On_The_Fly* auto-casing will be done character-by-character while typing. For the *End_Of_Line*, *End_Of_Word* and *On_The_Fly* policies it is always possible to force the casing of the current line by pressing the indentation key (Tab by default).

It is also possible to disable the casing for a single character (action *No Casing/indentation on Next Key*, default Ctrl-Q) or temporarily (action *Toggle Auto Casing/indentation*, default Alt-Q).

Reserved word casing How the editor should handle reserved words casing. *Unchanged* will keep the casing as-is; *Upper* will change the casing of all reserved words to upper case; *Lower* will change the casing to lower case; *Mixed* will change the casing to mixed case (all characters to lower case except first character and characters after an underscore which are set to upper case); *Smart_Mixed* As above but do not force upper case characters to lower case.

Identifier casing How the editor should handle identifiers casing. The values are the same as for the *Reserved word casing* preference.

Format operators/delimiters Whether the editor should add extra spaces around operators and delimiters if needed. If enabled, an extra space will be added when needed in the following cases: before an opening parenthesis; after a closing parenthesis, comma, semicolon; around all Ada operators (e.g. <=, :=, =>, ...)

Align colons in declarations Whether the editor should automatically align colons in declarations and parameter lists. Note that the alignment is computed by taking into account the current buffer up to the current line (or end of the current selection), so if declarations continue after the current line, you can select the declarations lines and hit the reformat key.

Align associations on arrows Whether the editor should automatically align arrows in associations (e.g. aggregates or function calls). See also previous preference.

Align declarations after colon

Whether the editor should align continuation lines in variable declarations based on the colon character.

Consider the following code:

```

Variable : constant String :=
    "a string";

```

If this preference is enabled, it will be indented as follows:

```

Variable : constant String :=
    "a string";

```

Indent comments Whether to indent lines containing only comments and blanks, or to keep these lines unchanged.

Align comments on keywords Whether to align comment lines following *record* and *is* keywords immediately with no extra space.

When enabled, the following code will be indented as:

```
package P is
-- Comment

    [...]
end P;
```

When disabled, the indentation will be:

```
package P is
  -- Comment

  [...]
end P;
```

C & C++

Auto indentation How the editor should indent C/C++ sources. None means no indentation; Simple means that indentation from the previous line is used for the next line; Extended means that a language specific parser is used for indenting sources.

Use tabulations Whether the editor should use tabulations when indenting. If True, the editor will replace each occurrence of eight characters by a tabulation character.

Default indentation The number of spaces for the default indentation.

Extra indentation Whether to indent loops, if and switch statements an extra level. If this preference is enabled, the following layout will be chosen:

```
if (condition)
{
    int x;
}
```

If disabled, the same code will be indented as:

```
if (condition)
{
  int x;
}
```

Indent comments Whether to indent lines containing only comments and blanks, or to keep these lines unchanged.

- **Debugger** .. index:: debugger

Preserve State on Exit If this preference is enabled, the debugger will automatically save breakpoints when it exists, and restore them the next time the same executable is debugged. This is a convenient way to work on an executable, where the typical usage looks like compile, debug, compile, debug, ...

When the preference is enabled, the debugger will also preserve the contents of the data window whenever it is closed. Reopening the window either during the same debugger session, or automatically when a new debugger is started on the same executable, will recreate the same boxes within the data window.

Debugger Windows

This preference controls what happens to debugger-related windows, like the call stack, the data window, the tasks view, ..., when the debugger is terminated. There are three possible behavior:

Close Windows In this case, all these windows are closed. This saves memory and space on the screen, but you will need to explicitly reopen them and put them in the right location on the desktop the next time you start a debugger session.

Keep Windows In this case, the windows are cleared, but kept on the desktop. When you start a new debugger session, the windows will be automatically reused. This ensures that you won't have to reopen and reposition them, but takes space on your screen

Hide Windows The windows are cleared, and hidden. When you start a new debugger session, they are automatically made visible again and reused. This also ensures you will not have to reopen and reposition them, but requires a bit of memory. If you move some windows around while these windows are hidden, they might reappear in unexpected location the next time, although you then just have to move them.

Break on exceptions Specifies whether a breakpoint on all exceptions should be set by default when loading a program. This setup is only taken into account when a new debugger is initialized, and will not modify a running debugger (use the breakpoint editor for running debuggers).

Execution window Specifies whether the debugger should create a separate execution window for the program being debugged.

Note that this preference cannot be taken into account for the current debug session: you need to terminate the current debug session and restart a new one.

If true, a separate console will be created. Under Unix systems, this console is another window in the bottom part of the main window; under Windows, this is a separate window created by the underlying gdb, since Windows does not have the notion of separate terminals (aka ttys).

Note that in this mode under Windows, the *Debug->Interrupt* menu will only interrupt the debugged program with recent versions of gdb. If you are using older versions of gdb, you need to hit `Ctrl-C` in the separate execution window to interrupt it while it is running. Note also that this separate execution window uses the default system-wide console properties (the size of the window, the colors...). It is possible to change those properties using e.g. the default console menu (top-left of the console) on Windows XP.

If false, no execution window will be created. The debugger assumes that the program being debugged does not require input, or that if it does, input is handled outside GPS. For example, when you attach to a running process, this process already has a separate associated terminal.

Show lines with code Specifies whether the source editor should display blue dots for lines that contain code. If set to *False*, gray dots will be displayed instead on each line, allowing breakpoint on any line. Disabling this option provides a faster feedback, since GPS does not need to query the debugger about which lines contain code.

Detect aliases If enabled, do not create new items when an item with the same address is already present on the canvas.

Assembly range size Number of assembly lines to display in the initial display of the assembly window. If the size is 0, then the whole subprogram is displayed, but this can take a very long time on slow machines.

Current assembly line Color used to highlight the assembly code for the current line.

Color highlighting Color used for highlighting in the debugger console.

Clickable item Indicates color to be used for the items that are click-able (e.g pointers).

Changed data Indicates color to be used to highlight fields in the data window that have changed since the last update.

Memory color Color used by default in the memory view window.

Memory highlighting Color used for highlighted items in the memory view.

Memory selection Color used for selected items in the memory view.

Item name Indicates the font to be used for the name of the item in the data window.

Item type Indicates font to be used to display the type of the item in the data window.

Max item width The maximum width an item can have.

Max item height The maximum height an item can have.

- **External Commands** .. index:: helper .. index:: external commands

List processes Command used to list processes running on the machine.

Remote shell Program used to run a process on a remote machine. You can specify arguments, e.g. *rsh -l user*

Remote copy Program used to copy a file from a remote machine. You can specify arguments, e.g. *rcp -l user*

Execute command Program used to execute commands externally.

HTML Browser Only used under Unix, not relevant under Windows where the default HTML browser is used. Program used to execute view HTML files, for instance the documentation. Empty by default, which means that GPS will try to find a suitable HTML browser automatically. Only change the value if GPS cannot find a HTML browser, or if the browser found is not your preferred one.

Print command External program used to print files.

This program is required under Unix systems in order to print, and is set to *a2ps* by default. If *a2ps* is not installed on your system, you can download it from <ftp://ftp.enst.fr/pub/unix/a2ps/>, although other printing programs such as *lp* can be specified instead.

Under Windows systems, this program is optional and is empty by default, since a built-in printing is provided. An external tool will be used if specified, such as the PrintFile freeware utility available from <http://www.lerup.com/printfile/descr.html>

- **Search** .. index:: search

Confirmation for “Replace all” Enable or disable the confirmation popup for the replace all action.

Close on Match If this option is enabled, the search window will be closed when a match is found.

Select on Match If this option is enabled, the focus will be given to the editor when a match is found.

Preserve Search Context If this option is enabled, the contents of the “Look in:” field will be preserved between consecutive searches in files.

- **Browsers** .. index:: browsers

General

Selected item color Color to use to draw the selected item.

Background color Color used to draw the background of the browsers.

Hyper link color Color used to draw the hyper links in the items.

Selected link color Color to use for links between selected items.

Default link color Color used to draw the links between unselected items.

Ancestor items color Color to use for the background of the items linked to the selected item.

Offspring items color Color to use for the background of the items linked from the selected item.

Vertical layout Whether the layout of the graph should be vertical (*True*) or horizontal (*False*). This setting applies to most browsers (call *graph* for instance), but does not apply to the entities browsers.

Show elaboration cycles Whether GPS should display an elaboration graph after each compilation showing an elaboration cycle.

- **VCS** .. index:: vcs

Implicit status Whether a status action can be launched as part of another action. For example to get the revision numbers of new files after an update command. If the network connection with the repository is slow disabling this command can speed-up the VCS actions.

Default VCS The default VCS to use when the project does not define a VCS.

- **Visual diff** .. index:: visual diff .. index:: file comparison

Note that in order to perform visual comparison between files, GPS needs to call external tool (not distributed with GPS) such as *diff* or *patch*. These tools are usually found on most unix systems, and may not be available by default on other OSes. Under Windows, you can download them from one of the unix toolsets available, such as *msys* (<http://www.mingw.org>) or *cygwin* (<http://www.cygwin.com>).

mode How GPS displays visual diffs between two files:

Side_By_Side Editors are displayed side-by-side; new editors are created as needed

Unified No new editor is created, and changes are displayed directly in the reference editor.

Diff command Command used to compute differences between two files. Arguments can also be specified. The visual diff expects a standard diff output with no context (that is, no *-c* nor *-u* switch). Arguments of interest may include (this will depend on the version of diff used):

-b Ignore changes in amount of white space.

-B Ignore changes that just insert or delete blank lines.

-i Ignore changes in case; consider upper and lower case letters equivalent.

-w Ignore white space when comparing lines.

Patch command Command used to apply a patch. Arguments can also be specified. This command is used internally by GPS to perform the visual comparison on versioned files (e.g. when performing a comparison with a version control system).

This command should be compatible with the *GNU patch* utility.

Use old diff Use the old version of the visual comparison.

Diff3 command This item is only displayed if the preference *Use old diff* is disabled. Command used to query a 3-way diff. See *Diff command* for a description of the parameters.

Default color This item is only displayed if the preference *Use old diff* is disabled. The color used to indicate lines on which there is a difference, in the “reference” editor.

Old color This item is only displayed if the preference *Use old diff* is disabled. The color used to indicate spaces used by lines not present in one of the editors in a 3-way diff and present in the other editors.

Append color This item is only displayed if the preference *Use old diff* is disabled. The color used to display the lines that are present in an editor but not in the reference editor.

Remove color This item is only displayed if the preference *Use old diff* is disabled. The color used to display the lines that are present in the reference editor but not in other editors.

Change color This item is only displayed if the preference *Use old diff* is disabled. The color used to display the lines that have changed between the reference editor and the other editors.

Fine change color This item is only displayed if the preference *Use old diff* is disabled. The color used to highlight fine differences within a modified line.

Context length This item is only displayed if the preference *Use old diff* is enabled. The number of lines displayed before and after each chunk of differences. Specifying -1 will display the whole file.

- **Messages .. index:: messages**

Color highlighting Color used to highlight text in the messages window.

Errors highlighting Color used to highlight lines causing compilation errors, in the source editors. When this color is set to white, the errors are not highlighted. (*Compilation/Build*)

Warnings highlighting Color used to highlight lines causing compilation warnings, in the source editors. When this color is set to white, the warnings are not highlighted.

Style errors highlighting Color used to highlight lines containing style errors, in the source editors. When this color is set to white, the errors are not highlighted.

Compiler info highlighting Color used to highlight lines containing compiler information, in the source editors. When this color is set to white, the information is not highlighted.

File pattern Pattern used to detect file locations and the type of the output from the messages window. This is particularly useful when using an external tool such as a compiler or a search tool, so that GPS will highlight and allow navigation through source locations. This is a standard system V regular expression containing from two to five parenthesized subexpressions corresponding to the file, line, column, warnings or style error patterns.

File index Index of filename in the file pattern.

Line index Index of the line number in the file pattern.

Column index Index of the column number in the file pattern.

Warning index Index of the warning identifier in the file pattern.

Style index Index of the style error identifier in the file pattern.

Info index Index of the compiler info identifier in the file pattern.

Secondary File pattern Pattern used to detect additional file locations from the messages window. This is a standard system V regular expression containing from two to three parenthesized subexpressions corresponding to the file, line, and column patterns.

Secondary File index Index of filename in the file pattern.

Secondary Line index Index of the line number in the file pattern.

Secondary Column index Index of the column number in the file pattern.

Alternate Secondary File pattern Pattern used to detect additional file locations in alternate form from the messages window. This is a standard system V regular expression containing one parenthesized subexpressions corresponding to the line patterns.

Alternate Secondary Line index Index of the line number in the file pattern.

- **Project**

Relative project paths Whether paths should be absolute or relative when the projects are modified.

Fast Project Loading If the project respects a number of restrictions, activating the preference will provide major speed up when GPS parses the project. This is especially noticeable if the source files are on a network drive.

GPS assumes that the following restrictions are true when the preference is activated. If this isn't the case, no error is reported, and only minor drawbacks will be visible in GPS (no detection that two files are the

same if one of them is a symbolic link for instance, although GPS will still warn you if you are trying to overwrite a file modified on the disk).

The restrictions are the following:

Symbolic links shouldn't be used in the project. More precisely, you can only have symbolic links that point to files outside of the project, but not to another file in the project

Directories can't have source names. No directory name should match the naming scheme defined in the project. For instance, if you are using the default GNAT naming scheme, you cannot have directories with names ending with ".ads" or ".adb"

Hidden directories pattern A regular expression used to match hidden directories. Such directories are not displayed by default in the project view, and are not taken into account for VCS operations working on directories.

- **Documentation .. _Documentation_Preferences:**

This section specifies preferences that apply to the *Documentation Generator*. [Documentation Generation](#) for more information.

Process body files If this preference is enabled, implementation files will be processed. Otherwise, only the specification files will.

Show private entities By default, no documentation is generated for private entities. Enabling this preference will change this behavior.

Call graph If enabled, the documentation tool will compute and take advantage of source references to e.g. generate call graph information. Activating this option will slow down the documentation generation process.

Up-to-date files only If enabled, only files having up-to-date cross references information will be documented.

Comments filter regexp A regular expression used to filter to comments found in the source code before using them for generating documentation. For example "^.!" will remove all comments starting with '!'.

Spawn a browser If enabled, a browser is spawned after each documentation generation to view the generated files. This browser is not spawned if disabled.

Find xrefs in comments If enabled, GPS will try to find references to entities in comments, and generate links to them when generating the documentation.

- **Coverage Analysis .. _Coverage_Analysis_Preferences:**

Coverage toolchain Select which coverage toolchain (*gcov* or *xcov*) to use from the *Tools->Coverage* menu.

13.2 GPS Themes

GPS provides an extensive support for themes. Themes are predefined set of value for the preferences, for the key bindings, or any other configurable aspect of GPS.

For instance, color themes are a convenient way to change all colors in GPS at once, according to predefined choices (strongly contrasted colors, monochrome,...). It is also possible to have key themes, defining a set of key bindings to emulate e.g. other editors.

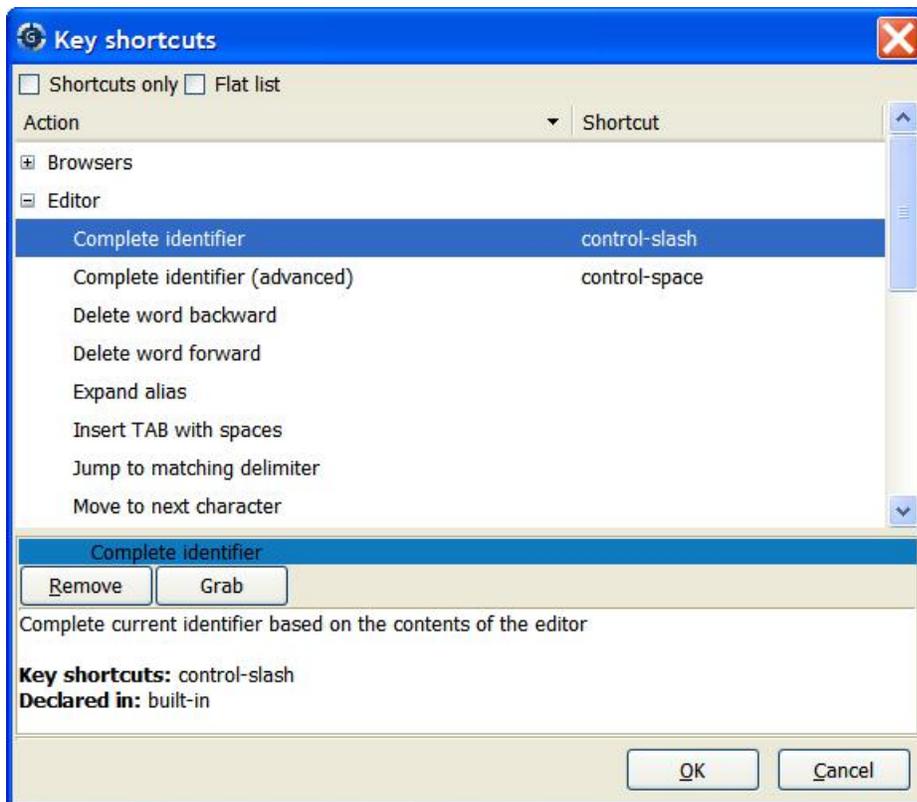
Any number of themes can be activated at the same time through the preferences dialog (*Edit->Preferences*). This dialog contains a list of all themes that GPS knows about, organized into categories for convenient handling. Just click on the buttons on the left of each theme name to activate that theme.

Note that this will immediately change the current preferences settings. For instance, if the theme you just selected changes the colors in the editor, these are changed immediately in the *Editor->Fonts & Colors*. You can of course still press *Cancel* to keep your previous settings

If multiple themes are active at the same time and try to override the same preferences, the last theme which is loaded by GPS will override all previously loaded themes. However, there is no predefined order in which the themes are loaded.

13.3 The Key Manager Dialog

The key manager is accessible through the menu *Edit->Key Shortcuts*. This dialog provides an easy way to associate key shortcuts with actions. These actions are either predefined in GPS, or defined in your own customization files, as documented in *Customizing through XML and Python files*. It also provides an easy way to redefine the menu shortcuts.



Actions are referenced by their name, and are grouped into categories. These categories indicate when the action applies. For instance, the indentation command only applies in source editors, whereas the command to change the current window applies anywhere in GPS. The categories can be explicitly specified when you created your own actions through XML files (*Defining Actions*).

Through the key manager, you can define key bindings similar to what Emacs uses (`control-x` followed by `control-k` for instance). To register such key bindings, you need to press the *Grab* button as usual, and then type the shortcut. The recording of the key binding will stop a short while after the last key stroke.

If you define complex shortcuts for menus, they will not appear next to the menu name when you select it with the mouse. This is expected, and is due to technical limitations in the graphical toolkit that GPS uses.

When you assign a new shortcut to an action, the following happens:

- All actions and menus currently associated with the same key will no longer be executed when the key is pressed.
- All key shortcuts defined for this action are replaced by the new one. As a result, the action is only executable through this new shortcut.

13.4 The Plug-ins Editor

GPS can be extensively customized through external plug-ins. You can write your own plug-ins (*Customization files and plug-ins*), but GPS also comes with its own collection of plug-ins.

Some of them are loaded by default when GPS starts (for instance the support for the CVS version management system or support for highlighting in various programming languages); others are available for any user but not loaded automatically by GPS, for instance an Emacs emulation mode.

Among the plug-ins that are provided with GPS, you will find:

- Emacs emulation .. index:: Emacs

Several plug-ins emulate some of the functions provided by Emacs, such as the interactive search, manipulation of rectangles, navigation in the editor, and of course the usual Emacs key shortcuts

This emacs mode used to be activated in the preferences dialog, on the Themes page, but you should now activate it by loading the `emacs.xml` plug-in.

- Makefile support .. index:: Makefile

A plug-in is provided that parses a Makefile and creates menus for each of its possible targets, so that you can easily start a make command.

- Cross-references enhancements

Various plug-ins take advantage of GPS's cross-references information to create additional menus to navigate (for instance to jump to the primitive operations of Ada tagged types, to the body of Ada separate entities, ...)

- Text manipulation

Several plug-ins provide support for advanced text manipulation in the editors, for instance to be able to align a set of lines based on various criteria, or to manipulate a rectangular selection of text.

You can choose graphically which plug-ins should or should not be loaded on startup. To do so, select the menu */Tools/Plug-ins*. This brings up a new window, containing two parts:

- On the left is the list of all known plug-ins.

As described in *Customization files and plug-ins*, GPS will search for candidates in various directories, and based on these directories decide whether to automatically load the plug-in or not.

This list indicates the name of the plug-in, and whether it has been loaded in this GPS session (when the toggle button is checked).

- On the right are the details for the selected plug-in.

This window is displayed as a notebook with two pages: on the first one you will see the exact location of the plug-in, the reason why it was loaded or not, and, more importantly, the source of the plug-in. By convention, each plug-in starts with a general comment that indicates the purpose of this plug-in, and some more detailed documentation on its usage.

For those interested, this also contains the plug-in itself, so that this can act as an example to create your own customization script.

Technically, the list of plug-ins to load or not to load are stored in the file `HOME/.gps/startup.xml`.

If you have modified anything through this dialog (the list of plug-ins to load or unload), you will need to restart GPS. GPS cannot unload a module, since it can have too many possible effects on GPS (adding menus, overriding key shortcuts, ...).

A dialog is displayed asking you whether you would like to exit GPS now. This will properly save all your files.

13.5 Customizing through XML and Python files

13.5.1 Customization files and plugins

You can customize lots of capabilities in GPS using files that are loaded by GPS at start up.

For example, you can add items in the menu and tool bars, as well as defining new key bindings, new languages, new tools, ...; Using Python as a programming language, you can also add brand new facilities and integrate your own tools in the GPS platform.

These customization files are searched for at startup in several different places. Depending on the location where they are found, these files will either be automatically loaded by GPS (and thus can immediately modify things in GPS), or will simply be made visible in the Plug-ins Editor (*The Plug-ins Editor*).

These directories are searched for in the order given below. Any script loaded latter can override setups done by previously loaded scripts. For instance, they could override a key shortcut, remove a menu, redefine a GPS action, ...

In the directory names below, `INSTALL` is the name of the directory in which you have installed GPS. `HOME` is the user's home directory, either by default or as overridden by the `GPS_HOME` environment variable. If none of these exists, GPS will use the `USERPROFILE` environment variable.

In all these directories, only the files with `.xml` or `.py` extensions are taken into account. Other files are ignored, although for compatibility with future versions of GPS it is recommended not to keep other files in the same directory.

- Automatically loaded system wide modules

The `INSTALL/share/gps/plugin-ins` directory should contain the files that GPS will automatically load by default (unless overridden by the user through the Plug-ins Editor). These plug-ins are visible to any user on the system that uses the same GPS installation. This directory should be reserved for critical plug-ins that almost everyone should use.

- Not automatically loaded system wide modules

The `INSTALL/share/gps/library` directory should contain the files that GPS should show in the Plug-ins Editor, but not load automatically. Typically, these would be files that add optional capabilities to GPS, for instance an emacs emulation mode, or additional editor capabilities that a lot of users would not generally use.

- `GPS_CUSTOM_PATH`

This environment variable can be set before launching GPS. It should contain a list of directories, separated by semicolons (;) on Windows systems and colons (:) on Unix systems. All the files in these directories with the appropriate extensions will be automatically loaded by default by GPS, unless overridden by the user through the Plug-ins Editor.

This is a convenient way to have project-specific customization files. You can for instance create scripts, or icons, that set the appropriate value for the variable and then start GPS. Depending on your project, this allows you to load specific aliases which do not make sense for other projects.

- Automatically loaded user directory

The directory `HOME/.gps/plugin-ins` is searched last. Any script found in there will be automatically loaded unless overridden in the Plug-ins Editor.

This is a convenient way for users to create their own plug-ins, or test them before they are made available to the whole system by copying them to one of the other directories.

Any script loaded by GPS can contain customization for various aspects of GPS, mixing aliases, new languages or menus, ... in a single file. This is a convenient way to distribute your plug-ins to other users.

Python files

Although the format of the python plug-ins is free (as long as it can be executed by Python), the following organization is suggested. These plug-ins will be visible in the Plug-ins Editor, and therefore having a common format makes it easier for users to understand the goal of the plug-ins:

- Comment

The first part of the script should be a general comment on the goal and usage of the script. This comment should use python's triple-quote convention, rather than start-of-line hash ('#') signs.

The first line of the comment should be a one liner explaining the goal of the script. It is separated by a blank line from the rest of the comment.

The rest of the comment is free-form.

- Customization variables

If your script can be configured by the user by changing some global variables, they should be listed in their own section, and fully documented. The user can then, through the /Tools/Plug-ins editor change the value of these variables

- Implementation

The implementation should be separated from the initial comment by a form-feed (control-L) character. The startup scripts editor will know not to display the rest of the script on the first page of the editor.

Generally speaking, scripts should avoid executing code as soon as they are loaded. This gives a chance to the user to change the value of global variables or even override functions before the script is actually launched.

The solution is to connect to the "gps_started" hook, as in:

```
^L
#####
## No user customization below this line
#####

import GPS

def on_gps_started (hook_name):
    ... launch the script

GPS.Hook ("gps_started").add (on_gps_started)
```

XML files

XML files must be utf8-encoded by default. In addition, you can specify any specific encoding through the standard `<?xml encoding="..." ?>` declaration, as in the following example:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- general description -->
<submenu>
  <title>encoded text</title>
</submenu>
```

These files must be valid XML files, i.e. must start with the `<?xml?>` tag, and contain a single root XML node, the name of which is left to your consideration. The general format is therefore:

```
<?xml version="1.0" ?>
<root_node>
  ...
</root_node>
```

It is also recommended that the first line after the `<?xml?>` tag contains a general comment describing the purpose and usage of the script. This comment will be made visible in the Plug-ins editor.

The list of valid XML nodes that can be specified under `<root>` is described in later sections. It includes:

- `<action>` (*Defining Actions*)
- `<key>` (*Binding actions to keys*)
- `<submenu>` (*Adding new menus*)
- `<pref>` (*Preferences support in custom files*)
- `<preference>` (*Preferences support in custom files*)
- `<alias>` (*Defining text aliases*)
- `<language>` (*Adding support for new languages*)
- `<button>` (*Adding tool bar buttons*)
- `<entry>` (*Adding tool bar buttons*)
- `<vsearch-pattern>` (*Defining new search patterns*)
- `<tool>` (*Adding support for new tools*)
- `<filter>` (*Filtering actions*)
- `<contextual>` (*Adding contextual menus*)
- `<case_exceptions>` (*Adding casing exceptions*)
- `<documentation_file>` (*Adding documentation*)
- `<doc_path>` (*Adding documentation*)
- `<stock>` (*Adding stock icons*)
- `<project_attribute>` (*Defining project attributes*)
- `<remote_machine_descriptor>` (*Defining a remote server*)
- `<remote_path_config>` (*Defining a remote path translation*)
- `<remote_connection_config>` (*Defining a remote connection tool*)
- `<rsync_configuration>` (*Configuring rsync usage*)

13.5.2 Defining Actions

This facility distinguishes the actions from their associated menus or key bindings. Actions can take several forms: external commands, shell commands and predefined commands, as will be explained in more details below.

The general form to define new actions is to use the `<action>` tag. This tag accepts the following attributes:

name (mandatory) This tag must be specified. It provides the name by which the action is referenced in other parts of the customization files, for instance when it is associated with a menu or a toolbar button. The name can contain any character, although it is recommended to avoid XML special characters. It mustn't start with a '/

output (optional) If specified, this attribute indicates where the output of the commands will be sent by default. This can be overridden by each command, using the same attribute for `<shell>` and `<external>` tags, [Redirecting the command output](#).

show-command (optional, default true) If specified, this attribute indicates whether the text of the command itself should be displayed at the same location as its output. Neither will be displayed if the output is hidden. The default is to show the command along with its output. This attribute can be overridden for each command.

show-task-manager (optional, default false) This attribute indicates whether an entry should be created in the task manager to show this command. Associated with this entry is the progress bar indicator, so if you hide the entry, no progress will be shown. On the other hand, several progress bars might be displayed for your action if you show the progress bar here, which might be an issue depending on the context. This attribute can be overridden for each external command.

category (optional, default "General") The category in the keybindings editor (menu *Edit/Key bindings*) in which the action should be shown to the user. If you specify an empty string, the action is considered as an implementation detail, and not displayed in the editor. The user will thus not be able to assign it a keybinding through the graphical user interface (although this is still doable through XML commands).

If you are defining the same action multiple times, the last definition will be kept. However, existing menus, buttons, ... that already reference that action will keep their existing semantic. The new definition will only be used for all new menus created from that point on.

The `<action>` can have one or several children, all of which define a particular command to execute. All of these commands are executed one after the other, unless one of them fails in which case the following commands are not executed.

The following XML tags are valid children for `<action>`.

<external> This defines a command to execute through the system (i.e. a standard Unix or Windows command)

Note for Windows users: like under UNIX, scripts can be called from custom menu. In order to do that, you need to write your script in a `.bat` or `.cmd` file, and call this file as usual. The *external* tag would e.g. look like:

```
<?xml version="1.0" ?>
<external_example>
  <action name="my_command">
    <external>c:\\.gps\\my_scripts\\my_cmd.cmd</external>
  </action>
</external_example>
```

This tag accepts the following attributes:

server (optional) This attribute can be used to execute the external command on a remote server. The accepted values are `"gps_server"` (default), `"build_server"`, `"execution_server"`, `"debug_server"` and `"tools_server"`. [Remote operations](#), for explanation of what these servers are.

check-password (optional) This attribute can be used to tell GPS to check and handle password prompts from the external command. The accepted values are `"false"` (default) and `"true"`.

show-command (optional) This attribute can be used to override the homonym attribute specified for the `<action>` tag.

output (optional) This attribute can be used to override the homonym attribute specified for the `<action>` tag.

progress-regexp (optional) This attribute specifies a regular expression that the output of the command will be checked against. Every time the regular expression matches, it should provide two numeric values that are

used to display the usual progress indicators at the bottom-right corner of the GPS window, as happens during regular compilations.

The name of the action is printed in the progress bar while the action is executing:

```
<?xml version="1.0" ?>
<progress_action>
  <action name="progress" >
    <external
      progress-regexp="(\\d+) out of (\\d+).*$"
      progress-current="1"
      progress-final="2"
      progress-hide="true">gnatmake foo.adb
    </external>
  </action>
</progress_action>
```

progress-current (optional, default is 1) This is the opening parenthesis count index in *progress-regexp* that contains the current step.

progress-final (optional, default is 2) This is the opening parenthesis count index in *progress-regexp* that contains the current last step. This last index can grow as needed. For example, gnatmake will output the number of the file it is currently examining, and the total number of files to be examined. However, that last number may grow up, since parsing a new file might generate a list of additional files to parse later on.

progress-hide (optional, default is true) If this attribute is set to the value “true”, then all the lines that match *progress-regexp* and are used to compute the progress will not be displayed in the output console. For any other value of this attribute, these lines are displayed along with the rest of the output.

show-task-manager (optional, default inherited from ‘<action>’) This attribute indicates whether an entry should be created in the task manager to show this command. Associated with this entry is the progress bar indicator, so if you hide the entry, no progress will be shown. On the other hand, several progress bars might be displayed for your action if you show the progress bar here, which might be an issue depending on the context.

If you have set a value for *progress-regexp*, this will automatically be set to true by default so that the progress bar is indeed displayed in the task manager. You can still override it explicitly for that *<external>* element to force hiding the progress bar.

<on-failure> This tag specifies a group of command to be executed if the previous external command fails. Typically, this is used to parse the output of the command and fill the location window appropriately (*Processing the tool output*).

For instance, the following action spawns an external tool, and parses its output to the location window and the automatic fixing tool if the external tool happens to fail.

In this group of commands the %... and \$... macros can be used (*Macro arguments*):

```
<?xml version="1.0" ?>
<action_launch_to_location>
  <action name="launch tool to location" >
    <external>tool-path</external>
    <on-failure>
      <shell>Locations.parse "%1" category<shell>
      <external>echo the error message is "%2"</external>
    </on-failure>
    <external>echo the tool succeeded with message %1</external>
  </action>
</action_launch_to_location>
```

<shell> As well as external commands, you can use custom menu items to invoke GPS commands using the *shell* tag. These are command written in one of the shell scripts supported by GPS.

This tag supports the same *show-command* and *output* attributes as the **<action>** tag.

The following example shows how to create two actions to invoke the *help* interactive command and to open the file `main.c`:

```
<?xml version="1.0" ?>
<help>
  <action name="help">
    <shell>help</shell>
  </action>
  <action name="edit">
    <shell>edit main.c</shell>
  </action>
</help>
```

By default, commands are expected to be written in the GPS shell language. However, you can specify the language through the *lang* attribute. Its default value is “*shell*”.

The value of this attribute could also be “python”.

When programming with the GPS shell, you can execute multiple commands by separating them with semicolons. Therefore, the following example adds a menu which lists all the files used by the current file, in a project browser:

```
<?xml version="1.0" ?>
<current_file_uses>
  <action name="current file uses">
    <shell lang="shell">File %f</shell>
    <shell lang="shell">File.uses %l</shell>
  </action>
</current_file_uses>
```

<description> This tag contains a description for the command, which is used in the graphical editor for the key manager. *The Key Manager Dialog*.

<filter>, **<filter_and>**, **<filter_or>** This is the context in which the action can be executed, *Filtering actions*.

It is possible to mix both shell commands and external commands. For instance, the following command opens an xterm (on Unix systems only) in the current directory, which depends on the context:

```
<?xml version="1.0" ?>
<xterm_directory>
  <action "xterm in current directory">
    <shell lang="shell">cd %d</shell>
    <external>xterm</external>
  </action>
</xterm_directory>
```

As seen in some of the examples above, some special strings are expanded by GPS just prior to executing the command. These are the “%f”, “%d”,... See below for a full list.

More information on chaining commands is provided in *Chaining commands*.

Some actions are also predefined in GPS itself. This include for instance aliases expansion, manipulating MDI windows, ...; All known actions (predefined and the ones you have defined in your own customization files) can be discovered by opening the key shortcut editor (*Edit->Key shortcuts* menu).

13.5.3 Macro arguments

When an action is defined, you can use macro arguments to pass to your shell or external commands. Macro arguments are special parameters that are transformed every time the command is executed. The following macro arguments are provided.

The equivalent python command is given for all tests. These commands are useful when you are writing a full python script, and want to test for yourself whether the context is properly defined.

%a If the user clicked within the Locations Window, this is the name of the category to which the current line belongs

%builder Replaced by the default builder configured in GPS. This can be e.g. *gnatmake* if your project contains only Ada code, or *gprbuild* for non Ada or multi-language projects. Note: this macro is only available in the commands defined in the Build Manager and the Build Launcher dialogs.

%c This is the column number on which the user clicked. Python equivalent:

```
GPS.current_context().column()
```

%d The current directory. Python equivalent:

```
GPS.current_context().directory()
```

%dk The krunched name of the current directory.

%e Name of the entity the user clicked on. Python equivalent:

```
GPS.current_context().entity().name()
```

%ef Name of the entity the user clicked on, possibly followed by “(best guess)” if there is an ambiguity on which entity it really, for instance because the cross-reference information is not up-to-date.

%E The full path to the executable name corresponding to the target.

%ek Krunched name of the entity the user clicked on. This is the same as *%e*, except long names are shorted as in *%fk*.

%eL Replaced by either an empty string, or *-eL*, depending on whether the *Fast Project Loading* preference is set or not. *-eL* is used by GNAT tools to specify whether symbolink links should be followed or not when parsing projects. Note: this macro is only available in the commands defined in the Build Manager and the Build Launcher dialogs.

%external Replaced by the command line specified in the preference *External Commands->Execute command*.

%f Base name of the currently selected file. Python equivalent:

```
import os.path
os.path.basename(GPS.current_context().file().name())
```

%F Absolute name of the currently opened file. Python equivalent:

```
GPS.current_context().file().name()
```

%fk Krunched base name of the currently selected file. This is the same as *%f*, except that long names are shortened, and their middle letters are replaced by “[...]”. This should be used in particular in menu labels, to keep the menus narrow.

%fp Base name of the currently selected file. If the file is not part of the project tree, or no file is selected, generate an error on the Messages window. Note: this macro is only available in the commands defined in the Build Manager and the Build Launcher dialogs.

%gnatmake Replaced by the *gnatmake* executable configured in your project file.

%gprbuild Replaced by the *gprbuild* command line configured in your project file.

%gprclean Replaced by the default cleaner configured in GPS. This can be e.g. *gnat clean*, or *gprclean*. Note: this macro is only available in the commands defined in the Build Manager and the Build Launcher dialogs.

%GPS Replaced by the home directory for GPS (ie the `.gps` directory in which GPS stores its own configuration files).

%i If the user clicked within the Project View, this is the name of the parent project, ie the one that is importing the one the user clicked on. Note that with this definition of parent project, a given project might have multiple parents. The one that is returned is read from the Project View itself.

%l This is the line number on which the user clicked. Python equivalent:

```
GPS.current_context().line()
```

%o The object directory of the current project.

%O The object directory of the root project.

%p The current project. This is the name of the project, not the project file, ie the `.gpr` extension is not included in this name, and the casing is the one found inside the project file, not the one of the file name itself. If the current context is an editor, this is the name of the project to which the source file belongs. Python equivalent:

```
GPS.current_context().project().name()
```

%P The root project. This is the name of the project, not the project file. Python equivalent:

```
GPS.Project.root().name()
```

%Pb The basename of the root project file.

%Pl The name of the root project, all lower case.

%pp The current project file pathname. If a file is selected, this is the project file to which the source file belongs. Python equivalent:

```
GPS.current_context().project().file().name()
```

%PP The root project pathname. Python equivalent:

```
GPS.Project.root().file().name()
```

%pps This is similar to `%pp`, except it returns the project name prepended with `-P`, or an empty string if there is no project file selected and the current source file doesn't belong to any project. This is mostly for use with the GNAT command line tools. The project name is quoted if it contains spaces. Python equivalent:

```
if GPS.current_context().project():
    return "-P" & GPS.current_context().project().file().name()
```

%PPs This is similar to `%PP`, except it returns the project name prepended with `-P`, or an empty string if the root project is the default project. This is mostly for use with the GNAT command line tools.

%(p|P)[r](d|s)[f] Substituted by the list of sources or directories of a given project. This list is a list of space-separated, quoted names (all names are surrounded by double quotes, for proper handling of spaces in directories or file names).

P the root project.

p the selected project, or the root project if there is no project selected.

r recurse through the projects: sub projects will be listed as well as their sub projects, etc...

d list the source directories.

Python equivalent:

```
GPS.current_context().project().source_dirs()
```

s list the source files.

Python equivalent:

```
GPS.current_context().project().sources()
```

f output the list into a file and substitute the parameter with the name of that file. This file is never deleted by GPS, it is your responsibility to do so.

%s This is the text selected by the user, if a single line was selected. When multiple lines were selected, this returns the empty string

%S This is either the text selected by the user, of the current entity if there is no selection. If the entity is part of an expression (“A.B.C”), then the whole expression is used instead of the entity name.

%switches(tool) Replaced by *IDE’Default_Switches (tool)*, in other words, if you have a tool whose switches are defined via an xml file in GPS, they are stored as *Default_Switches (xxx)* in the *IDE* package and can be retrieved using this macro. The value returned is a list of switches, or an empty list if not set.

Note: This macro is only available in the commands defined in the Build Manager and Build Launcher dialogs.

%T Replaced by the subtarget being considered for building. Depending on the context, this can correspond to e.g. the base filename of a Main source, or makefile targets. Note: this macro is only available in the commands defined in the Build Manager and the Build Launcher dialogs.

%TT Same as **%T**, but returns the full path to main sources rather than the base filename.

%attr(Package’Name[,default]) Replaced by the project attribute *Package’Name*, in other words, the attribute *Name* from the package *Package*. *Package’* is optional if *Name* is a top level attribute (e.g. *Object_Dir*).

If the attribute is not defined in the project, an optional *default* value is returned, or an empty string if not.

Note: This macro is only available in the commands defined in the Build Manager and Build Launcher dialogs, and only supports single string attributes, not lists.

%dirattr(Package’Name[,default]) Replaced by the directory part of an attribute. The attribute is specified as in **%attr** above.

%baseattr(Package’Name[,default]) Replaced by the base name of an attribute. The attribute is specified as in **%attr** above.

%vars Replaced by a list of switches of the form *<variable>=<value>*, where *<variable>* is the name of a scenario variable and *<value>* its current value, as configured in the Scenario View. All the scenario variables defined in the current project tree will be listed. Alternatively, you can also use **%vars(-D)** to generate a list of switches of the form *-D<variable>=<value>*. Note: this macro is only available in the commands defined in the Build Manager and the Build Launcher dialogs.

%X Replaced by a list of switches of the form *-X<variable>=<value>*, where *<variable>* is the name of a scenario variable and *<value>* its current value, as configured in the Scenario View. All the scenario variables defined in the current project tree will be listed. Note: this macro is only available in the commands defined in the Build Manager and the Build Launcher dialogs.

%target Replaced by *-target=<t>* where *<t>* is the build target detected by the toolchain being used.

%% Replaced by the % sign.

Examples:

%Ps Replaced by a list of source files in the root project.

%prs Replaced by a list of files in the current project, and all imported sub projects, recursively.

%prdf Replaced by the name of a file that contains a list of source directories in the current project, and all imported sub projects, recursively.

Another type of macros are expanded before commands are executed: These all start with the \$ character, and represent parameters passed to the action by its caller. Depending on the context, GPS will give zero, one or more arguments to the action. This is in particular used when you define your own VCS system. See also the shell function *execute_action*, which you can use yourself to execute an action and pass it some arguments.

These arguments are the following

\$1, \$2, ... \$n Where n is a number. These are each argument passed to the action

\$1-, \$2-, ... \$n- This represents a string concatenating the specified argument and all arguments after it

\$* This represents a string concatenating all arguments passed to the action

\$repeat This is the number of times the action has been repeated in a row. It will in general be 1 (ie this is the first execution of the action), unless the user has first executed the action “Repeat Next”, which allows automatic repetition of an action.

By default, when the action “Repeat Next” is invoked by the user, it will repeat the following action as many times as the user specified. However, in some cases, either for efficiency reasons or simply for technical reasons, you might want to handle yourself the repeat. This can be done with the following action declaration:

```
<action name="my_action">
  <shell lang="python">if $repeat==1: my_function($remaining + 1)</shell>
</action>

def my_function (count):
    """Perform an action count times"""
    ...
```

Basically, the technique here is to only perform something the first time the action is called (hence the if statement), but pass your shell function the number of times that it should repeat (hence the *\$remaining* parameter).

\$remaining This is similar to \$repeat, and indicates the number of times that the action remains to be executed. This will generally be 0, unless the user has chosen to automatically repeat the action a number of times.

13.5.4 Filtering actions

By default, an action will execute in any context in GPS. The user just selects the menu or key, and GPS tries to execute the action.

It is possible to restrict when an action should be considered as valid. If the current context is incorrect for the action, GPS will not attempt to run anything, and will display an error message for the user.

Actions can be restricted in several ways:

- Using macro arguments (*Macro arguments*). If you are using one of the macro arguments defined in the previous section, anywhere in the chain of commands for that action, GPS will first check that the information is available, and if not will not start running any of the shell commands or external commands for that action.

For instance, if you have specified *%F* as a parameter to one of the commands, GPS will check prior to running the action that there is a current file. This can be either a currently selected file editor, or for instance that the project view is selected, and a file node inside it is also selected.

You do not have to specify anything else, this filtering is automatic

Note however that the current context might contain more information than you expect. For instance, if you click on a file name in the Project View, then the current context contains a file (thus satisfies *%F*), but also contains a project (and thus satisfies *%p* and similar macros).

- Defining explicit filters Explicit restrictions can be specified in the customization files. These are specified through the `<filter>`, `<filter_and>` and `<filter_or>` tags, see below.

These tags can be used to further restrict when the command is valid. For instance, you can use them to specify that the command only applies to Ada files, or only if a source editor is currently selected.

The filters tags

Such filters can be defined in one of two places in the customization files:

- At the toplevel. At the same level as other tags such as `<action>`, `<menu>` or `<button>` tags, you can define named filters. These are general filters, that can be referenced elsewhere without requiring code duplication.
- As a child of the `<action>` tag. Such filters are anonymous, although they provide exactly the same capabilities as the ones above. These are mostly meant for simple filters, or filters that you use only once.

There are three different kinds of tags:

`<filter>` This defines a simple filter. This tag takes no child tag.

`<filter_and>` All the children of this tag are composed together to form a compound filter. They are evaluated in turn, and as soon as one of them fails, the whole filter fails. Children of this tag can be of type `<filter>`, `<filter_and>` and `<filter_or>`.

`<filter_or>` All the children of this tag are composed together to form a compound filter. They are evaluated in turn, and as soon as one of them succeeds, the whole filter succeeds. Children of this tag can be of type `<filter>`, `<filter_and>` and `<filter_or>`.

If several such tags are found following one another under an `<action>` tag, they are combined through “or”, i.e. any of the filters may match for the action to be executed.

The `<filter>`, `<filter_and>` and `<filter_or>` tags accept the following set of common attributes:

name (optional) This attribute is used to create named filters, that can be reused elsewhere in actions or compound filters through the `id` attribute. The name can take any form.

error (optional) This is the error message printed in the GPS console if the filter doesn’t match, and thus the action cannot be executed. If you are composing filters through `<filter_and>` and `<filter_or>`, only the error message of the top-level filter will be printed.

In addition, the `<filter>` has the following specific attributes:

id (optional)

If this attribute is specified, all other attributes are ignored. This is used to reference a named filter previously defined. Here is for instance how you can make an action depend on a named filter:

```
<?xml version="1.0" ?>
<test_filter>
  <filter name="Test filter" language="ada" />
  <action name="Test action" >
    <filter id="Test filter" />
    <shell>pwd</shell>
  </action>
</test_filter>
```

A number of filters are predefined by GPS itself.

Source editor This filter will only match if the currently selected window in GPS is an editor.

Explorer_Project_Node Matches when clicking on a project node in the Project View

Explorer_Directory_Node Matches when clicking on a directory node in the Project View

Explorer_File_Node Matches when clicking on a file node in the Project View

Explorer_Entity_Node Matches when clicking on an entity node in the Project View

File Matches when the current context contains a file (for instance the focus is on a source editor, or the focus is on the Project view and the currently selected line contains file information).

language (optional) This attribute specifies the name of the language that must be associated with the current file to match. For instance, if you specify *ada*, you must have an Ada file selected, or the action won't execute. The language for a file is found by GPS following several algorithms (file extensions, and via the naming scheme defined in the project files).

shell_cmd (optional) This attribute specifies a shell command to execute. The output value of this command is used to find whether the filter matches: if it returns "1" or "true", the filter matches. In any other case, the filter fails.

Macro arguments (%f, %p, ...) are fully supported in the text of the command to execute.

shell_lang (optional) This attribute specifies in which language the shell command above is written. Its default value indicates that the command is written using the GPS shell.

module (optional) This attribute specifies that the filter only matches if the current window was setup by this specific GPS module. For instance, if you specify "Source_Editor", this filter will only match when the active window is a source editor.

The list of module names can be obtained by typing *lsmod* in the shell console at the bottom of the GPS window.

This attribute is mostly useful when creating new contextual menus.

When several attributes are specified for a *<filter>* node (which is not possible with *id*), they must all match for the action to be executed:

```
<?xml version="1.0" ?>
<!-- The following filter will only match if the currently selected
      window is a text editor editing an Ada source file -->
<ada_editor>
  <filter_and name="Source editor in Ada" >
    <filter language="ada" />
    <filter id="Source editor" />
  </filter_and>

  <!-- The following action will only be executed for such an editor -->

  <action name="Test Ada action" >
    <filter id="Source editor in Ada" />
    <shell>pwd</shell>
  </action>

  <!-- An action with an anonymous filter. It will be executed if the
        selected file is in Ada, even if the file was selected through
        the project view -->

  <action name="Test for Ada files" >
    <filter language="ada" />
    <shell>pwd</shell>
  </action>
</ada_editor>
```

13.5.5 Adding new menus

These commands can be associated with menus, tool bar buttons and keys. All of these use similar syntax.

Binding a menu to an action is done through the `<menu>` and `<submenu>` tags.

The `<menu>` tag takes the following attributes:

action (mandatory) This attribute specifies which action to execute when the menu is selected by the user. If no action by this name was defined, no new menu is added. The action name can start with a `'`, in which case it represents the absolute path to a menu to execute instead.

This attribute can be omitted only when no title is specified for the menu to make it a separator (see below).

If a filter is associated with the action through the `<filter>` tag, then the menu will be greyed out when the filter doesn't match. As a result, users will not be able to click on it.

before (optional) It specifies the name of another menu item before which the new menu should be inserted. The reference menu must have been created before, otherwise the new menu is inserted at the end. This attribute can be used to control where precisely the new menu should be made visible.

after (optional) This attribute is similar to *before*, but has a lower priority. If it is specified, and there is no *before* attribute, it specifies a reference menu after which the new menu should be inserted.

It should also have one XML child called `<title>` which specifies the label of the menu. This is really a path to a menu, and thus you can define submenus by specifying something like `"/Parent1/Parent2/Menu"` in the title to automatically create the parent menus if they don't exist yet.

You can define the accelerator keys for your menus, using underscores in the titles. Thus, if you want an accelerator on the first letter in a menu named *File*, set its title as `_File`.

The tag `<submenu>` accepts the following attributes:

before (optional) See description above, same as for `<menu>`

after (optional) See description above, same as for `<menu>`

It accepts several children, among `<title>` (which must be specified at most once), `<submenu>` (for nested menus), and `<menu>`.

Since `<submenu>` doesn't accept the *action* attribute, you should use `<menu>` for clickable items that should result in an action, and `<submenu>` if you want to define several menus with the same path.

You can specify which menu the new item is added to in one of two ways:

- Specify a path in the *title* attribute of `<menu>`
- Put the `<menu>` as a child of a `<submenu>` node This requires slightly more typing, but it allows you to specify the exact location, at each level, of the parent menu (before or after an existing menu).

For example, this adds an item named *mymenu* to the standard *Edit* menu:

```
<?xml version="1.0" ?>
<test>
  <submenu>
    <title>Edit</title>
    <menu action="current file uses">
      <title>mymenu</title>
    </menu>
  </submenu>
</test>
```

The following has exactly the same effect:

```
<?xml version="1.0" ?>
<test>
  <menu action="current file uses">
    <title>Edit/mymenu</title>
```

```

    </menu>
</test>

```

The following adds a new item “stats” to the “unit testing” submenu in “my_tools”:

```

<?xml version="1.0" ?>
<test>
  <menu action="execute my stats">
    <title>/My_Tools/unit testing/stats</title>
  </menu>
</test>

```

The previous syntax is shorter, but less flexible than the following, where we also force the My_Tools menu, if it doesn't exist yet, to appear after the File menu. This is not doable by using only `<menu>` tags. We also insert several items in that new menu:

```

<?xml version="1.0" ?>
<test>
  <submenu after="File">
    <title>My_Tools</title>
    <menu action="execute my stats">
      <title>unit testing/stats</title>
    </menu>
    <menu action="execute my stats2">
      <title>unit testing/stats2</title>
    </menu>
  </submenu>
</test>

```

Adding an item with an empty title or no title at all inserts a menu separator. For instance, the following example will insert a separator followed by a File/Custom menu:

```

<?xml version="1.0" ?>
<menus>
  <action name="execute my stats" />
  <submenu>
    <title>File</title>
    <menu><title/></menu>
    <menu action="execute my stats">
      <title>Custom</title>
    </menu>
  </submenu>
</menus>

```

13.5.6 Adding contextual menus

The actions can also be used to contribute new entries in the contextual menus everywhere in GPS. These menus are displayed when the user presses the right mouse button, and should only show actions relevant to the current context.

Such contributions are done through the `<contextual>` tag, which takes the following attributes:

“action” (mandatory) Name of the action to execute, and must be defined elsewhere in one of the customization files.

If this attribute is set to an empty string, a separator will be inserted in the contextual menu instead. If you specify a reference item with one of the “before” or “after” attribute, the separator will be visible only when the reference item is visible.

“before” (optional, default=“”) If it is specified, this attribute should be the name of another contextual, before which the new menu should appear. The name of predefined contextual menus can be found by looking at the output

of “Contextual.list” in the shell console. The name of the contextual menus you define yourself is the value of the `<title>` child.

There is no guarantee that the new menu will appear just before the referenced menu. In particular, it won't be the case if the new menu is created before the reference menu was created, or if another later contextual menu indicates that it must be displayed before the same reference item.

“**after**” (optional, default=”) Same as “before”, except it indicates the new menu should appear after the reference item.

If both “after” and “before” are specified, only the latter is taken into account.

“**group**” (optional, default=”) Group attribute allows you to create groups of contextual menus that will be put together. Items of the same group appear before all items with a greater group number.

It accepts one child tag, `<Title>` which specifies the name of the menu entry. If this child is not specified, the menu entry will use the name of the action itself. The title is in fact the full path to the new menu entry. Therefore, you can create submenus by using a title of the form “Parent1/Parent2/Menu”.

Special characters can be used in the title, and will be automatically expanded based on the current context. These are exactly the ones described in the section for macros arguments, *Macro arguments*.

The new contextual menu will only be shown if the filters associated with the action match the current context.

For instance, the following example inserts a new contextual menu which prints the name of the current file in the GPS console. This contextual menu is only displayed in source editors. This contextual menu entry is followed by a separator line, visible when the menu is visible:

```
<?xml version="1.0" ?>
<print>
  <action name="print current file name" >
    <filter module="Source_Editor" />
    <shell>echo %f</shell>
  </action>

  <contextual action="print current file name" >
    <Title>Print Current File Name</Title>
  </contextual>
  <contextual action="" after="Print Current File Name" />
</print>
```

13.5.7 Adding tool bar buttons

As an alternative to creating new menu items, you can create new buttons on the tool bar, with a similar syntax, by using the `<button>` tag. As for the `<menu>` tag, it requires an *action* attribute which specifies what should be done when the button is pressed. The button is not created if no such action was created.

Within this tag, the tag `< pixmap>` can be used to indicate the location of an image file (of the type *jpeg*, *png*, *gif* or *xpm*) to be used as icon for the button. An empty `<button>` tag indicates a separator in the tool bar.

A title can also be specified with `<title>`. This will be visible only if the user chooses to see both text and icons (or text only) in the tool bar. This title also acts as a tooltip (popup help message) when the button is displayed as an icon only.

The following example defines a new button:

```
<?xml version="1.0" ?>
<stats>
  <button action="execute my stats">
    <title>stats</title>
    <pixmap>/my_pixmaps/button.jpg</pixmap>
  </button>
</stats>
```

```

</button>
</stats>

```

The `<button>` tag allows you to create a simple button that the user can press to start an action. GPS also supports another type of button, a combo box, from which the user can choose among a list of choices. Such a combo box can be created with the `<entry>` tag.

This tag accepts the following arguments:

***id* (mandatory)** This should be a unique id for this combo box, and will be used later on to refer it, in particular from the scripting languages. It can be any string

***label* (default is "")** The text of a label to display on the left of the combo box. If this isn't specified, no text will be displayed

***on-changed* (default is "")** The name of a GPS action to execute whenever the user selects a new value in the combo box. This action is called with two parameters, the unique id of the combo box and the newly selected text respectively.

It also accepts any number of `<choice>` tags, each of which defines one of the values the user can choose from. These tags accept one optional attribute, "on-selected", which is the name of a GPS action to call when that particular value is selected:

```

<action name="animal_changed">
  <shell>echo A new animal was selected in combo $1: animal is $2</shell>
</action>
<action name="gnu-selected">
  <shell>echo Congratulations on choosing a Gnu</shell>
</action>
<entry id="foo" label="Animal" on-changed="animal_changed">
  <choice>Elephant</choice>
  <choice on-selected="gnu-selected">Gnu</choice>
</entry>

```

A more convenient interface exists for Python, the `GPS.Toolbar` class, which gives you the same flexibility as above, but also gives you dynamic control over the entry, and allows placement of buttons at arbitrary positions in the toolbar. See the python documentation.

13.5.8 Binding actions to keys

All the actions defined above can be bound to specific key shortcuts through the `<key>` attribute. As usual, it requires one `<action>` attribute to specify what to do when the key is pressed. The name of the action can start with a `'/'` to indicate that a menu should be executed instead of a user-defined action.

If the action is the empty string, then instead the key will no longer be bound to any action.

This tag doesn't contain any child tag. Instead, its text contents specify the keyboard shortcut. The name of the key can be prefixed by `control-`, `alt-`, `shift-` or any combination of these to specify the key modifiers to apply.

You can also define multiple key bindings similar to Emacs's by separating them by a space. For instance, `control-x control-k` means that the user should press `control-x`, followed by a `control-k` to activate the corresponding action. This is only possible if the prefix key is not already bound to an action. If it is, you should first unbind it by passing an empty action to `<key>`.

Use an empty string to describe the key binding if you wish to deactivate a preexisting binding. The second example below deactivates the standard binding:

```

<?xml version="1.0" ?>
<keys>

```

```
<key action="expand alias">control-o</key>
<key action="Jump to matching delimiter" />

<!-- Bind a key to a menu -->
<key action="/Window/Close">control-x control-w</key>
</key>
```

Multiple actions can be bound to the same key binding. They will all be executed in turn, followed by any menu for which this key is an accelerator.

When GPS processes a `<key>` tag, it does the following:

- Removes all actions bound to that key. This ensures that if you press the key, any action associated with it by default in GPS or in some other XML file will no longer be executed, and only the last one will be executed.
- Adds the new key to the list of shortcuts that can execute the action. Any existing shortcut on the action is preserved, and therefore there are multiple possible shortcuts for this action.

13.5.9 Preferences support in custom files

Creating new preferences

GPS has a number of predefined preferences to configure its behavior and its appearance. They are all customizable through the Edit->Preferences menu.

However, you might wish to add your own kind of preferences for your extension modules. This can easily be done through the usual GPS customization files. Preferences are different from project attributes (*Defining project attributes*), in that the latter will vary depending on which project is loaded by the user, whereas preferences are always set to the same value no matter what project is loaded.

Such preferences are created with the `<preference>` tag, which takes a number of attributes.

name (mandatory) This is the name of the preference, used when the preference is saved by GPS in the `$HOME/.gps/preferences` file, and to query the value of a preference interactively through the `GPS.Preference` class in the GPS shell or python. There are a few limitation to the form of these names: they cannot contain space or underscore characters. You should replace the latter with minus signs for instance.

page (optional, default is “General”) The name of the page in the preferences editor where the preference can be edited. If this is the name of a non-existing page, GPS will automatically create it. If this is the empty string (“”), the preference will not be editable interactively. This could be used to save a value from one session of GPS to the next, without allowing the user to alter it.

Subpages are references by separating pages name with colons (‘:’).

default (optional, default depends on the type of the preference) The default value of the preference, when not set by the user. This is 0 for integer preferences, the empty string for string preferences, True for boolean values, and the first possible choice for choice preferences.

tip (optional, default is “”) This is the text of the tooltip that appears in the preferences editor dialog.

label (mandatory) This is the name of the preference as it appears in the preferences editor dialog

type (mandatory) This is the type of the preference, and should be one of:

“boolean” The preference can be True or False.

“integer” The preference is an integer. Two optional attributes can be specified for `<preference>`, “minimum” and “maximum”, which define the range of valid values for that integer. Default values are 0 and 10 respectively.

“string” The preference is a string, which might contain any value

“color” The preference is a color name, in the format of a named color such as “yellow”, or a string similar to “#RRGGBB”, where RR is the red component, GG is the green component, and BB is the blue component

“font” The preference is a font

“choices” The preference is a string, whose value is chosen among a static list of possible values. Each possible value is defined in a `<choice>` child of the `<preference>` node.

Here is an example that defines a few new preferences:

```
<?xml version="1.0"?>
<custom>
  <preference name="my-int"
    page="Editor"
    label="My Integer"
    default="30"
    minimum="20"
    maximum="35"
    page="Manu"
    type="integer" />

  <preference name="my-enum"
    page="Editor:Fonts & Colors"
    label="My Enum"
    default="1"
    type="choices" >
    <choice>Choice1</choice>
    <choice>Choice2</choice>  <!-- The default choice -->
    <choice>Choice3</choice>
  </preference>
</custom>
```

The values of the above preferences can be queried in the scripting languages:

- GPS shell:

```
Preference "my-enum"
Preference.get %1
```

- Python:

```
val = GPS.Preference ("my-enum").get ()
val2 = GPS.Preference ("my-int").get ()
```

Setting preferences values

You can force specific default values for the preferences in the customization files through the `<pref>` tag. This is the same tag that is used by GPS itself when it saves the preferences edited through the preferences dialog.

This tag requires on attribute:

name This is the name of the preference of which you are setting a default value. Such names are predefined when the preference is registered in GPS, and can be found by looking at the `$HOME/.gps/preferences` file for each user, or by looking at one of the predefined GPS themes.

It accepts no child tag, but the value of the `<pref>` tag defines the default value of the preference, which will be used unless the user has overridden it in his own preferences file.

Any setting that you have defined in the customization files will be overridden by the user’s preferences file itself, unless the user was still using the default value of that preference.

This `<pref>` tag is mostly intended for use through the themes (*Creating themes*).

13.5.10 Creating themes

You can create your own themes and share them between users. You can then selectively chose which themes they want to activate through the preferences dialog (*GPS Themes*).

Creating new themes is done in the customization files through the `<theme>` tag.

This tag accepts a number of attributes:

name (mandatory) This is the name of the theme, as it will appear in the preferences dialog

description (optional) This text should explain what the text does. It appears in the preferences dialog when the user selects that theme.

category (optional, default is General) This is the name of the category in which the theme should be presented in the preferences dialog. Categories are currently only used to organize themes graphically. New categories are created automatically if you chose one that doesn't exist yet.

This tag accepts any other customization tag that can be put in the customization files. This includes setting preferences (`<pref>`), defining key bindings (`<key>`), defining menus (`<menu>`), ...

If the same theme is defined in multiple locations (multiple times in the same customization file or in different files), their effects will be cumulated. The first definition of the theme seen by GPS will set the description and category for this theme.

All the children tags of the theme will be executed when the theme is activated through the preferences dialog. Although there is no strict ordering in which order the children will be executed, the global order is the same as for the customization files themselves: first the predefined themes of GPS, then the ones defined in customization files found through the `GPS_CUSTOM_PATH` directories, and finally the ones defined in files found in the user's own GPS directory:

```
<?xml version="1.0" ?>
<my-plugin-in>
  <theme name="my theme" description="Create a new menu">
    <menu action="my action"><title>/Edit/My Theme Menu</title></menu>
  </theme>
</my-plugin-in>
```

13.5.11 Defining new search patterns

The search dialog contains a number of predefined search patterns for Ada, C and C++. These are generally complex regular expressions, presented in the dialog with a more descriptive name. This includes for instance “Ada assignment”, which will match all such assignments.

You can define your own search patterns in the customization files. This is done through the `<vsearch-pattern>` tag. This tag can have a number of children tags:

<name> This tag is the string that is displayed in the search dialog to represent the new pattern. This is the text that the user will effectively see, instead of the often hard to understand regular expression.

<regexp> This tag provides the regular expression to use when the pattern has been selected by the user. Be careful that you must protect reserved XML characters such as ‘<’ and replace them by their equivalent expansion (“<” for this character).

This accepts one optional attribute, named *case-sensitive*. This attribute accepts one of two possible values (“true” or “false”) which indicates whether the search should distinguish lower case and upper case letters. Its default value is “false”.

<string> This tag provides a constant string that should be searched. Only one of *<regexp>* and *<string>* should be provided. If both exists, the first *<regexp>* child found is used. If there is none, the first *<string>* child is used.

The tag accepts the same optional attribute *case-sensitive* as above

Here is a small example on how the “Ada assignment” pattern was defined:

```
<?xml version="1.0" ?>
<search>
  <vsearch-pattern>
    <name>Ada: assignment</name>
    <regexp case-sensitive="false">\b(\w+)\s*:=</regexp>
  </vsearch-pattern>
</search>
```

13.5.12 Adding support for new languages

You can define new languages in a custom file by using the *Language* tag. Defining languages gives GPS the ability to highlight the syntax of a file, explore a file (using e.g. the project view), find files associated with a given language, ...

As described previously for menu items, any file in the `plug-ins` directory will be loaded by GPS at start up. Therefore, you can either define new languages in a separate file, or reuse a file where you already define actions and menus.

The following tags are available in a *Language* section:

Name A short string describing the name of the language.

Parent If set to the name of an existing language (e.g. *Ada*, *C++*) or another custom language, this language will inherit by default all its properties from this language. Any field explicitly defined for this language will override the inherited settings.

Spec_Suffix A string describing the suffix of spec/definition files for this language. If the language does not have the notion of spec or definition file, you can ignore this value, and consider using the *Extension* tag instead. This tag must be unique.

Body_Suffix A string describing the suffix of body/implementation files for this language. This tag works in coordination with the *Spec_Suffix*, so that the user can choose to easily go from one file to the other. This tag must be unique.

Extension A string describing one of the valid extensions for this language. There can be several such children. The extension must start with a ‘.’ character

Keywords A V7 style regular expression for recognizing and highlighting keywords. Multiple *Keywords* tags can be specified, and will be concatenated into a single regular expression. If the regular expression needs to match characters other than letters and underscore, you must also edit the *Wordchars* node. If a parent language has been specified for the current language definition it is possible to append to the parent *Keywords* by setting the *mode* attribute to *append*, the default value is *override* meaning that the keywords definition will replace the parent’s one.

The full grammar of the regular expression can be found in the spec of the file `g-regpat.ads` in the GNAT run time.

Wordchars Most languages have keywords that only contain letters, digits and underscore characters. However, if you want to also include other special characters (for instance ‘<’ and ‘>’ in XML), you need to use this tag to let GPS know. The value of this node is a string made of all the special word characters. You do not need to include letters, digits or underscores.

Engine The name of a dynamic library providing one or several of the functions described below.

The name can be a full pathname, or a short name. E.g. under most Unix systems if you specify *custom*, GPS will look for *libcusom.so* in the *LD_LIBRARY_PATH* run time search path. You can also specify explicitly e.g. *libcusom.so* or */usr/lib/libcusom.so*.

For each of the following five items, GPS will look for the corresponding symbol in *Engine* and if found, will call this symbol when needed. Otherwise, it will default to the static behavior, as defined by the other language-related items describing a language.

You will find the required specification for the C and Ada languages to implement the following functions in the directory `<prefix>/share/examples/gps/language` of your GPS installation. `language_custom.ads` is the Ada spec file; `language_custom.h` is the C spec file; `gpr_custom.ad?` are example files showing a possible Ada implementation of the function *Comment_Line* for the GPS project files (`.gpr` files), or any other Ada-like language; `gprcustom.c` is the C version of `gpr_custom.adb`.

Comment_Line Name of a symbol in the specified shared library corresponding to a function that will comment or uncomment a line (used to implement the menu *Edit->Un/Comment Lines*).

Parse_Constructs Name of a symbol in the specified shared library corresponding to a function that will parse constructs of a given buffer.

This procedure is used by GPS to implement several capabilities such as listing constructs in the project view, highlighting the current block of code, going to the next or previous procedure, ...

Format_Buffer Name of a symbol in the specified shared library corresponding to a function that will indent and format a given buffer.

This procedure is used to implement the auto indentation when hitting the `enter` key, or when using the format key on the current selection or the current line.

Parse_Entities Name of a symbol in the specified shared library corresponding to a function that will parse entities (e.g. comments, keywords, ...) of a given buffer. This procedure is used to highlight the syntax of a file, and overrides the *Context* node described below.

Context Describes the context used to highlight the syntax of a file.

Comment_Start A string defining the beginning of a multiple-line comment.

Comment_End A string defining the end of a multiple-line comment.

New_Line_Comment_Start A regular expression defining the beginning of a single line comment (ended at the next end of line). This regular expression may contain multiple possible line starts, such as `;/#` for comments starting after a semicolon or after the hash sign. If a parent language has been specified for the current language definition it is possible to append to the parent *New_Line_Comment_Start* by setting the *mode* attribute to *append*, the default value is *override* meaning that the *New_Line_Comment_Start* definition will replace the parent's one.

String_Delimiter A character defining the string delimiter.

Quote_Character A character defining the quote character, used for e.g. canceling the meaning of a string delimiter (`\` in C).

Constant_Character A character defining the beginning of a character literal.

Can_Indent A boolean indicating whether indentation should be enabled for this language. The indentation mechanism used will be the same for all languages: the number of spaces at the beginning of the current line is used when indenting the next line.

Syntax_Highlighting A boolean indicating whether the syntax should be highlighted/colorized.

Case_Sensitive A boolean indicating whether the language (and in particular the identifiers and keywords) is case sensitive.

Accurate_Xref A boolean indicating whether cross reference information for this language is supposed to be fully accurate or not (e.g. approximate, or none). Default to False.

Use_Semicolon A boolean indicating whether semicolons are expected in sources and may be used as a delimiter for syntax highlighting purposes. Default to False.

Categories Optional node to describe the categories supported by the project view for the current language. This node contains a list of *Category* nodes, each describing the characteristics of a given category, with the following nodes:

Name Name of the category, which can be either one of the following predefined categories: package, namespace, procedure, function, task, method, constructor, destructor, protected, entry, class, structure, union, type, subtype, variable, local_variable, representation_clause, with, use, include, loop_statement, case_statement, if_statement, select_statement, accept_statement, declare_block, simple_block, exception_handler, or any arbitrary name, which will create a new category.

Pattern Regular expression used to detect a language category. As for the *Keywords* node, multiple *Pattern* tags can be specified and will be concatenated into a single regular expression.

Index Index in the pattern used to extract the name of the entity contained in this category.

End_Index Optional attribute that indicates the index in the pattern used to start the next search. Default value is the end of the pattern.

Icon Name of a stock icon that should be used for that category (*Adding stock icons*). This attribute is currently ignored, and is reserved for future uses.

Project_Field This tag describes the tools that are used to support this language. The name of these tools is stored in the project files, and therefore only a limited number of tools can be specified. Note that this tag is currently only used by the project properties and wizard, and is not taken into account by other components.

This node has two attributes:

Name Name of the attribute in the project file. Currently, only “*compiler_command*” can be specified.

Index If present, this attribute indicates the index to use for the attribute in the project file. The line defining this attribute would therefore look like:

```
for Name ("Index") use "value";
```

e.g:

```
for Compiler_Command ("my_language") use "my_compiler";
```

The value of the index should be either the empty string or the name of the language.

The value of this tag is the string to use in the project properties editor when editing this project field.

Here is an example of a possible language definition for the GPS project files:

```
<?xml version="1.0"?>
<Custom>
  <Language>
    <Name>Project File</Name>
    <Spec_Suffix>.gpr</Spec_Suffix>
    <Keywords>^(case|e(nd|xte(nds|rnal))|for|is|</Keywords>
    <Keywords>limited|null|others|</Keywords>
    <Keywords>p(ackage|roject)|renames|type|use|w(hen|ith))\\b</Keywords>

  <Context>
    <New_Line_Comment_Start>--</New_Line_Comment_Start>
    <String_Delimiter>&quot;</String_Delimiter>
    <Constant_Character>&apos;</Constant_Character>
```

```
<Can_Indent>True</Can_Indent>
<Syntax_Highlighting>True</Syntax_Highlighting>
<Case_Sensitive>False</Case_Sensitive>
</Context>

<Categories>
  <Category>
    <Name>package</Name>
    <Pattern>^[ \t]*package[ \t]+((\w|\.)+)</Pattern>
    <Index>1</Index>
  </Category>
  <Category>
    <Name>type</Name>
    <Pattern>^[ \t]*type[ \t]+(\w+)</Pattern>
    <Index>1</Index>
  </Category>
</Categories>

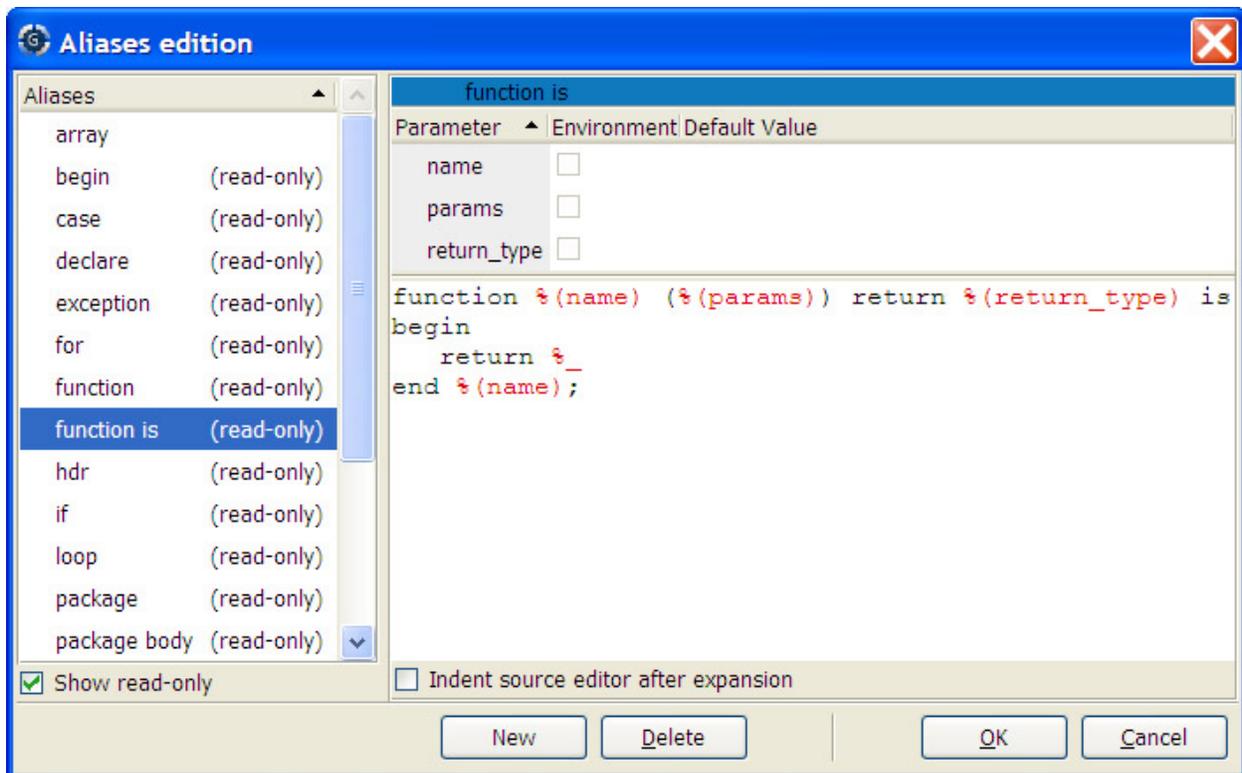
<Engine>gpr</Engine>
<Comment_Line>gpr_comment_line</Comment_Line>
</Language>
</Custom>
```

13.5.13 Defining text aliases

GPS provides a mechanism known as **aliases**. These are defined through the menu *Edit->Aliases*.

Each alias has a name, which is generally a short string of characters. When you type them in any textual entry in GPS (generally a source editor, but also entry fields for instance in the file selector), and then press the special activation key (by default `control-o`, controlled by a preference), this name is removed from the source editor, and replaced by the text you have associated with it.

Alias names may be composed of any character except newlines, but must start with a letter. GPS will jump to the start of each word before the current cursor position, and if the characters between this word start and the cursor position is an alias name (the comparison is case insensitive), this alias is expanded.



The alias editor is divided into three main parts: on the left side, the list of currently defined aliases is shown. Clicking on any of them will display the replacement text for this alias. If you click again the selected alias, GPS displays a text entry which you can use to rename an existing alias. Alias names must start with a letter. A check button at the bottom selects whether the read-only aliases (i.e. system-wide aliases) should be displayed.

The second part is the expansion text for the alias, at the bottom right corner. This replacement text can used multiple lines, and contain some special text that act as a special replacement. These special texts are highlighted in a different color. You can insert these special entities either by typing them, or by right-clicking in the editor, and select the entity in the contextual menu.

The following special entities are currently defined:

%_ This is the position where the cursor should be put once the replacement text has been inserted in the editor.

%(name) This is the name of a parameter. *name* can be any string you want, excluding closing parenthesis. See below for more information on parameters.

%D This is the current date, in ISO format. The year is displayed first, then the month and the day

%H This is the current time (hour, minutes and seconds)

%l If the expansion of the alias is done in a source editor, this is the line on which the cursor is when pressing `control-o`.

%c This is similar to **%l**, except it returns the current column.

%f If the expansion is done in a source editor, this is the name of the current file (its base name only, this doesn't include the directory)

%d If the expansion is done in a source editor, this is the directory in which the current file is

%p If the expansion is done in a source editor, this is the base name of the project file to which the file belongs.

%P If the expansion is done in a source editor, this is the full path name to the project file (directory and base name).

%O Used for recursive aliases expansion. This special character will expand the text seen before it in the current alias, after replacement of the parameters and possibly other recursive expansions. This is similar to pressing `control-o` (or any key you have defined for alias expansion) in the expanded form of the alias.

%% Inserts a percent sign as part of the expanded text

You cannot expand an alias recursively when already expanding that alias. For instance, if the alias expansion for *procedure* contains *procedure%O*, the inner procedure will not be expanded.

The indentation as set in the expansion of the alias is preserved when the alias is expanded. All the lines will be indented the same amount to the right as the alias name. You can override this default behavior by selecting the check button *Indent source editor after expansion*. In this case, GPS will replace the name of the alias by its expansion, and then automatically recompute the position of each line with its internal indentation engine, as if the text had been inserted manually.

The third part of the aliases editor, at the top right corner, lists the parameters for the currently selected alias. Any time you insert a *%(name)* string in the expansion text, GPS automatically detects there is a new parameter reference (or an old reference has changed name or was removed); the list of parameters is automatically updated to show the current list.

Each parameters has three attributes:

name This is the name you use in the expansion text of the alias in the *%(name)* special entity.

Environment This specifies whether the default value of the parameter comes from the list of environment variables set before GPS was started.

default value Instead of getting the default value from the environment variable, you can also specify a fixed text. Clicking on the initial value of the currently selected variable opens a text entry which you can use to edit this default value.

When an alias that contains parameters is expanded, GPS will first display a dialog to ask for the value of the parameters. You can interactively enter this value, which replaces all the *%(name)* entities in the expansion text.

13.5.14 Aliases files

The customization files described earlier can also contain aliases definition. This can be used for instance to create project or system wide aliases. All the customization files will be parsed to look for aliases definition.

All these customization files are considered as read-only by GPS, and therefore cannot be edited through the graphical interface. It is possible to override some of the aliases in your own custom files.

There is one specific files, which must contain only aliases definition. This is the file `$HOME/.gps/aliases`. Whenever you edit aliases graphically, or create new ones, they are stored in this file, which is the only one that GPS will ever modify automatically.

The system files are loaded first, and aliases defined there can be overridden by the user-defined file.

These files are standard XML customization files. The specific XML tag to use is `<alias>`, one per new alias. The following example contains a standalone customization file, but you might wish to merge the `<alias>` tag in any other customization file.

The following tags are available:

alias This indicates the start of a new alias. It has one mandatory attribute, *name*, which the text to type in the source editor before pressing `control-o`. It has one optional attribute, *indent*, which, if set to *true*, indicate that GPS should recompute the indentation of the newly inserted paragraph after the expansion.

param These are children of the *alias* node. There is one per parameter of the alias. They have one mandatory attribute, *name*, which is the name to type between *%(name)* in the alias expansion text.

They have one optional attribute, *environment*, which indicates the default value must be read from the environment variables if it is set to true.

These tags contain text, which is the default value for the parameter.

text This is a child of the *alias* node, whose value is the replacement text for the alias.

Here is an example of an alias file:

```
<?xml version="1.0"?>
<Aliases>
  <alias name="proc" >
    <param name="p" >Proc1</param>
    <param environment="true" name="env" />
    <text>procedure %(p) is
%(env) %_
end %(p); </text>
  </alias>
</Aliases>
```

13.5.15 Defining project attributes

The project files are required by GPS, and are used to store various pieces of information related to the current set of source files. This includes how to find the source files, how the files should be compiled, or manipulated through various tools,

However, the default set of attributes that are usable in a project file is limited to the attributes needed by the tool packaged with GPS or GNAT.

If you are delivering your own tools, you might want to store similar information in the project files themselves, since these are a very convenient place to associate some specific settings with a given set of source files.

GPS lets manipulate the contents of projects through XML customization files and script commands. You can therefore add you own typed attributes into the projects, so that they are saved automatically when the user saves the project, and reloaded automatically the next time GPS is started.

Declaring the new attributes

New project attributes can be declared in two ways: either using the advanced XML tags below, or using the `<tool>` tag (*Defining tool switches*).

The customization files support the `<project_attribute>` tag, which is used to declare all the new attributes that GPS should expect in a project. Attributes that have not been declared explicitly will not be accessible through the GPS scripting languages, and will generate warnings in the Messages window.

Project attributes are typed: they can either have a single value, or have a set of such values (a list). The values can in turn be a free-form string, a file name, a directory name, or a value extracted from a list of preset values.

Attributes that have been declared in these customization files will also be graphically editable through the project properties dialog, or the project wizard. Therefore, you should specify when an attribute is defined how it should be presented to the GPS user.

The `<project_attribute>` tag accepts the following attributes:

- *package* (a string, default value: “”)

This is the package in the project file in which the attribute is stored. Common practice suggests that one such package should be used for each tool. These packages provide namespaces, so that attributes with the same name, but for different tools, do not conflict with each other.

- *name* (a string, mandatory)

This is the name of the attribute. This should be a string with no space, and that represents a valid Ada identifier (typically, it should start with a letter and be followed by a set of letters, digits or underscore characters). This is an internal name that is used when saving the attribute in a project file.

- *editor_page* (a string, default value: “General”)

This is the name of the page in the Project Properties editor dialog in which the attribute is presented. If no such page already exists, a new one will be created as needed. If the page already exists, the attribute will be appended at its bottom.

- *editor_section* (a string, default value: “”)

This is the name of the section, inside editor page, in which the attribute is displayed. These sections are surrounded by frames, the title of which is given by the *editor_section* attribute. If this attribute is not specified, the attribute is put in an untitled section.

- *label* (a string, default value: the name of the attribute)

If this attribute is set to a value other than the empty string “”, a textual label is displayed to the left of the attribute in the graphical editor. This should be used to identify the attribute. However, it can be left to the empty string if the attribute is in a named section of its own, since the title of the section might be a good enough indication.

- *description* (a string, default value: “”)

This is the help message that describes the role of the attribute. It is displayed in a tooltip if the user leaves the mouse on top of the attribute for a while.

- *list* (a boolean, default value: “false”)

If this is set to “true”, the project attribute will in fact contains a list of values, as opposed to a single value. This is used for instance for the list of source directories in standard projects.

- *ordered* (a boolean, default value: “false”)

This is only relevant if the project attribute contains a list of values. This indicates whether the order of the values is relevant. In most cases, it will not matter. However, for instance, the order of source directories matters, since this also indicates where the source files will be searched, stopping at the first match.

- *omit_if_default* (a boolean, default value: “true”)

This indicates whether the project attribute should be set explicitly in the project if the user has left it to its default value. This can be used to keep the project files as simple as possible, if all the tools that will use this project attribute know about the default value. If this isn't the case, set *omit_if_default* to “false” to force the generation of the project attribute.

- *base_name_only* (a boolean, default value: “false”)

If the attribute contains a file name or a directory name, this indicates whether the full path should be stored, or only the base name. In most cases, the full path should be used. However, since GPS automatically looks for source files in the list of directories, for instance, the list of source files should only contain base names. This also increases the portability of project files.

- *case_sensitive_index* (a string (“true”, “false” or “file”), default: “false”)

This XML attribute is only relevant for project attributes that are indexed on another one (see below for more information on indexed attributes). It indicates whether two indexes that differ only by their casing should be considered the same. For instance, if the index is the name of one of the languages supported by GPS, the index is case insensitive since “Ada” is the same as “C”.

As a special case, the value “file” can be passed to indicate that the case sensitivity is the same as on the filesystem of the local host. This should be used when the index is the name of a file.

- *hide_in* (a string, default value: “”)

This XML attribute defines the various context in which this attribute should not be editable graphically. Currently, GPS provides three such contexts (“wizard”, “library_wizard” and “properties”, corresponding to the project creation wizards and the project properties editor). If any of those context is specified in *hide_in*, then the widget to edit this attribute will not be shown. The goal is to keep the graphical interface simple.

- *disable_if_not_set* (a boolean, default value: “false”)

If this attribute is set to “true”, the editor for this attribute will be greyed out if the attribute is not explicitly set in the project. In most cases, this is not needed, since the default value of the attribute can be used to leave the editor active at all time. However, when the value of the attribute is automatically computed depending on other attributes, the default value cannot be easily specified in the XML file, and in this case it might be easier to grey out the editor. An extra check box is displayed next to the attribute so that the user can choose to activate the editor and add the attribute to the project.

- *disable* (a space-separated list of attribute names, default: “”)

This is a list of attribute whose editor should be greyed out if the current attribute is specified. This only works if both the current attribute and the referenced attributes have their *disable_if_not_set* attribute set to “true”. This can be used to have mutually exclusive attributes present in the editor

Declaring the type of the new attributes

The type of the project attribute is specified through one or several child tags of *<project_attribute>*. The following tags are recognized.

- *<string>*

This tag indicates that the attribute is made of one (or more if it is a list) strings. This tag accepts the following XML attributes:

- *default* (a string, default value: “”)

This gives the default value to be used for the string (and therefore the project attribute), in case the user hasn’t overridden it.

If the attribute’s type is a file or a directory, the default value will be normalized (ie an absolute path will be generated from it, based on the project’s location, where “.” will represent the project’s directory). As a special case, if default is surrounded by parenthesis, no normalization takes place, so that you can later on test whether the user is still using the default value or not).

A special case if when *default* is set to “project source files”. In this case, this is automatically replaced by the known list of source files for the project. This doesn’t work from the project wizard, since the list of source files hasn’t been computed at that stage.

- *type* (one of “” (default), “file”, “directory” or “unit”)

This indicates what the string represents. In the first case, any value can be used. In the second case, it should represent a file name, although no check is done to make sure the file actually exists on the disk. But GPS will be able to do some special marshalling with the file name. The third case indicates that GPS should expect a directory. The fourth case indicates the GPS should expect the name of one of the project’s units.

- *filter* (one of “none”, “project”, “extending_project”)

This attribute is ignored for all types except “file”. In this case, it further specifies what kind of files can be used in this attribute. If the filter is “none”, then any file anywhere on the system is valid. If the filter is “project”, then only files from the selected project can be specified. If the filter is “extended_project”,

then only the files from the project extended by the current project can be specified. The attribute will not be shown if the current project is not an extending project.

- *allow_empty* (one of “True” or “False, default “True”)

This attribute indicates whether the value for this attribute can be an empty string. If not, the user must specify a value or an error message will be displayed in the project properties editor and project wizard.

- *<choice>*

This tag can be repeated several times. It indicates one of the valid values for the attribute, and can be used to provide a static list of such values. If it is combined with a *<string>* tag, this indicates that the attribute can be any string, although a set of possible values is provided to the user for ease of use. This tag accepts one optional attribute, “*default*”, which is a boolean. It indicates whether this value is the default to use for the project attribute.

If several *<choice>* tags are used, it is possible that several of them are part of the default value if the project attribute is a list, as opposed to a single value.

- *<shell>*

This tag is a GPS scripting command to execute to get a list of valid values for the attribute. The command should return a list. As for the *<choice>* tag, the *<shell>* tag can be combined with a *<string>* tag to indicate that the list of values returned by the scripting command is only a set of possible values, but that the project attribute can in fact take any value.

The *<shell>* tag accepts two attributes:

- *lang* (a string, default value: “shell”)

The scripting language in which the command is written. Currently, the only other possible value is “python”.

- *default* (a string, default value: “”)

The default value that the project attribute takes if the user hasn’t overridden it.

In some cases, the type of the project attribute, or at least its default value, depends on what the attribute applies to. The project file support this in the form of indexed project attribute. This is for instance used to specify what should be the name of the executable generated when compiling each of the main files in the project (ie the executable name for *gps.adb* should be *gps.exe*, the one for *main.c* should be *myapp.exe*, and so on).

Such attributes can also be declared through XML files. In such cases, the *<project_attribute>* tag should have one *<index>* child, and zero or more *<specialized_index>* children. Each of these two tags in turn take one of the already mentioned *<string>*, *<choice>* or *<shell>* tag.

The *<index>* tag indicates what other project attribute is used to index the current one. In the example given above for the executable names, the index is the attribute that contains the list of main files for the project.

It accepts the following XML attributes:

- *attribute* (a string, mandatory)

The name of the other attribute. This other attribute must be declared elsewhere in the customization files, and must be a list of values, not a single value.

- *package* (a string, default value: “”)

The package in which the index project attribute is defined. This is used to uniquely identify homonym attributes.

The *<specialized_index>* is used to override the default type of the attribute for specific values of the index. For instance, the project files contains an attribute that specify what the name of the compiler is for each language. It is indexed on the project attribute that list the languages used for the source files of the project. Its default value depends on the language (“gnatmake” for Ada, “gcc” for C, and so on). This attribute accepts requires one XML attribute:

- *value* (a string, mandatory)

This is the value of the attribute for which the type is overridden.

Note that almost all the standard project attributes are defined through an XML file, `projects.xml`, which is part of the GPS installation. Check this file to get advanced examples on how to declare project attributes.

Examples

The following example declares three attributes, with a single string as their value. This string represents a file or a directory in the last two cases. You can simply copy this into a `.xml` file in your `$HOME/.gps/plug-ins` directory, as usual:

```
<?xml version="1.0"?>
<custom>
  <project_attribute
    name="Single1"
    package="Test"
    editor_page="Tests single"
    editor_section="Single"
    description="Any string">

    <string default="Default value" />
  </project_attribute>

  <project_attribute
    name="File1"
    package="Test"
    editor_page="Tests single"
    editor_section="Single"
    description="Any file" >

    <string type="file" default="/my/file" />
  </project_attribute>

  <project_attribute
    name="Directory1"
    package="Test"
    editor_page="Tests single"
    editor_section="Single"
    description="Any directory" >

    <string type="directory" default="/my/directory/" />
  </project_attribute>
</custom>
```

The following example declares an attribute whose value is a string. However, a list of predefined possible values is also provided, as an help for interactive edition for the user. If the `<string>` tag wasn't given, the attribute's value would have to be one of the three possible choices:

```
<?xml version="1.0" ?>
<custom>
  <project_attribute
    name="Static2"
    package="Test"
    editor_page="Tests single"
    editor_section="Single"
    description="Choice from static list (or any string)" >
```

```
    <choice>Choice1</choice>
    <choice default="true" >Choice2</choice>
    <choice>Choice3</choice>
    <string />
  </project_attribute>
</custom>
```

The following example declares an attribute whose value is one of the languages currently supported by GPS. Since this list of languages is only known when GPS is executed, a script command is used to query this list:

```
<?xml version="1.0" ?>
<custom>
  <project_attribute
    name="Dynamic1"
    package="Test"
    editor_page="Tests single"
    editor_section="Single"
    description="Choice from dynamic list" >

    <shell default="C" >supported_languages</shell>
  </project_attribute>
</custom>
```

The following example declares an attribute whose value is a set of file names. The order of files in this list matters to the tools that are using this project attribute:

```
<?xml version="1.0" ?>
<custom>
  <project_attribute
    name="File_List1"
    package="Test"
    editor_page="Tests list"
    editor_section="Lists"
    list="true"
    ordered="true"
    description="List of any file" >

    <string type="file" default="Default file" />
  </project_attribute>
</custom>
```

The following example declares an attribute whose value is a set of predefined possible values. By default, two such values are selected, unless the user overrides this default setting:

```
<?xml version="1.0" ?>
<custom>
  <project_attribute
    name="Static_List1"
    package="Test"
    editor_page="Tests list"
    editor_section="Lists"
    list="true"
    description="Any set of values from a static list" >

    <choice>Choice1</choice>
    <choice default="true">Choice2</choice>
    <choice default="true">Choice3</choice>
  </project_attribute>
</custom>
```

The following example declares an attribute whose value is a string. However, the value is specific to each language (this could for instance be used for the name of the compiler to use for a given language). This is an indexed project attribute. It has two default values, one for Ada, one for C. All other languages have no default value:

```
<?xml version="1.0" ?>
<custom>
  <project_attribute
    name="Compiler_Name"
    package="Test"
    editor_page="Tests indexed"
    editor_section="Single"
    <index attribute="languages" package="">
      <string default="" />
    </index>
    <specialized_index value="Ada" >
      <string default="gnatmake" />
    </specialized_index>
    <specialized_index value="C" >
      <string default="gcc" />
    </specialized_index>
  </project_attribute>
</custom>
```

Accessing the project attributes

The new attributes that were defined are accessible from the GPS scripting languages, like all the standard attributes, *Querying project switches*.

You can for instance access the Compiler_Name attribute we created above with a python command similar to:

```
GPS.Project.root().get_attribute_as_string ("Compiler_Name", "Test", "Ada")
```

You can also access the list of main files for the project, for instance, by calling:

```
GPS.Project.root().get_attribute_as_list ("main")
```

13.5.16 Adding casing exceptions

A set of case exceptions can be declared in this file. Each case exception is put inside the tag `<word>` or `<substring>`. These exceptions are used by GPS to set identifiers or keywords case when editing case insensitive languages (except if corresponding case is set to Unchanged). *The Preferences Dialog*:

```
<?xml version="1.0" ?>
<exceptions>
  <case_exceptions>
    <word>GNAT</word>
    <word>OS_Lib</word>
    <substring>IO</substring>
  </case_exceptions>
</exceptions>
```

13.5.17 Adding documentation

New documentation can be added in GPS in various ways. This is useful if you want to point to your own project documentation for instance.

The first possibility is to create a new menu, through a `<menu>` tag in an XML file, associated with an action that either spawn an external web browser or calls the internal `GPS.Help.browse()` shell command.

However, this will not show the documentation in the `Help->Contents` menu, which you also might want to do.

To have both results, you should use the `<documentation_file>` tag in an XML file. These tags are generally found in a `gps_index.xml` files but you can in fact add them in any of your customization files.

The documentation files you display can contain the usual type of html links. In addition, GPS will treat specially links starting with `'%'`, and consider them as script commands to execute instead of file to display. The following example show how to insert a link that will in effect open a file in GPS when clicked by the user:

```
<a href="%shell:Editor.editor g-os_lib.ads">Open runtime file</a>
```

The first word after `'%'` is the name of the language, and the command to execute is found after the `'.'` character.

The `<documentation_file>` accepts a number of child nodes:

name This is the name of the file. It can be either an absolute file name, or a file name relative to one of the directories in `GPS_DOC_PATH`. If this child is omitted, you must specify a `<shell>` child.

This name can contain a reference to a specific anchor in the html file, using the standard HTML syntax:

```
<name>file#anchor</name>
```

shell This child specifies the name of a shell command to execute to get the name of the HTML file. This command can for instance create the HTML file dynamically, or download it locally using some special mechanism. This child accepts one attribute, `"lang"`, which is the name of the language in which the command is written

descr This is the description for this help file. It appears in a tool tip for the menu item.

category This is used in the `Help->Contents` menu to organize all the documentation files.

menu This is the full path to the menu. It behaves like a UNIX path, except it reference the various menus, starting from the menu bar itself. The first character of this path must be `"/"`. The last part of the path is the name of the new menu item. If not set, no menu is displayed for this file, although it will still appear in the `Help->Contents` menu

The `<menu>` child tag accepts two attributes.

before (optional, default="") The name of the menu before which the new entry should be inserted. If the new menu is inserted in some submenus, this tag controls the deeper nesting. Parent menus are created as needed, but if you wish to control their specific order, you should create them first with a `<menu>` tag.

after (optional, default="") The name of the menu after which the new entry should be inserted.

The following example shows how to create a new entry "item" in the Help menu, that will display `file.html`. The latter is searched in the `GPS_DOC_PATH` list of directories:

```
<?xml version="1.0"?>
<index>
  <documentation_file>
    <name>file.html</name>
    <descr>Tooltip text</descr>
    <category>name</category>
    <menu>/Help/item</menu>
  </documentation_file>
</index>
```

As mentioned above, HTML files are looked for through the `GPS_DOC_PATH` environment variable. However, you can also use the `<doc_path>` XML node to defined additional directories to be searched.

Such a directory is relative to the installation directory of GPS:

```
<?xml version="1.0"?>
<GPS>
  <doc_path>doc/application/</doc_path>
</GPS>
```

will add the directory `<prefix>/doc/application` to the search path for the documentation.

Such a directory can also be added through Python, as in:

```
GPS.HTML.add_doc_directory ('doc/application')
```

13.5.18 Adding stock icons

XML files can be used to define ‘stock icons’. Stock icons are pictures that are identified by their label, and which are used through GPS in various places, such as buttons, menus, toolbars, and so on.

The stock icons must be declared using the tag `<icon>`, within the global tag `<stock>`. The attribute `id` indicates the label used to identify the stock icon, and the attribute `file` points to the file which contains the actual picture, either in absolute format, or relative to the GPS icons directory. If you would like to read the icons from a directory relative to the location of the plug-in, you should use a python script like:

```
XML = r"""<GPS><stock>
  <icon id="gtk-new" file="${icons}/file.svg"/>
</stock></GPS>"""

icons = os.path.normpath(
    os.path.join(os.path.dirname(__file__), "../dir"))
XML = XML.replace("${icons}", icons)
GPS.parse_xml(XML)
```

If the stock icon is to be used in a toolbar, use the attribute `label` to specify the text to display in the toolbar, under the button, when the toolbar is configured to show text.

For icons that are intended to be displayed at multiple sizes, you can specify multiple files corresponding to these multiple sizes. This is done by adding children to the main icon node, with the tag `alternate`, containing a `file` attribute and a `size` attribute which correspond to the size for which this alternate source should be used.

Possible sizes are:

- 1 Menu item (ideal size: 16x16 pixels)
- 2 Button in a small toolbar (ideal size: 18x18 pixels)
- 3 Button in a large toolbar (ideal size: 24x24 pixels)
- 4 Image for a standard button (ideal size: 20x20 pixels)
- 5 Image used during drag-and-drop operation (ideal size: 32x32 pixels)
- 6 Main image in a dialog (ideal size: 48x48 pixels)

Here is an example:

```
<?xml version="1.0"?>
<my_visual_preferences>
  <stock>
    <icon id="myproject-my-picture" file="icons/my-picture.png" />

    <icon id="myproject-multipurpose-image"
      label="do something"
      file="icons/icon_default.png">
```

```
<alternate file="icons/icon_16.png" size="menu" />
<alternate file="icons/icon_24.png" size="large_toolbar" />
<alternate file="icons/icon_20.png" size="button" />
</icon>

</stock>
</my_visual_preferences>
```

Note: as shown in the example above, it is a good practice to prefix the label by a unique name (e.g. *myproject-*), in order to make sure that predefined stock icons will not get overridden by your icons.

13.5.19 Remote programming customization

The configuration of the remote programming functionality has two separate parts: the tools configuration (remote connection tools, shells, and rsync parameters) and the servers configuration.

The first part (see *Defining a remote connection tool*, *Defining a shell* and *Configuring rsync usage*) is handled by a pre-installed file in the plug-ins directory called `protocols.xml`.

The second part (see *Defining a remote server* and *Defining a remote path translation*), when configured via the user interface (see *Setup the remote servers*), will create a `remote.xml` file in the user's `gps` directory. System-wide servers can be also installed.

Defining a remote connection tool

Several remote access tools are already defined in GPS: `ssh`, `rsh`, `telnet` and `plink`. It is possible to add other tools, using the node *remote_connection_config*.

The attributes for this node are:

name (string) (mandatory) The name of the tool. This name does not necessarily correspond to the command used to launch the tool.

The following children are defined:

start_command (mandatory) The command used to launch the tool. This tag supports the *use_pipes* attribute. This attribute selects on Windows the way GPS will launch the remote tools, and can take the following values:

true use pipes to launch the tool.

false (default) use a tty emulation, which is a bit slower but allow password prompts retrieval with some tools.

Note that this argument has effects only on Windows platforms.

start_command_common_args (optional) The arguments that are provided to the tool. This string can contain the following replacement macros:

%C is replaced by the command executed on the remote host (e.g. the shell command)

%h is replaced by the remote host name

%U is replaced by the `start_command_user_args`, if a user is specified

%u is replaced by the user name

Note that if neither *%u* nor *%U* is found, and a user is specified in the remote connection configuration, then the `start_command_user_args` is placed at the beginning of the arguments.

start_command_user_args (optional) The arguments used to define a specific user during connection. *%u* is replaced by the user name

send_interrupt (optional) The characters sequence to send to the remote tool to interrupt the remote application. If unset, then an Interrupt signal is sent directly to the remote tool.

user_prompt_ptrn (optional) A regular expression, used to catch user name prompts from the connection tool. If undefined, a default regular expression is used.

password_prompt_ptrn (optional) A regular expression, used to catch password prompts from the connection tool. If undefined, a default regular expression is used.

passphrase_prompt_ptrn (optional) A regular expression, used to catch passphrase prompts from the connection tool. If undefined, a default regular expression is used.

extra_ptrn (optional) Complex child. Used to catch extra prompts from the connection tool, other than password, passphrase or username prompts. This tag has an attribute *auto_answer* telling if GPS automatically answers to this prompt, or ask the user. If *auto_answer* is *true*, then this tag needs an *answer* child, whose value is used for the answer. If *auto_answer* is *false*, then this tag needs a *question* child, whose value is used as question to the end user.

Defining a shell

Several shells are already defined in GPS: sh, bash, csh, tcsh and cmd.exe (Windows). It is possible to add other shells, using the node *remote_shell_config*.

The attributes for this node are:

name (string) (mandatory) The name of the shell. This name does not necessarily correspond to the command used to launch the shell.

The following children are defined:

start_command (mandatory) The command used to launch the shell. If arguments are required, they should be put here, separated with spaces.

generic_prompt (optional) The regular expression used to identify a prompt after the initial connection. If not set, a default value is used.

gps_prompt (mandatory) The regular expression used to identify a prompt after the initial setup is performed. If not set, a default value is used.

filesystem (mandatory) Takes the following values: *unix* or *windows*. This is the filesystem used by the shell.

init_commands (optional) Complex child. Each *cmd* child contains a command used to initialise a new session.

exit_commands (optional) Complex child. Each *cmd* child contains a command used to exit a session.

no_echo_command (optional) Command used to suppress the echo of the remote shell.

cd_command (mandatory) Command used to go to a directory. *%d* is replaced by the directory's full name.

get_status_command (mandatory) Command used to retrieve the status of the last command launched.

get_status_ptrn (mandatory) Regular expression used to retrieve the status returned by *get_status_command*. A pair of parenthesis is required, and identifies the status.

Configuring rsync usage

GPS has native support for the rsync tool, for paths synchronization during remote programming operations.

By default, GPS will use *-rsh=ssh* option if *ssh* is the main connection tool for the concerned server. It will also define the *-L* switch when transferring files to a Windows local host.

It is possible to define additional arguments to rsync using the *rsync_configuration* tag.

This tag accepts the child tagged *arguments*, and containing additional arguments to pass to rsync.

Defining a remote server

Remote servers can be defined via the user interface, as described in *Setup the remote servers*. This user interface will create a `remote.xml` file in the user's `gps` directory, which in turn can be installed in any plug-ins directory to set the values system-wide. This file will define for each server the node *remote_machine_descriptor*.

The attributes for this node are:

nickname (mandatory) Identifies uniquely the server in GPS.

network_name (mandatory) The server's network name or IP address.

remote_access (mandatory) The tool's name used to access the server. Shall point to one of the tools defined in *Defining a remote connection tool*.

remote_shell (mandatory) The shell's name used to access the server. Shall point to one of the shells defined in *Defining a shell*.

remote_sync (mandatory) The remote file synchronisation tool used to synchronize files between the local host and the server. Only *rsync* is recognized currently.

debug_console (optional) Can take the value *True* or *False*. Tells if a debug console should be displayed during connection with a remote host. False by default.

The children for this node are:

extra_init_commands (optional) Complex child. Can contain *cmd* children whose values are used to set server specific initialization commands.

max_nb_connections (optional) Positive number representing the maximum number of simultaneous connections GPS can launch.

timeout (optional) Positive number representing a timeout value (in ms) used for every action performed on the remote host.

Defining a remote path translation

Remote path translation can also be defined via the user interface, as described in *Setup the remote servers*. The remote paths translation are defined with the node *remote_path_config*.

The attributes for this node are:

server_name (mandatory) The server name concerned by the paths translation.

The *remote_path_config* node contains *mirror_path* children.

The attributes for the node *mirror_path* are:

local_path (mandatory) The absolute local path, expressed using the local filesystem standards.

remote_path (mandatory) The absolute remote path, expressed using the remote filesystem standards.

sync (mandatory) Specify the synchronization mechanism used for the paths (see *Paths settings*). Possible values are *NEVER*, *ONCE_TO_LOCAL*, *ONCE_TO_REMOTE* and *ALWAYS*.

13.5.20 Customizing build Targets and Models

The information displayed in *The Target Configuration Dialog* and in the Mode selection can be customized through XML.

Defining new Target Models

Models are defined in a *target-model* node which has one attributes, *name*, which contains the name of the model, and which supports the following sub-nodes:

<icon> The stock name of the icon to associate by default with targets of this model.

<description> A one-line description of what the Model supports

<server> Optional, defaulting to *Build_Server*. Indicates the server used for launching Targets of this model. *Remote operations*.

<is-run> Optional, defaulting to *False*. A boolean indicating whether this target corresponds to the launching of an executable rather than a build. Targets with such a model are launched through an interactive console in GPS, and their output is not parsed for errors.

<uses-shell> Optional, defaulting to *False*. A boolean indicating whether Targets of this model should be launched via the shell pointed to by the SHELL environment variable.

<command-line> Contains a number of *<arg>* nodes, each containing an argument of the default command line for this model, starting with the executable.

<switches command="executable_name"> The graphical description of the switches. (*Defining tool switches*):

```
<?xml version="1.0" ?>
<my_model>
  <target-model name="gprclean" category="">
    <description>Clean compilation artefacts with gprclean</description>
    <command-line>
      <arg>gprclean</arg>
      <arg>-P%PP</arg>
      <arg>%X</arg>
    </command-line>
    <icon>gps-clean</icon>
    <switches command="% (tool_name)s" columns="1">
      <check label="Clean recursively" switch="-r"
        tip="Clean all projects recursively" />
    </switches>
  </target-model>
</my_model>
```

Defining new Targets

Targets are defined in a *target* node which has three attributes:

name Contains the name of the Target. It must be a unique name. Underscores are interpreted as menu mnemonics. To represent an actual underscore, use a double underscore.

category The category which contains the Target, for purposes of ordering the tree in the Target Configuration Dialog, and for ordering in the Build menu. Underscores are interpreted as menu mnemonics. To represent an actual underscore, use a double underscore. If *category* begins and ends with an underscore, the menu for the Target is placed in the toplevel Build menu.

messages_category The name of the category to be used to organize messages in Locations window.

model The name of the Model of which this Target inherits initially.

<icon> The stock name of the icon to associate by default with the Target.

<in-toolbar> Optional, defaulting to *False*. A boolean indicating whether the Target should have an associated icon in the Toolbar.

<in-menu> Optional, defaulting to *True*. A boolean indicating whether the Target should have an associated entry in the Build menu.

<in-contextual-menus-for-projects> Optional, defaulting to *False*. A boolean indicating whether the Target should have an associated entry in the contextual menu for projects.

<in-contextual-menus-for-files> Optional, defaulting to *False*. A boolean indicating whether the Target should have an associated entry in the contextual menu for files.

<visible> Optional, defaulting to *True*. A boolean indicating whether the Target should initially be visible in GPS.

<read-only> Optional, defaulting to *False*. A boolean indicating whether the Target can be removed by the user.

<target-type> Optional, defaulting to an empty string. A string indicating whether the Target represents a simple target (if empty), or a family of Targets. The name represents a parameter passed to the *compute_build_targets* hook. If set to *main*, a new subtarget will be create for each Main source defined in the project.

<launch-mode> Optional, defaulting to *MANUALLY*. Indicates how the Target should be launched. Possible values are *MANUALLY*, *MANUALLY_WITH_DIALOG*, *MANUALLY_WITH_NO_DIALOG*, and *ON_FILE_SAVE*.

<server> Optional, defaulting to *Build_Server*. Indicates the server used for launching Target. *Remote operations*.

<command-line> Contains a number of **<arg>** nodes, each containing an argument of the default command line for this Target, starting with the executable:

```
<?xml version="1.0" ?>
<my_target>
  <target model="gprclean" category="C_clean" name="Clean _All">
    <in-toolbar>TRUE</in-toolbar>
    <icon>gps-clean</icon>
    <launch-mode>MANUALLY_WITH_DIALOG</launch-mode>
    <read-only>TRUE</read-only>
    <command-line>
      <arg>%gprclean</arg>
      <arg>-r</arg>
      <arg>%eL</arg>
      <arg>-P%PP</arg>
      <arg>%X</arg>
    </command-line>
  </target>
</my_target>
```

Defining new Modes

Modes are defined in a *builder-mode* node which has one attributes, *name*, which contains the name of the model, and which supports the following sub-nodes:

<description> A one-line description of what the Mode does

<subdir> Optional. The base name of the subdirectory to create for this Mode. The macro argument *%subdir* in the *extra-args* nodes will be substituted with this.

<supported-model> The name of a model supported by this Mode. There can be multiple *supported-model* nodes, each corresponding to a supported Model. Optionally, you can specify a *filter* attribute for this node, corresponding to the switches that are relevant for this mode. By default, all switches will be taken into account. The *extra-args* of the Mode that match *filter* will be passed to commands of the supported Models.

<extra-args> Contains a list of **<arg>** nodes, each containing one extra argument to append to the command line when launching Targets while this Mode is active. Macros are supported in the **<arg>** nodes:

```

<?xml version="1.0" ?>
<my_mode>
  <builder-mode name="optimization">
    <description>Build with code optimization activated</description>
    <subdir>optimized_objects</subdir>
    <supported-model>builder</supported-model>
    <supported-model>gnatmake</supported-model>
    <supported-model filter="--subdirs=">gprclean</supported-model>
    <extra-args>
      <arg>--subdirs=%subdir</arg>
      <arg>-cargs</arg>
      <arg>-O2</arg>
    </extra-args>
  </builder-mode>
</my_mode>

```

13.5.21 Toolchains customization

The list of toolchains and their values presented in the project editor (*The Project Wizard*) can be customized through XML. The GPS default list is contained in *toolchains.xml*. You can add your own toolchain by providing an xml description following the below described structure:

<toolchain_default> Contains the default names for the different tools used by all toolchains. The final name used will be *toolchain_name-default_name*.

<toolchain name="name"> Defines a toolchain using name “name”. This toolchain can override the default values defined in the *toolchain_default* above.

Each of the above tags can have the following children

<gnat_driver> Defines the gnat driver to use.

<gnat_list> Defines the gnat list tool to use.

<debugger> Defines the debugger to use.

<cpp_filt> Not used by GPS.

<compiler lang="lang"> Defines the compiler to use to compile language “lang”

The *toolchain_default* values can either be overridden or nullified by just providing the same tag with an empty value in a toolchain definition.

13.6 Adding support for new tools

GPS has built-in support for external tools. This feature can be used to support a wide variety of tools (in particular, to specify different compilers). Regular enhancements are done in this area, so if you are planning to use the external tool support in GPS, check for the latest GPS version available.

Typically, the following things need to be achieved to successfully use a tool:

- Specify its command line switches
- Pass it the appropriate arguments depending on the current context, or on user input
- Spawn the tool
- Optionally parse its result and act accordingly

Each of these points is discussed in further sections. In all these cases, most of the work can be done statically through XML customization files. These files have the same format as other XML customization files (*Customizing through XML and Python files*), and the tool descriptions are found in `<tool>` tags.

This tag accepts the following attributes:

name (mandatory) This is the name of the tool. This is purely descriptive, and will appear throughout the GPS interface whenever this tool is referenced. This includes for instances the tabs of the switches editor.

package (Default value is `ide`) This optional attribute specifies which package should be used in the project to store information about this tool, in particular its switches. Most of the time the default value should be used, unless you are working with one of the predefined packages.

See also *Defining project attributes*, for more information on defining your own project attributes. Using the “package”, “attribute” or “index” XML attributes of `<tool>` will implicitly create new project attributes as needed.

If this attribute is set to “`ide`”, then the switches cannot be set for a specific file, only at the project level. Support for file-specific switches currently requires modification of the GPS sources themselves.

attribute (Default value is `default_switches`) This optional attribute specifies the name of the attribute in the project which is used to store the switches for that tool.

index (Default value is the tool name) This optional attribute specifies what index is used in the project. This is mostly for internal use by GPS, and describes what index of the project attribute is used to store the switches for that tool.

override (Default value is ‘false’) This optional attribute specifies whether the tool definition can be redefined. The accepted values are ‘true’ or ‘false’. If `override` is not set, and the tool is defined several times, then a Warning will be displayed.

This tag accepts the following children, described in separate sections:

`<switches>` (*Defining tool switches*)

`<language>` (*Defining supported languages*)

`<initial-cmd-line>` (*Defining default command line*)

13.6.1 Defining supported languages

This is the language to which the tool applies. There can be from no to any number of such nodes for one `<tool>` tag.

If no language is specified, the tool applies to all languages. In particular, the switches editor page will be displayed for all languages, no matter what languages they support.

If at least one language is specified, the switches editor page will only be displayed if that language is supported by the project:

```
<?xml version="1.0" ?>
<my_tool>
  <tool name="My Tool" >
    <language>Ada</language>
    <language>C</language>
  </tool>
</my_tool>
```

13.6.2 Defining default command line

It is possible to define the command line that should be used for a tool when the user is using the default project, or hasn't overridden this command line in the project.

This is done through the `<initial-cmd-line>` tag, as a child of the `<tool>` tag. Its value is the command line that would be passed to the tool. This command line is parsed as usual, e.g. quotes are taken into account to avoid splitting switches each time a space is encountered:

```
<?xml version="1.0" ?>
<my_tool>
  <tool name="My tool" >
    <initial-cmd-line>-a -b -c</initial-cmd-line>
  </tool>
</my_tool>
```

13.6.3 Defining tool switches

The user has to be able to specify which switches to use with the tool. If the tool is simply called through custom menus, you might want to hard code some or all of the switches. However, in the general case it is better to use the project properties editor, so that project-specific switches can be specified.

This is what GPS does by default for Ada, C and C++. You can find in the GPS installation directory how the switches for these languages are defined in an XML file. These provide extended examples of the use of customization files.

The switches editor in the project properties editor provides a powerful interface to the command line, where the user can edit the command line both as text and through GUI widgets.

The switches are declared through the `<switches>` tag in the customization file, which must be a child of a `<tool>` tag as described above.

This `<switches>` tag accepts the following attributes:

lines (default value is 1) The switches in the project properties editor are organized into boxes, each surrounded by a frame, optionally with a title. This attribute specifies the number of rows of such frames.

columns (default value is 1) This attribute specifies the number of columns of frames in the project properties page.

separator (default value is "") This attribute specifies the default character that should go between a switch and its value, to distinguish cases like "-a 1", "-a1" and "-a=1". This can be overridden separately for each switch. Note that if you want the separator to be a space, you must use the value " " rather than " ", since XML parser must normalize the latter to the empty string when reading the XML file.

use_scrolled_window (Default value is false) This optional attribute specifies if the boxes of the project editor are placed into scrolled window. This is particularly useful if the number of displayed switches is important.

show_command_line (Default value is true) If this attribute is set to "false", the command line will not be displayed in the project properties editor. This can be used for instance if you only want users to edit it through the buttons and other widgets, and not directly.

switch_char (Default value is "-") This is the leading character of command line arguments that indicate they are considered as switches. Arguments not starting with this character will be kept as is, and cannot have graphical widgets associated with them

sections (Default value is empty) This is a space separated list of switches delimiting a section (such as "-bargs -cargs -largs"). A section of switches is a set of switches that need to be grouped together and preceded by a specific switch. Sections are always placed at the end of the command line, after regular switches.

This `<switches>` tag can have any number of child tag, among the following. They can be repeated multiple times if you need several check boxes. For consistency, most of these child tags accept attributes among the following:

line (*default value is 1*) This indicates the row of the frame that should contain the switch. See the description of *lines* above.

column (*default value is 1*) This indicates the column of the frame that should contain the switch. See the description of *columns* above.

label (*mandatory*) This is the label which is displayed in the graphical interface

switch (*mandatory*) This is the text that should be put on the command line if that switch is selected. Depending on its type, a variant of the text might be put instead, see the description of *combo* and *spin* below. This switch shouldn't contain any space.

switch-off (*default value is empty*) This attribute is used for `<check>` tags, and indicates the switch used for deactivating the concerned feature. This is useful for features that are on by default on certain occasions, but can be individually deactivated.

section (*default value is empty*) This is the switch section delimiter (such as “-cargs”). See the ‘sections’ attribute of the tag ‘switches’ for more information.

tip (*default value is empty*) This is the tooltip which describes that switch more extensively. It is displayed in a small popup window if the user leaves the mouse on top of the widget. Note that tags accepting the tip attribute also accept a single child `<tip>` whose value will contain the text to be displayed. The advantage of the latter is that the text formatting is then kept.

before (*default value is “false”*) This attribute is used to indicate that a switch needs to be always inserted at the beginning of the command line.

min (*default value is 1*) This attribute is used for `<spin>` tags, and indicates the minimum value authorized for that switch.

max (*default value is 1*) This attribute is used for `<spin>` tags, and indicates the maximum value authorized for that switch.

default (*default value is 1*) This attribute is used for `<check>` and `<spin>` tags. See the description below.

noswitch (*default is empty*) This attribute is only valid for `<combo>` tags, and described below.

nodigit (*default is empty*) This attribute is only valid for `<combo>` tags, and described below.

value (*mandatory*) This attribute is only valid for `<combo-entry>` tags.

separator (*default is the value given to ‘<switches>’*) This attribute specifies the separator to use between the switch and its value. See the description of this attribute for `<switches>`.

Here are the valid children for `<switches>`:

<title> This tag, which accepts the *line* and *column* attributes, is used to give a name to a specific frame. The value of the tag is the title itself. You do not have to specify a name, and this can be left to an empty value.

Extra attributes for `<title>` are:

line-span (*default value is 1*) This indicates how many rows the frame should span. If this is set to 0, then the frame is hidden from the user. See for instance the Ada or C switches editor.

column-span (*default value is 1*) This indicates how many columns the frame should span. If this is set to 0, then the frame is hidden from the user. See for instance the Ada or C switches editor.

<check> This tag accepts the *line*, *column*, *label*, *switch*, *switch-off*, *section*, *default*, *before* and *tip* attributes.

This tag doesn't have any value. An optional `<tip>` child can be present.

It creates a toggle button. When the latter is active, the text defined in the switch attribute is added as is to the command line. The switch can be also activated by default (*default* attribute is “on” or “true”). In this case, deactivating the switch will add *switch-off* to the command line.

<spin> This tag accepts the *line*, *column*, *label*, *switch*, *section*, *tip*, *min*, *max*, *separator* and *default* attributes.

This tag doesn't have any value. An optional **<tip>** child can be present.

This switch will add the contents of the *switch* attribute followed by the current numeric value of the widget to the command line. This is typically used to indicate indentation length for instance. If the current value of the widget is equal to the *default* attribute, then nothing is added to the command line.

<radio> This tag accepts the *line* and *column* attributes. It groups any number of children, each of which is associated with its own switch. However, only one of the children can be selected at any given time.

The children must have the tag *radio-entry*. This tag accepts the attributes *label*, *switch*, *section*, *before* and *tip*. As a special case, the switch attribute can have an empty value ("") to indicate this is the default switch to use in this group of radio buttons.

This tag doesn't have any value. An optional **<tip>** child can also be present.

<field> This tag accepts the *line*, *column*, *label*, *switch*, *section*, *separator*, *before* and *tip* attributes.

This tag doesn't have any value. An optional **<tip>** child can be present.

This tag describes a text edition field, which can contain any text the user types. This text will be prefixed by the value of the *switch* attribute, and the separator (by default nothing). If no text is entered in the field by the user, nothing is put on the command line.

This tag accepts two extra attributes:

as-directory (optional) If this attribute is specified and set to "true", then an extra "Browse" button is displayed, so that the user can easily select a directory.

as-file (optional) This attribute is similar to *as-directory*, but opens a dialog to select a file instead of a directory. If both attributes are set to "true", the user will select a file.

<combo> This tag accepts the *line*, *column*, *label*, *switch*, *section*, *before*, *tip*, *noswitch*, *separator* and *nodigit* attributes.

The tag **<combo>** accepts any number of *combo-entry* children tags, each of which accepts the *label* and *value* attribute. An optional **<tip>** child can also be present.

The text inserted in the command line is the text from the *switch* attribute, concatenated with the text of the *value* attribute for the currently selected entry. If the value of the current entry is the same as that of the *nodigit* attribute, then only the text of the *switch* attribute is put on the command line. This is in fact necessary to interpret the gcc switch "-O" as "-O1".

If the value of the current entry is that of the *noswitch* attribute, then nothing is put in the command line.

<popup> This tag accepts the *line*, *column*, *label*, *lines* and *columns* attributes. This displays a simply button that, when clicked, displays a dialog with some extra switches. This dialog, just as the switches editor itself, is organized into lines and columns of frames, the number of which is provided by the *lines* and *columns* attributes.

This tag accepts any number of children, which are the same as the **<switches>** attribute itself.

<dependency> This tag is used to describe a relationship between two switches. It is used for instance when the "Debug Information" switch is selected for "Make", which forces it for the Ada compiler as well.

It has its own set of attributes:

master-page master-switch master-section These two attributes define the switch that possibly forces a specific setting on the slave switch. In our example, they would have the values "Make" and "-g". The switch referenced by these attributes must be of type **<check>** or **<field>**. If it is part of a section, then 'master-section' needs to be defined. If the check button is selected, it forces the selection of the slave check button. Likewise, if the field is set to any value, it forces the selection of the slave.

slave-page slave-switch slave-section These two attributes define the switch which is acted upon by the master switch. In our example, they would have the values “Ada” and “-g”. The switch referenced by these attributes must be of type `<check>`.

master-status slave-status These two switches indicate which state of the master switch forces which state of the slave-status. In our example, they would have the values “on” and “on”, so that when the make debug information is activated, the compiler debug information is also activated. However, if the make debug information is not activated, no specific setup is forced for the compiler debug information. If master-status is “off” and the master switch is a field, then the status of the slave will be changed when no value is set in the field.

`<default-value-dependency>` This tag is used to describe a relationship between two switches. It is slightly different from the `<dependency>` tag in that the relationship concerns only the default activation states. It is used for instance when the “-gnatwa” switch is selected for the “Ada” Compiler, which imply that the default values for “-gnatwc”, “-gnatwd”, etc. become activated by default. They can however still be deactivated with respectively “-gnatwC” and “-gnatwD”.

It has its own set of attributes:

master-switch This is the switch that triggers the dependency. If *master-switch* is present in the command line, then the switch’s default status of *slave-switch* is modified accordingly.

slave-switch This is the switch whose default value depends on *master-switch*. This needs to be a switch already defined in a `<switch>` tag. It can match its ‘switch’ or ‘switch-off’ attributes. In the latter case, the slave-switch default value is deactivated if master-switch is present.

`<expansion>` This tag is used to describe how switches can be grouped together on the command line to keep it shorter. It is also used to define aliases between switches.

It is easier to explain it through an example. Specifying the GNAT switch “-gnaty” is equivalent to specifying “-gnaty3abcefhiklmnrst”. This is in fact a style check switch, with a number of default values. But it is also equivalent to decomposing it into several switches, as in “-gnatya”, “-gnatyb”, ...; With this information, GPS will try to keep the command line length as short as possible, to keep it readable.

Both these aspects are defined in a unique `<expansion>` tag, which accepts two attributes: *switch* is mandatory, and *alias* is optional. *Alias* contains the text “-gnatyabcefhiklmnrst” in our example.

There are two possible uses for this tag:

- If the “alias” attribute is not specified, then the “switch” attribute indicates that all switches starting with that prefix should be grouped. For instance, if you pass “-gnatw” as the value for the “switch” attribute, then a command line with “-gnatwa -gnatw.b” will in fact result in “-gnatwa.b”.
- If the “alias” attribute is specified, then the “switch” attribute is considered as a shorter way of writing “alias”. For instance, if “switch” is “-gnaty” and “alias” is “-gnaty3abcefhiklmnrst”, then the user can simply type “-gnaty” to mean the whole set of options.

The same “switch” attribute can be used in two expansion nodes if you want to combine the behavior.

For historical reasons, this tag accepts `<entry>` children, but these are no longer used.

13.6.4 Executing external tools

The user has now specified the default switches he wants to use for the external tool. Spawning the external tool can be done either from a menu item, or as a result of a key press.

Both cases are described in an XML customization file, as described previously, and both are setup to execute what GPS calls an action, i.e. a set of commands defined by the `<action>` tag.

Chaining commands

This action tag, as described previously, executes one or more commands, which can either be internal GPS commands (written in any of the scripting language supported by GPS), or external commands provided by executables found on the PATH.

The command line for each of these commands can either be hard-coded in the customization file, or be the result of previous commands executed as part of the same action. As GPS executes each command from the action in turn, it saves its output on a stack as needed. If a command line contains a special construct `%1`, `%2`... then these constructs will be replaced by the result of respectively the last command executed, the previous from last command, and so on. They are replaced by the returned value of the command, not by any output it might have done to some of the consoles in GPS.

Every time you execute a new command, it pushes the previous `%1`, `%2`... parameters one step further on the stack, so that they become respectively `%2`, `%3`... and the output of that command becomes `%1`.

The result value of the previous commands is substituted exactly as is. However, if the output is surrounded by quotes, they are ignored when a substitution takes place, so you need to put them back if they are needed. The reason for this behavior is so that for scripting languages that systematically protect their output with quotes (simple or double), these quotes are sometimes in the way when calling external commands:

```
<?xml version="1.0" ?>
<quotes>
  <action name="test quotes">
    <shell lang="python">' -a -b -c'</shell>
    <external> echo with quotes: "%1"</external>
    <external> echo without quotes: %2</external/>
  </action>
</quotes>
```

If one of the commands in the action raises an error, the execution of the action is stopped immediately, and no further command is performed.

Saving open windows

Before launching the external tool, you might want to force GPS to save all open files, the project...; This is done using the same command GPS itself uses before starting a compilation. This command is called `MDI.save_all`, and takes one optional boolean argument which specifies whether an interactive dialog should be displayed for the user.

Since this command aborts when the user presses cancel, you can simply put it in its own `<shell>` command, as in:

```
<?xml version="1.0" ?>
<save_children>
  <action name="test save children">
    <shell>MDI.save_all 0</shell>
    <external>echo Run unless Cancel was pressed</external>
  </action>
</save_children>
```

Querying project switches

Some GPS shell commands can be used to query the default switches set by the user in the project file. These are `get_tool_switches_as_string`, `get_tool_switches_as_list`, or, more generally, `get_attribute_as_string` and `get_attribute_as_list`. The first two require a unique parameter which is the name of the tool as specified in the `<tool>` tag. This name is case-sensitive. The last two commands are more general and can be used to query the status of any attribute from the project. See their description by typing the following in the GPS shell console window:

```
help Project.get_attribute_as_string
help Project.get_attribute_as_list
```

The following is a short example on how to query the switches for the tool “Find” from the project, *Tool example*. It first creates an object representing the current project, then passes this object as the first argument of the `get_tool_switches_as_string` command. The last external command is a simple output of these switches:

```
<?xml version="1.0" ?>
<find_switches>
  <action name="Get switches for Find">
    <shell>Project %p</shell>
    <shell>Project.get_tool_switches_as_string %1 Find </shell>
    <external>echo %1</external>
  </action>
</find_switches>
```

The following example shows how something similar can be done from Python, in a simpler manner. For a change, this function queries the Ada compiler switches for the current project, and prints them out in the messages window. The:

```
<?xml version="1.0" ?>
<query_switches>
  <action name="Query compiler switches">
    <shell lang="python">GPS.Project ("%p").get_attribute_as_list
      (package="compiler",
       attribute="default_switches",
       index="ada")</shell>
    <external>echo compiler switches= %1</external>
  </action>
</query_switches>
```

Querying switches interactively

Another solution to query the arguments for the tool is to ask the user interactively. The scripting languages provides a number of solutions for these.

They generally have their own native way to read input, possibly by creating a dialog.

In addition, the simplest solution is to use the predefined GPS commands for this. These are the two functions:

yes_no_dialog This function takes a single argument, which is a question to display. Two buttons are then available to the user, “Yes” and “No”. The result of this function is the button the user has selected, as a boolean value.

input_dialog This function is more general. It takes a minimum of two arguments, with no upper limit. The first argument is a message describing what input is expected from the user. The second, third and following arguments each correspond to an entry line in the dialog, to query one specific value (as a string). The result of this function is a list of strings, each corresponding to these arguments.

From the GPS shell, it is only convenient to query one value at a time, since it doesn’t have support for lists, and would return a concatenation of the values. However, this function is especially useful with other scripting languages.

The following is a short example that queries the name of a directory and a file name, and displays each in the Messages window:

```
<?xml version="1.0" ?>
<query_file>
  <action name="query file and dir">
    <shell lang="python">list=GPS.MDI.input_dialog \\  

```

```

    ("Please enter directory and file name", "Directory", "File")</shell>
    <shell lang="python">print ("Dir=" + list[0], "File=" + list[1])</shell>
  </shell>
</action>
</query_file>

```

Redirecting the command output

The output of external commands is sent by default to the GPS console window. In addition, finer control can be exercised using the *output* attribute of the `<external>` and `<shell>` tags.

This attribute is a string that may take any value. Two values have specific meanings:

“*none*” The output of the command, as well as the text of the command itself, will not be shown to the user at all.

“” The output of the command is sent to the GPS console window, entitled “Messages”.

other values A new window is created, with the title given by the attribute. If such a window already exists, it is cleared up before any of the command in the chain is executed. The output of the command, as well as the text of the command itself, are sent to this new window.

This attribute can also be specified at the `<action>` tag level, in which case it defines the default value for all `<shell>` and `<external>` tags underneath. If it isn’t specified for the action itself, its default value will always be the empty string, i.e. output is sent to the GPS console:

```

<?xml version="1.0" ?>
<ls>
  <action name="ls current directory" output="default output" >
    <shell output="Current directory" >pwd</shell>
    <external output="Current directory contents" >/bin/ls</external>
  </action>
</ls>

```

Processing the tool output

The output of the tool has now either been hidden or made visible to the user in one or more windows.

There are several additional things that can be done with this output, for further integration of the tool in GPS.

- Parsing error messages .. `index::Locations.parse`

External tools can usually display error messages for the user that are associated with specific files and locations in these files. This is for instance the way the GPS builder itself analyzes the output of *make*.

This can be done for your own tools using the shell command *Locations.parse*. This command takes several arguments, so that you can specify your own regular expression to find the file name, line number and so on in the error message. By default, it is configured to work seamlessly with error message of the forms:

```

file:line: message
file:line:column: message

```

Please refer to the online help for this command to get more information (by e.g. typing *help Locations.parse* in the GPS Shell).

Here is a small example on how to run a *make* command and send the errors to the location window afterward.

For languages that support it, it is also recommended that you quote the argument with triple quotes, so that any special character (newlines, quotes, ...) in the output of the tool are not specially interpreted by GPS. Note also that you should leave a space at the end, in case the output itself ends with a quote:

```
<?xml version="1.0" ?>
<make>
  <action name="make example" >
    <external>make</external>
    <on-failure>
      <shell>Locations.parse ""%1 "" make_example</shell>
    </on-failure>
  </action>
</make>
```

- Auto-correcting errors .. index:: Codefix.parse

GPS has support for automatically correcting errors for some of the languages. You can get access to this auto-fixing feature through the *Codefix.parse* shell command, which takes the same arguments as for *Locations.parse*.

This will automatically add pixmaps to the relevant entries in the location window, and therefore *Locations.parse* should be called first prior to calling this command.

Errors can also be fixed automatically by calling the methods of the *Codefix* class. Several codefix sessions can be active at the same time, each of which is associated with a specific category. The list of currently active sessions can be retrieved through the *Codefix.sessions()* command.

If support for python is enabled, you can also manipulate the fixable errors for a given session. To do so, you must first get a handle on that session, as shown in the example below. You can then get the list of fixable errors through the *errors* command.

Each error is of the class *CodefixError*, which has one important method *fix* which allows you to perform an automatic fixing for that error. The list of possible fixes is retrieved through *possible_fixes*:

```
print GPS.Codefix.sessions ()
session = GPS.Codefix ("category")
errors = session.errors ()
print errors [0].possible_fixes ()
errors [0].fix ()
```

13.7 Customization examples

13.7.1 Menu example

This section provides a full example of a customization file. It creates a top-level menu named *custom menu*. This menu contains a menu item named *item 1*, which is associated to the external command *external-command 1*, a sub menu named *other menu*, etc...:

```
<?xml version="1.0"?>
<menu-example>
  <action name="action1">
    <external>external-command 1</external>
  </action>

  <action name="action2">
    <shell>edit %f</shell>
  </action>

  <submenu>
    <title>custom menu</title>
    <menu action="action1">
      <title>item 1</title>
```

```

</menu>

<submenu>
  <title>other menu</title>
  <menu action="action2">
    <title>item 2</title>
  </menu>
</submenu>
</submenu>
</menu-example>

```

13.7.2 Tool example

This section provides an example that defines a new tool. This is only a short example, since Ada, C and C++ support themselves are provided through such a file, available in the GPS installation.

This example adds support for the “find” Unix utility, with a few switches. All these switches are editable through the project properties editor.

It also adds a new action and menu. The action associated with this menu gets the default switches from the currently selected project, and then ask the user interactively for the name of the file to search:

```

<?xml version="1.0" ?>
<toolexample>
  <tool name="Find" >
    <switches columns="2" >
      <title column="1" >Filters</title>
      <title column="2" >Actions</title>

      <spin label="Modified less than n days ago" switch="-mtime-"
        min="0" max="365" default="0" />
      <check label="Follow symbolic links" switch="-follow" />

      <check label="Print matching files" switch="-print" column="2" />
    </switches>
  </tool>

  <action name="action find">
    <shell>Project %p</shell>
    <shell>Project.get_tool_switches_as_string %1 Find </shell>
    <shell>MDI.input_dialog "Name of file to search" Filename</shell>
    <external>find . -name %1 %2</external>
  </action>

  <Submenu>
    <Title>External</Title>
    <menu action="action find">
      <Title>Launch find</Title>
    </menu>
  </Submenu>
</toolexample>

```

13.8 Scripting GPS

13.8.1 Scripts

Scripts are small programs that interact with GPS and allow you to perform complex tasks repetitively and easily. GPS includes support for two scripting languages currently, although additional languages might be added in the future. These two languages are described in the following section.

Support for scripting is currently work in progress in GPS. As a result, not many commands are currently exported by GPS, although their number is increasing daily. These commands are similar to what is available to people who extend GPS directly in Ada, but with a strong advantage: they do not require any recompilation of the GPS core, and can be tested and executed interactively.

The goal of such scripts is to be able to help automate processes such as builds, automatic generation of graphs, ...

These languages all have a separate console associated with them, which you can open from the *Tools* menu. In each of these console, GPS will display a prompt, at which you can type interactive commands. These console provide completion of the command names through the `tab` key.

For instance, in the GPS shell console you can start typing:

```
GPS> File
```

then press the `tab` key, which will list all the functions whose name starts with “File”.

A similar feature is available in the python console, which also provides completion for all the standard python commands and modules.

All the scripting languages share the same set of commands exported by GPS, thanks to an abstract interface defined in the GPS core. As a result, GPS modules do not have to be modified when new scripting languages are added.

Scripts can be executed immediately upon startup of GPS by using the command line switch `-load`. Specifying the following command line:

```
gps --load=shell:mytest.gps
```

will force the `gps` script `mytest.gps` to be executed immediately, before GPS starts reacting to user’s requests. This is useful if you want to do some special initializations of the environment. It can also be used as a command line interface to GPS, if you script’s last command is to exit GPS.

In-line commands can also be given directly on the command line through `-eval` command line switch.

For instance, if you want to analyze an entity in the entity browser from the command line, you would pass the following command switches:

```
gps --eval=shell:'Entity entity_name file_name; Entity.show %1'
```

See the section *Customizing through XML and Python files* on how to bind key shortcuts to shell commands.

13.8.2 Scripts and GPS actions

There is a strong relationship between GPS actions, as defined in the customization files (*Defining Actions*), and scripting languages

Actions can be bound to menus and keys through the customization files or the *Edit->Key shortcuts* dialog.

These actions can execute any script command, *Defining Actions*. This is done through the `<shell>` XML tag.

But the opposite is also true. From a script, you can execute any action registered in GPS. This can for instance be used to split windows, highlight lines in the editor, ... when no equivalent shell function exists. This can also be used to execute external commands, if the scripting language doesn't support this in an easy manner.

Such calls are made through a call to *execute_action*, as in the following example:

```
execute_action "Split horizontally"

GPS.execute_action (action="Split horizontally")
```

The list of actions known to GPS can be found through the *Edit->Key shortcuts* dialog. Action names are case sensitive.

Some of the shell commands take subprograms as parameters. If you are using the GPS shell, this means you have to pass the name of a GPS action. If you are using Python, this means that you pass a subprogram, *Subprogram parameters*.

13.8.3 The GPS Shell

The GPS shell is a very simple-minded, line-oriented language. It is accessible through the *Shell* window at the bottom of the GPS window. It is similar to a Unix shell, or a command window on Windows systems.

Type *help* at the prompt to get the list of available commands, or *help* followed by the name of a command to get more information on that specific command.

The following example shows how to get some information on a source entity, and find all references to this entity in the application. It searches for the entity "entity_name", which has at least one reference anywhere in the file "file_name.adb". After the first command, GPS returns an identifier for this entity, which can be used for all commands that need an entity as a parameter, as is the case for the second command. When run, the second command will automatically display all matching references in the location window:

```
GPS> Entity my_entity file_name.adb
<Entity_0x09055790>
GPS> Entity.find_all_refs <Entity_0x09055790>
```

Since the GPS shell is very simple, it doesn't provide any reference counting for the result types. As a result, all the values returned by a command, such as *<Entity_0x09055790>* in the example above, are kept in memory.

The GPS shell provides the command *clear_cache* which removes all such values from the memory. After this command is run, you can no longer use references obtained from previous commands, although of course you can run these commands again to get a new reference.

The return value of the 9 previous commands can easily be recalled by passing *%1*, *%2*, ... on the command line. For instance, the previous example could be rewritten as:

```
GPS> Entity my_entity file_name.adb
<Entity_0x09055790>
GPS> Entity.find_all_refs %1
```

These return values will be modified also for internal commands sent by GPS, so you should really only use this when you emit multiple commands at the same time, and don't do any other action in GPS. This is mostly useful when used for command-line scripts (see *-eval* and *-load*), or for custom files, *Customizing through XML and Python files*.

Arguments to commands can, but need not, be quoted. If they don't contain any space, double-quote ("") or newline characters, you do not need to quote them. Otherwise, you should surround them with double-quotes, and protect any double-quote part of the argument by preceding it with a backslash.

There is another way to quote a command: use three double-quotes characters in a row. Any character loses its special meaning until the next three double-quotes characters set. This is useful if you do not know in advance the contents of the string you are quoting:

```
Locations.parse "%1" category_name
```

13.8.4 The Python Interpreter

Python is an interpreted object-oriented language, created by Guido Van Rossum. It is similar in its capabilities to languages such as Perl, Tcl or Lisp. This section is not a tutorial on python programming. See <http://docs.python.org/> to access the documentation for the current version of python.

If python support has been enabled, the python shell is accessible through the *Python* window at the bottom of the GPS window. You can also display it by using the menu *Tools->Consoles->Python*.

The full documentation on what GPS makes visible through python is available through the */Help/Python extensions*.

The same example that was used to show the GPS shell follows, now using python. As you can notice, the name of the commands is similar, although they are not run exactly in the same way. Specifically, GPS benefits from the object-oriented aspects of python to create classes and instances of these classes.

In the first line, a new instance of the class *Entity* is created through the *create_entity* function. Various methods can then be applied to that instance, including *find_all_refs*, which lists all references to that entity in the location window:

```
>>> e=GPS.Entity ("entity_name", GPS.File ("file_name.adb"))
>>> e.find_all_refs()
```

The screen representation of the classes exported by GPS to python has been modified, so that most GPS functions will return an instance of a class, but still display their output in a user-readable manner.

Python has extensive introspection capabilities. Continuing the previous example, you can find what class *e* is an instance of with the following command:

```
>>> help(e)
Help on instance of Entity:

<GPS.Entity instance>
```

It is also possible to find all attributes and methods that can be applied to *e*, as in the following example:

```
>>> dir (e)
['__doc__', '__gps_data__', '__module__', 'called_by', 'calls',
'find_all_refs']
```

Note that the list of methods may vary depending on what modules were loaded in GPS, since each module can add its own methods to any class.

In addition, the list of all existing modules and objects currently known in the interpreter can be found with the following command:

```
>>> dir ()
['GPS', 'GPSStdout', '__builtins__', '__doc__', '__name__', 'e', 'sys']
```

You can also load and execute python scripts with the *execfile* command, as in the following example:

```
>>> execfile ("test.py")
```

Python supports named parameters. Most functions exported by GPS define names for their parameters, so that you can use this Python feature, and make your scripts more readable. A notable exception to this rule are the functions that take a variable number of parameters. Using named parameters allows you to specify the parameters in any order you wish, e.g:

```
>>> e=GPS.Entity (name="foo", file=GPS.File("file.adb"))
```

13.8.5 Python modules

On startup, GPS will automatically import (with python's *import* command) all the files with the extension `.py` found in the directory `$HOME/.gps/plugin`, the directory `$prefix/share/gps/plugin` or in the directories pointed to by `GPS_CUSTOM_PATH`. These files are loaded only after all standard GPS modules have been loaded, as well as the custom files, and before the script file or batch commands specified on the command lines with the `-eval` or `-load` switches.

As a result, one can use the usual GPS functions exported to python in these startup scripts. Likewise, the script run from the command line can use functions defined in the startup files.

Since the *import* command is used, the functions defined in this modules will only be accessible by prefixing their name by the name of the file in which they are defined. For instance if a file `mystartup.py` is copied to the startup directory, and defines the function *func*, then the latter will be accessible in GPS through *mystartup.func*.

Python's own mechanism for loading files at startup (the environment variable `PYTHONSTARTUP`) is not suitable for use within the context of GPS. When python is loaded by GPS, the GPS module itself is not yet available, and thus any script that depends on that module will fail to load correctly. Instead, copy your script to one of the `plugin` directories, as documented above.

If you are writing a set of python scripts that other people will use, you need to provide the python files themselves. This is a set of `.py` files, which the user should install in the `plugin` directory.

To make the Python function accessible through GPS, this can be done:

- Either by exporting the APIs directly through Python, under the form of Actions (see the *Action* class), Menus (see the *Contextual* and *Menu* classes) or toolbar buttons (see the *ToolButton* and *Toolbar* classes);
- Or by writing an XML file with the format described in the customization section of this documentation. This XML file should create a set of actions, through the `<action>` tag, exported to the user. This allows him to either create menus to execute these commands or to bind them to special key shortcuts. The menus can be created directly in python, with the *GPS.Menu* class. The same XML can in fact be directly embedded in the python file itself and executed through *GPS.parse_xml*.

The following example defines a python command that inserts a line full of dashes ('-') at the current cursor location. This command is associated with the key binding `control-c n`, and can be distributed as a single Python file:

```
# This code can be stored in a file test.py in $HOME/.gps/plugin
from GPS import *

def add_dashes_line():
    Editor.replace_text (current_context().file().name(),
                        current_context().location().line(),
                        current_context().location().column(),
                        "-----", 0, 0)

GPS.parse_xml ("""
<action name="dashes line">
  <shell lang="python">test.add_dashes_line()</shell>
  <context>Source editor</context>
</action>
<key action="dashes line">control-c n</key>
""")
```

Several complex examples are provided in the GPS distribution, in the directory `examples/python`. These are modules that you might want to use for your own GPS, but more important that will show how GPS can be extended from Python.

If your script doesn't do what you expect it to do, there are several ways to debug it, among which the easiest is probably to add some "print" statements. Since some output of the scripts is sometimes hidden by GPS (for instance for interactive commands), you might not see this output.

In this case, you can reuse the tracing facility embedded in GPS itself. Modify the file `$HOME/.gps/traces.cfg`, and add the following line:

```
PYTHON.OUT=yes
```

This will include the python traces as part of the general traces available in the file `$HOME/.gps/log`. Note that it may slow down GPS if there is a lot of output to process.

13.8.6 Subprogram parameters

A few of the functions exported by GPS in the GPS shell or in python expect a subprogram as a parameter.

This is handled in different ways depending on what language you are using:

- GPS shell

It isn't possible to define new functions in the GPS shell. However, this concept is similar to the GPS actions (*Defining Actions*), which allow you to execute a set of commands and launch external processes.

Therefore, a subprogram parameter in the GPS shell is a string, which is the name of the action to execute.

For instance, the following code defines the action "on_edition", which is called every time a new file is edited. The action is defined in the shell itself, although this could be more conveniently done in a separate customization file:

```
parse_xml """<action name="on_edition">
           <shell>echo "File edited"</shell></action>"""
Hook "file_edited"
Hook.add %1 "on_edition"
```

- Python

Python of course has its own notion of subprogram, and GPS is fully compatible with it. As a result, the syntax is much more natural than in the GPS shell. The following example has the same result as above:

```
import GPS
def on_edition(self, *arg):
    print "File edited"
GPS.Hook ("file_edited").add (on_edition)
```

Things are in fact slightly more complex if you want to pass methods as arguments. Python has basically three notions of callable subprograms, detailed below. The following examples all create a combo box in the toolbar, which calls a subprogram whenever its value is changed. The documentation for the combo box indicates that the callback in this case takes two parameters:

- The instance of the combo
- The current selection in the combo box

The first parameter is the instance of the combo box associated with the toolbar widget, and, as always in python, you can store your own data in the instance, as shown in the examples below.

Here is the description of the various subprograms:

- Global subprograms

These are standard subprograms, found outside class definitions. There is no implicit parameter in this case. However, if you need to pass data to such a subprogram, you need to use global variables:

```

import GPS

my_var = "global data"

def on_changed (combo, choice):
    global my_var
    print "on_changed called: " + \\
        my_var + " " + combo.data + " " + choice

combo = GPS.Combo \\
    ("name", label="name", on_changed=on_changed)
GPS.Toolbar().append (combo)
combo.data = "My own data"

```

– Unbound methods

These are methods of a class. You do not specify, when you pass the method in parameter to the combo box, what instance should be passed as its first parameter. Therefore, there is no extra parameter either.

Note however than whatever class the method is defined in, the first parameter is always an instance of the class documented in the GPS documentation (in this case a `GPS.Combo` instance), not an instance of the current class.

In this first example, since we do not have access to the instance of `MyClass`, we also need to store the global data as a class component. This is a problem if multiple instances of the class can be created:

```

import GPS
class MyClass:
    my_var = "global data"
    def __init__ (self):
        self.combo = GPS.Combo \\
            ("name", label="name", on_changed=MyClass.on_changed)
        GPS.Toolbar().append (self.combo)
        self.combo.data = "My own data"

    def on_changed (combo, choice):
        ## No direct access to the instance of MyClass.
        print "on_changed called: " + \\
            MyClass.my_var + " " + combo.data + " " + choice

MyClass()

```

As the example above explains, there is no direct access to `MyClass` when executing `on_changed`. An easy workaround is the following, in which the global data can be stored in the instance of `MyClass`, and thus be different for each instance of `MyClass`:

```

import GPS
class MyClass:
    def __init__ (self):
        self.combo = GPS.Combo \\
            ("name", label="name", on_changed=MyClass.on_changed)
        GPS.Toolbar().append (self.combo)
        self.combo.data = "My own data"
        self.combo.myclass = self    ## Save the instance
        self.my_var = "global data"

    def on_changed (combo, choice):
        print "on_changed called: " + \\
            combo.myclass.my_var + " " + combo.data + " " + choice

```

```
MyClass()
```

– Bound methods

The last example works as expected, but is not convenient to use. The solution here is to use a bound method, which is a method for a specific instance of a class. Such a method always has an extra first parameter, set implicitly by Python or GPS, which is the instance of the class the method is defined in.

Notice the way we pass the method in parameter to `append()`, and the extra third argument to `on_changed` in the example below:

```
import GPS
class MyClass:
    def __init__ (self):
        self.combo = GPS.Combo \
            ("name", label="name", on_changed=self.on_changed)
        GPS.Toolbar().append (self.combo)
        self.combo.data = "My own data"
        self.my_var = "global data"

    def on_changed (self, combo, choice):
        # self is the instance of MyClass specified in call to append()
        print "on_changed called: " + \
            self.my_var + " " + combo.data + " " + choice

MyClass()
```

It is often convenient to use the object-oriented approach when writing python scripts. If for instance you want to spawn an external process, GPS provides the *GPS.Process* class. When you create an instance, you specify a callback to be called when some input is made available by the process. Matching the above example, the code would look something like:

```
class MyClass:
    def __init__ (self):
        self.process = GPS.Process
            ("command_line", on_match = self.on_match)

    def on_match (self, process, matched, unmatched);
        print "Process output: " + unmatched + matched + "\\n"
```

A more natural approach, rather than having a class that has a process field, is to directly extend the *GPS.Process* class, as in:

```
class MyClass (GPS.Process):
    def __init__ (self):
        GPS.Process.__init__ \
            (self, "command_line", on_match = self.on_match)

    def on_match (self, matched, unmatched);
        print "Process output: " + unmatched + matched + "\\n"
```

Any command that can be used on a process (such as *send*) can then directly be used on instances of *MyClass*.

There is one non-obvious improvement in the code above: the *on_match* callback has one less parameter. What happens is the following: as per the documentation of *GPS.Process.__init__*, GPS gives three arguments to its *on_match* callback: the instance of the process (*process* in the first example above), the string that matched the regular expression, and the string before that match.

In the first example above, we are passing `self.on_match`, ie a bound method, as a callback. That tells python that it should automatically, and transparently, add an extra first parameter when calling `MyClass.on_match`, which is `self`. This is why the first example has four parameters to `on_match`.

However, the second example only has three parameters, because GPS has detected that `self` (the instance of `MyClass`) and the instance of `GPS.Process` are the same in this case. Thus it doesn't add an extra parameter (`self` and `process` would have been the same).

13.8.7 Python FAQ

This section lists some of the problems that have been encountered while using Python inside GPS. This is not a general Python discussion.

Hello World! in python

Writing a python script to interact with GPS is very simple. Here we show how to create a new menu in GPS that when clicked, displays a dialog saying the famous 'Hello World!'.

Here is the code that you need to put in `hello_world.py`:

```
import GPS

def hello_world (self):
    GPS.MDI.dialog ("Hello World!")

GPS.Menu.create ("/Help/Hello World!", on_activate=hello_world)
```

In order to use this plug-in, you can launch GPS with the following command line:

```
$ gps --load=python:hello_world.py
```

If you would want the plug-in to be loaded every time you launch GPS without having to specify it on the command line, you should copy `hello_world.py` to your `$HOME/.gps/plugin/` directory or `%USERPROFILE%\gps\` under Windows.

Alternatively, you can add the directory in which your plug-in is located to your `GPS_CUSTOM_PATH` environment variable. For a description of the various environment variables used by GPS, see [Environment Variables](#).

Spawning external processes

There exist various solutions to spawn external processes from a script:

- Use the functionalities provided by the `GPS.Process` class
- Execute a GPS action through `GPS.execute_action`.
This action should have an `<external>` XML node indicating how to launch the process
- Create a pipe and execute the process with `os.popen()` calls
This solution doesn't provide a full interaction with the process, though.
- Use a standard expect library of Python

The use of an expect library may be a good solution. There are various python expect libraries that already exist.

These libraries generally try to copy the parameters of the standard `file` class. They may fail doing so, as GPS's consoles do not fully emulate all the primitive functions of that class (there is no file descriptor for instance).

When possible, it is recommended to use one of the methods above instead.

Redirecting the output of spawned processes

In general, it is possible to redirect the output of any Python script to any GPS window (either an already existing one, or creating one automatically), through the “*output*” attribute of XML configuration files.

However, there is a limitation in python that the output of processes spawned through `os.exec()` or `os.spawn()` is redirected to the standard output, and not to the usual python output that GPS has overridden.

There are two solutions for this:

- Execute the external process through a pipe

The output of the pipe is then redirected to Python’s output, as in:

```
import os, sys
def my_external():
    f = os.popen('ls')
    console = GPS.Console("ls")
    for l in f.readlines():
        console.write('    ' + l)
```

This solution allows you, at the same time, to modify the output, for instance to indent it as in the example above.

- Execute the process through GPS

You can go through the process of defining an XML customization string for GPS, and execute your process this way, as in:

```
GPS.parse_xml("""
    <action name="ls">
        <external output="output of ls">ls</external>
    </action>""")

def my_external():
    GPS.execute_action("ls")
```

This solution also allows you to send the output to a different window than the rest of your script. But you cannot filter or modify the output as in the first solution.

Contextual menus on object directories only

The following filter can be used for actions that can only execute in the Project View, and only when the user clicks on an object directory. The contextual menu entry will not be visible in other contexts:

```
<?xml version="1.0" ?>
<root>
    <filter name="object directory"
        shell_cmd="import os.path; os.path.samefile (GPS.current_context().project()).object_dirs ("
        shell_lang="python"
        module="Explorer" />

    <action name="Test on object directory">
        <filter id="object directory" />
        <shell>echo "Success"</shell>
    </action>
```

```

<contextual action="Test on object directory" >
  <Title>Test on object directory</Title>
</contextual>
</root>

```

Another example would be to have a filter so that the contextual menu only appears when on a project node in the Project View. Using `%P` in your command is not enough, since the current context when you click on a file or directory also contain information about the project this file or directory belongs to. Thus this implicit filter will not be enough to hide your contextual menu.

As a result, you need to do a slightly more complex test, where you check that the current context doesn't contains information on directories (which will disable the contextual menu for directories, files and entities). Since the command uses `%P`, GPS guarantees that a project is available.

We'll implement this contextual menu in a Python file, called `filters.py`:

```

import GPS
def on_project():
    try:
        GPS.current_context().directory()
        return False
    except:
        return True

GPS.parse_xml("""
<action name="test_filter">
<filter module="Explorer"
    shell_lang="python"
    shell_cmd="filters.on_project()" />
<shell>echo current project is %P</shell>
</action>
<contextual action="test_filter">
<title>Print current project</title>
</contextual>""")

```

The example above shows the flexibility of filters, since you can pretty much do anything you wish through the shell commands. However, it is complex to write for such a simple filter. Luckily, GPS provides a predefined filter just for that purpose, so that you can write instead, in an XML file:

```

<action name="test_filter" >
<filter id="Explorer_Project_Node" />
<shell>echo current project is %P</shell>
</action>

```

Redirecting the output to specific windows

By default, the output of all python commands is displayed in the Python console. However, you might want in some cases to create other windows in GPS for this output. This can be done in one of two ways:

- Define a new action

If the whole output of your script should be redirected to the same window, or if the script will only be used interactively through a menu or a key binding, the easiest way is to create a new XML action, and redirect the output, as in:

```

<?xml version="1.0" ?>
<root>
  <action name="redirect output" output="New Window">

```

```
<shell lang="python">print "a"</shell>
</action>
</root>
```

All the various shell commands in your action can be output in a different window, and this also applies for the output of external commands.

- Explicit redirection

If, however, you want to control in your script where the output should be sent, for instance if you can't know that statically when you write your commands, you can use the following code:

```
sys.stdin = sys.stdout = GPS.Console ("New window")
print "foo"
print (sys.stdin.read ())
sys.stdin = sys.stdout = GPS.Console ("Python")
```

The first line redirect all input and output to a new window, which is created if it doesn't exist yet. Note however that the output of `stderr` is not redirected, and you need to explicitly do it for `sys.stderr`.

The last line restore the default Python console. You must do this at the end of your script, or all scripts will continue to use the new consoles.

You can alternatively create separate objects for the output, and use them in turn:

```
my_out = GPS.Console ("New Window")
my_out2 = GPS.Console ("New Window2")

sys.stdout=my_out
print "a"
sys.stdout=my_out2
print "b"
sys.stdout=GPS.Console ("Python")
```

The parameter to the constructor `GPS.Console` indicates whether any output sent to that console should be saved by GPS, and reused for the `%1`, `%2`, ... parameters if the command is executed in a GPS action. That should generally be 1, except for `stderr` where it should be 0.

Reloading a python file in GPS

After you have made modification to a python file, you might want to reload it in GPS. This requires careful use of python commands.

Here is an example. Lets assume you have a python file ("mymod.py") which contains the following:

```
GPS.parse_xml ("""
<action name="my_action">
  <shell lang="python">mymod.myfunc()</shell>
</action>""")

def myfunc():
  print "In myfunc\n"
```

As you can guess from this file, it defines an action "my_action", that you can for instance associate with a keybinding through the Edit->Key shortcuts menu.

If this file has been copied in one of the `plug-ins` directories, it will be automatically loaded at startup.

Notice that the function `myfunc` is thus found in a separate namespace, with the name `mymod`, same as the file.

If you decide, during your GPS session, to edit this file and have the function print “In myfunc2” instead, you then have to reload the file by typing the following command in the Python console:

```
> execfile ("HOME/.gps/plugin-ins/mymod.py", mymod.__dict__)
```

The first parameter is the full path to the file that you want to reload. The second argument is less obvious, but indicates that the file should be reloaded in the namespace *mymod*.

If you omit the optional second parameter, Python will load the file, but the function *myfunc* will be defined in the global namespace, and thus the new definition is accessible through:

```
> myfunc()
```

Thus, the key shortcut you had set, which still executes *mymod.myfunc()* will keep executing the old definition.

By default, GPS provides a contextual menu when you are editing a Python file. This contextual menu (Python->Reload module) will take care of all the above details.

Printing the GPS Python documentation

The python extension provided by GPS is fully documented in this manual and a separate manual accessible through the Help menu in GPS.

However, this documentation is provided in HTML, and might not be the best suitable for printing, if you wish to do so.

The following paragraph explains how you can generate your own documentation for any python module, including GPS, and print the result:

```
import pydoc
pydoc.writedoc (GPS)
```

In the last command, *GPS* is the name of the module that you want to print the documentation for.

These commands generate a `.html` file in the current directory.

Alternatively, you can generate a simple text file with:

```
e=file("./python_doc", "w")
e.write (pydoc.text.document (GPS))
e.flush()
```

This text file includes bold characters by default. Such bold characters are correctly interpreted by tools such as `a2ps` which can be used to convert the text file into a postscript document.

Automatically loading python files at startup

At startup, GPS will automatically load all python files found in the directories `share/gps/plugin-ins` and `$HOME/.gps/plugin-ins`.

In addition, python files located under `<prefix>/share/gps/python` can be imported (using the *import* command) by any python script.

You can also set the *PYTHONPATH* environment variable to add other directories to the python search path.

Hiding contextual menus

GPS provides most of its tools through contextual menus, accessed by right clicking in various parts of GPS. Due to the number of tools provided by GPS, these contextual menus tend to be big, and you might want to control what should be displayed in them. There are several ways to control this:

- Define appropriate filters for your actions

If you are creating your own contextual menus through customization files and XML, these menus are associated with actions (*<action>*) that you have created yourself most of the time. In this case, you need to define filters appropriately, through the *<filter>* tag, to decide when the action is relevant, and therefore when the contextual menu should be displayed.

- Use shell commands to hide the menus

If you want to control the visibility of predefined contextual menus, or for menus where you cannot easily modify the associated filter, you can use shell and python commands to hide the menu entry. For this, you will need to find out the name of the menu, which can be done by checking the list returned by *GPS.Contextual.list()* and using the most likely entry. This name is also the value of the *<title>* tag for contextual menus that you have created yourself. Using this name, you can then disable the contextual menu by executing:

```
GPS.Contextual ("name").hide()
```

in the python console

Creating custom graphical interfaces

GPS is based on the Gtk+ graphical toolkit, which is available under many platforms and for many programming languages.

In particular, GPS comes with pygobject, a python binding to Gtk+. Using pygobject, you will be able to create your own dialogs and graphical windows using the python capabilities provided by GPS.

See the menu *Help->Python Extensions*, in particular the *GPS.MDI* documentation, for a sample of code on how to create your own graphical interfaces and integrate them in GPS.

13.8.8 Hooks

A **hook** is a named set of commands to be executed on particular occasions as a result of user actions in GPS.

GPS and its various modules define a number of standard hooks, which are called for instance when a new project is loaded, when a file is edited, and so on. You can define your own commands to be executed in such cases.

You can find out the list of hooks that GPS currently knows about by calling the **Hook.list** function, which takes no argument, and returns a list of hook names that you can use. More advanced description for each hook is available through the *Help->Python Extensions*:

```
GPS> Hook.list
project_changed
open_file_action_hook
preferences_changed
[...]
```

```
Python> GPS.Hook.list()
```

The description of each hooks includes a pointer to the type of the hook, that is what parameters the subprograms in this hook will receive. For instance:

The list of all known hook types can be found through the **Hook.list_types** command. This takes no argument and returns a list of all known types of hooks. As before, you can more information for each of these type through a call to **Hook.describe_type**.

Adding commands to hooks

You can add your own command to existing hooks through a call to the **Hook.add** command. Whenever the hook is executed by GPS or another script, your command will also be executed, and will be given the parameters that were specified when the hook is run. The first parameter is always the name of the hook being executed.

This **Hook.add** applies to an instance of the hook class, and takes one parameter, the command to be executed. This is a subprogram parameter (*Subprogram parameters*).

- GPS shell

The command can be any GPS action (*Defining Actions*). The arguments for the hook will be passed to the action, and are available as \$1, \$2, ...; In the following example, the message “Just executed the hook: project_changed” will be printed in the Shell console. Note that we are defining the action to be executed inline, but this could in fact be defined in a separate XML customization file for convenience:

```
GPS> parse_xml """<action name="my_action"><shell>echo "Just executed the hook"</shell></action>"""
GPS> Hook project_changed
GPS> Hook.add %1 "my_action"
```

- Python

The command must be a subprogram to execute. The arguments for the hook will be passed to this subprogram. In the following example, the message “The hook project_changed was executed by GPS” will be displayed in the Python console whenever the project changes:

```
def my_callback (name):
    print "The hook " + name + " was executed by GPS"
GPS.Hook ("project_changed").add (my_callback)
```

The example above shows the simplest type of hook, which doesn’t take any argument. However, most hooks receive several parameters. For instance, the hook “file_edited” receives the file name as a parameter.

- GPS shell

The following code will print the name of the hook (“file_edited”) and the name of the file in the shell console every time a file is open by GPS:

```
GPS> parse_xml """<action name="my_action"><shell>echo name=$1 file=$2</shell></action>"""
GPS> Hook "file_edited"
GPS> Hook.add %1 "my_action"
```

- Python

The following code prints the name of the file being edited by GPS in the python console whenever a new editor is opened. The second argument is of type GPS.File:

```
def my_file_callback (name, file):
    print "Editing " + file.name()
GPS.Hook ("file_edited").add (my_file_callback)
```

Action hooks

Some hooks have a special use in GPS. Their name always ends with “_action_hook”.

As opposed to the standard hooks described in the previous section, the execution of the action hooks stops as soon as one of the subprograms returns a True value (“1” or “true”). The subprograms associated with that hook are executed one after the other. If any such subprogram knows how to act for that hook, it should do the appropriate action and return “1”.

Other action hooks expect a string as a return value instead of a boolean. The execution will stop when a subprogram returns a non-empty string.

This mechanism is used extensively by GPS internally. For instance, whenever a file needs to be opened in an editor, GPS executes the “open_file_action_hook” hook to request its editing. Several modules are connected to that hook.

One of the first modules to be executed is the external editor module. If the user has chosen to use an external editor, this module will simply spawn Emacs or the external editor that the user has selected, and return 1. This immediately stops the execution of the “open_file_action_hook”.

However, if the user doesn’t want to use external editors, this module will return 0. This will keep executing the hook, and in particular will execute the source editor module, which will always act and open an editor internally in GPS.

This is a very flexible mechanism. In your own script, you could choose to have some special handling for files with a “.foo” extension for instance. If the user wants to open such a file, you would spawn for instance an external command (say “my_editor”) on this file, instead of opening it in GPS.

This is done with a code similar to the following:

```
from os.path import *
import os
def my_foo_handler(name, file, line, column,
                  column_end, enable_nav, new_file, reload):
    if splitext(file.name())[1] == ".foo":
        os.spawnv(
            os.P_NOWAIT, "/usr/bin/emacs", ("emacs", file.name()))
        return 1  ## Prevent further execution of the hook
    return 0  ## Let other subprograms in the hook do their job
```

```
GPS.Hook("open_file_action_hook").add(my_foo_handler)
```

Running hooks

Any module in GPS is responsible for running the hooks when appropriate. Most of the time, the subprograms exported by GPS to the scripting languages will properly run the hook. But you might also need to run them in your own scripts.

As usual, this will result in the execution of all the functions bound to that hook, whether they are defined in Ada or in any of the scripting languages.

This is done through the **Hook.run** command. This applies to an instance of the Hook class, and a variable number of arguments. These must be in the right order and of the right type for that specific type of hook.

If you are running an action hook, the execution will stop as usual as soon as one of the subprograms return a True value.

The following example shows how to run a simple hook with no parameter, and a more complex hook with several parameters. The latter will in fact request the opening of an editor for the file in GPS, and thus has an immediately visible effect on the interface. The file is opened at line 100. See the description of the hook for more information on the other parameters:

```
GPS.Hook("project_changed").run()
GPS.Hook("open_file_action_hook").run \
    (GPS.File("test.adb"), 100, 1, 0, 1, 1, 1)
```

Creating new hooks

The list of hooks known to GPS is fully dynamic. GPS itself declares a number of hooks, mostly for its internal use although of course you can also connect to them.

But you can also create your own hooks to report events happening in your own modules and programs. This way, any other script or GPS module can react to these events.

Such hooks can either be of a type exported by GPS, which constraints the list of parameters for the callbacks, but make such hooks more portable and secure; or they can be of a general type, which allows basically any kind of parameters. In the latter case, checks are done at runtime to ensure that the subprogram that is called as a result of running the hook has the right number of parameters. If this isn't the case, GPS will complain and display error messages. Such general hooks will also not pass their parameters to other scripting languages.

Creating new hooks is done through a call to **Hook.register**. This function takes two arguments: the name of the hook you are creating, and the type of the hook.

The name of the hook is left to you. Any character is allowed in that name, although using only alphanumerical characters is recommended.

The type of the hook must be one of the following:

- "" (the empty string)

This indicates that the hook doesn't take any argument. None should be given to **Hook.run**, and none should be expected by the various commands connected to that hook, apart from the hook name itself.
- one of the values returned by **Hook.list_types**

This indicates that the hook is of one of the types exported by GPS itself. The advantage of using such explicit types as opposed to "general" is that GPS is able to make more tests for the validity of the parameters. Such hooks can also be connected to from other scripting languages.
- "general"

This indicates that the hook is of the general type that allows any number of parameter, of any type. Other scripts will be able to connect to it, but will not be executed when the hook is run if they do not expect the same number of parameters that was given to **Hook.run**. Other scripts in other language will only receive the hook name in parameter, not the full list of parameters.

A small trick worth noting: if the command bound to a hook doesn't have the right number of parameters that this hook provides, the command will not be executed and GPS will report an error. You can make sure that your command will always be executed by either giving default values for its parameter, or by using python's syntax to indicate a variable number of arguments.

This is especially useful if you are connecting to a "general" hook, since you do not really know in advance how many parameters the call of **Hook.run** will provide:

```
## This callback can be connected to any type of hook
def trace (name, *args):
    print "hook=" + name

## This callback can be connected to hooks with one or two parameters
def trace2 (name, arg1, arg2=100):
    print "hook=" + str (arg1) + str (arg2)

Hook.register ("my_custom_hook", "general")
Hook ("my_custom_hook").add (trace2)
Hook ("my_custom_hook").run (1, 2) ## Prints 1 2
Hook ("my_custom_hook").run (1)   ## Prints 1 100
```

13.9 Adding support for new Version Control Systems

13.9.1 Custom VCS interfaces

The Version Control interface in GPS can be customized, either to refine the behavior of the existing system and adapt it to specific needs, or to add support for other Version Control systems.

Custom VCS interfaces are defined through XML files and Python plugins. Those files are read in the same location as all the other XML and Python customizations that GPS offers. See *Customizing through XML and Python files* for a complete description.

There are three steps to follow when creating a custom VCS interface. The first step is to describe the VCS itself, the second step is to implement actions corresponding to all the operations that this VCS can perform, and the third step is to define the layout of the menus. The following three sections (*Describing a VCS*, *Implementing VCS actions*, and *Implementing VCS menus*) describe those steps.

GPS is distributed with XML/Python files describing the interfaces to ClearCase, CVS, Subversion, Git and Mercurial (experimental support). These XML/Python files are located in the directory *share/gps/plug-ins* in the GPS installation, and can be used as a reference for implementing new custom VCS interfaces.

13.9.2 Describing a VCS

The VCS node

The *vcs* node is the toplevel node which contains the description of the general behavior expected from the VCS. It has the following attributes:

name The attribute *name* indicates the identifier of the VCS. The casing of this name is important, and the same casing must be used in the project files.

absolute_names The attribute *absolute_names* indicates the behavior of the VCS relative to file names, and can take the values *TRUE* or *FALSE*. If it is set to *TRUE*, it means that all commands in the VCS will work on absolute file names. If it set to *FALSE*, it means that all actions work on base file names, and that GPS will move to the appropriate directory before executing an action.

group_queries_by_directory The attribute *group_queries_by_directory* indicates that, when querying status for all the source files in a directory, a query for the directory should be launched, instead of launching a query for multiple files. This operation is faster on some Version Control systems. By default, this is set to *FALSE*.

ignore_file The attribute *ignore_file* specifies the name of the file used by the VCS Explorer to get the list of files to ignore. By default for the CVS mode this is set to *.cvsignore*.

atomic_commands The attribute *atomic_commands* specifies if the VCS supports atomicity and can take the values *TRUE* or *FALSE*. If it is set to *TRUE* it means that the VCS supports atomic commands. It is *FALSE* by default. This attribute is important to trigger the activities group commit feature. See *The VCS Activities*.

path_style The attribute *path_style* specifies which kind of directory separator is supported by the VCS and can take the values *UNIX*, *DOS*, *Cygwin* or *System_Default*. The later value is the default value. With this attribute it is possible to control the directory separator to use when specifying files to the VCS. For the *Cygwin* case the drive is specified as */cygdrive/<drive>*.

dir_sep Alias for *path_style*, obsolescent.

commit_directory The attribute *commit_directory* specifies if the VCS supports commit on directories and can take the values *TRUE* or *FALSE*. If it is set to *TRUE* it means that the VCS supports commit on directories this is the case for *Subversion* for example.

administrative_directory The attribute *administrative_directory* specifies the name of the directory where the external VCS stores the local repository information. For example for Subversion this is `.svn`. This information is used when the project is setup to select automatically the external VCS. *Version Control System*.

prev_revision A string which can be used when querying revisions to indicate the previous revision of a file, for instance “PREV” for subversion.

head_revision A string which can be used when querying revisions to indicate the latest revision of a file, for instance “HEAD” for subversion.

require_log The attribute *require_log* specifies if the VCS require a log for the commit/add/delete actions. It can take the values *TRUE* or *FALSE*. If it is set to *TRUE* GPS will ensure that a log is created for each file. If it is set to *FALSE* GPS will not ask for log, it is expected to be handled by the external VCS.

Note that to support group commit with shared log on GPS both *absolute_name* and *atomic_commands* must be true. This is the case for the Subversion VCS for example.

Here is an example, adapted to the use of CVS:

```
<vcs name="Custom CVS" absolute_names="FALSE">
    (... description of action associations ...)
    (... description of supported status ...)
    (... description of output parsers ...)
</vcs>
```

Associating actions to operations

GPS knows about a certain set of predefined ‘operations’ that a VCS can perform. The user can decide to implement some of them - not necessarily all of them - in this section.

The following node is used to associate a predefined operation to an action:

```
<OPERATION action="ACTION_LABEL" label="NAME OF OPERATION" />
```

Where:

OPERATION is the name of the predefined action. The list of predefined actions is described in *Implementing VCS actions*,

ACTION_LABEL is the name of the corresponding gps Action that will be launched when GPS wants to ask the VCS to perform *OPERATION*,

NAME OF OPERATION is the name that will appear in the GPS menus when working on a file under the control of the defined VCS.

Defining revision information

Some VCS reports revisions number from which it is possible to deduce the related branches. This is the case in CVS for example where a revision number for a branch uses as prefix the branch point revision number. For such VCS it is possible to specify two regular expressions:

‘parent_revision’ Parse the revision number and report as first match the parent revision:

```
<parent_revision regexp="..." />
```

For CVS on **1.2.4.5** it must match **1.2**.

‘branch_root_revision’ Parse the revision number and report as first match the branch root revision:

```
<branch_root_revision regexp="..." />
```

For CVS on **1.2.4.5** it must match **1.2.4**.

Defining status

All VCS have the notion of ‘status’ or ‘state’ to describe the relationship between the local file and the repository. The XML node *status* is used to describe the status that are known to a custom VCS, and the icons associated to it:

```
<status label="STATUS_LABEL" stock="STOCK_LABEL" />
```

Where:

STATUS_LABEL is the name of the status, for example ‘Up to date’ or ‘Needs update’ in the context of CVS.

STOCK_LABEL is the stock identifier of the icon associated to this status, that will be used, for example, in the VCS Explorer. See section *Adding stock icons* for more details on how to define stock icons.

Note that the order in which status are defined in the XML file is important: the first status to be displayed must correspond to the status ‘Up-to-date’ or equivalent.

Output parsers

There are cases in which GPS needs to parse the output of the VCS commands: when querying the status, or when ‘annotating’ a file.

The following parsers can be implemented in the *vcs* node.

‘<*status_parser*>’, ‘<*local_status_parser*>’ and ‘<*update_parser*>’ These parsers are used by the command `VCS.status_parse`, to parse a string for the status of files controlled by a VCS.

They accept the following child nodes:

‘<*regexp*>’ (*mandatory*) Indicates the regular expression to match.

‘<*file_index*>’ An index of a parenthesized expression in *regexp* that contains the name of a file.

‘<*status_index*>’ An index of a parenthesized expression in *regexp* that contains the file status. This status is passed through the regular expressions defined in the *status_matcher* nodes, see below.

‘<*local_revision_index*>’ An index of a parenthesized expression in *regexp* that contains the name of the local revision (the version of the file that was checked out).

‘<*repository_revision_index*>’ An index of a parenthesized expression in *regexp* that contains the name of the repository revision (the latest version of the file in the VCS).

‘<*status_matcher*>’ A regular expression which, when matching an expressions, identifies the status passed in the node attribute *label*.

‘<*annotations_parser*>’ This parser is used by the command `VCS.annotations_parse`, to parse a string for annotations in a file controlled by a VCS.

It accepts the following child nodes:

‘<*regexp*>’ (*mandatory*) Indicates the regular expression to match.

‘<*repository_revision_index*>’ (*mandatory*) An index of a parenthesized expression in *regexp* that contains the repository revision of the line.

‘<*author_index*>’ An index of a parenthesized expression in *regexp* that contains the author of the line.

‘<*date_index*>’ An index of a parenthesized expression in *regexp* that contains the date of the line.

<file_index> An index of a parenthesized expression in *regex* that indicates the part of the line that belongs to the file.

<tooltip_pattern> A template pattern that will be used to format the tooltip information. It can contain text and reference parenthesized expressions in *regex* using `\n` (where *n* represents the *n*th expression in *regex*).

<log_parser> This parser is used by the command `VCS.log_parse`, to parse a string for revision histories in a file controlled by a VCS.

It accepts the following child nodes:

<regex> (*mandatory*) Indicates the regular expression to match.

<repository_revision_index> (*mandatory*) An index of a parenthesized expression in *regex* that contains the repository revision of the log.

<author_index> An index of a parenthesized expression in *regex* that contains the author of the log.

<date_index> An index of a parenthesized expression in *regex* that contains the date of the log.

<log_index> An index of a parenthesized expression in *regex* that contains the actual text of the log.

<revision_parser> This parser is used by the command `VCS.revision_parse`, to parse a string for revision tags and branches in a file controlled by a VCS.

It accepts the following child nodes:

<regex> (*mandatory*) Indicates the regular expression to match.

<sym_index> (*mandatory*) An index of a parenthesized expression in *regex* that contains the tags or branches symbolic name of the revision.

<repository_revision_index> (*mandatory*) An index of a parenthesized expression in *regex* that contains the repository revision number of the revision.

13.9.3 Implementing VCS actions

A number of ‘standard’ VCS operations are known to GPS. Each of these operations can be implemented, using Actions. See *Defining Actions* for a complete description of how to implement actions.

Here is a list of all the defined VCS operations, and their parameters:

status_files

- \$1 = whether the log files should be cleared when obtaining up-to-date status
- \$2- = the list of files to query status for.

Query the status for a list of files. This should perform a complete VCS query and return results as complete as possible.

status_dir

- \$1 = the directory.

Same as above, but works on all the files in one directory.

status_dir_recursive

- \$1 = the directory.

Same as above, but works on all the files in one directory and all subdirectories, recursively.

local_status_files

- \$* = list of files

Query the local status for specified files. This query should be as fast as possible, not connecting to any remote VCS. The results need not be complete, but it is not useful to implement this command if the output does not contain at least the working revision.

open

- \$* = list of files

Open files or directories for editing. This command should be implemented on any VCS that require an explicit check-out/open/edit action before being able to edit a file.

update

- \$* = list of files

Bring the specified files in sync with the latest repository revision.

resolved

- \$* = list of files

Mark files' merge conflicts as resolved. Some version control systems (like Subversion) will block any commit until this action is called.

commit

- \$1 = log file
- \$2- = list of files

Commit/submit/check-in files or directories with provided log. The log is passed in a file.

commit_dir

- \$1 = log
- \$2 = directory

Commit/submit one directory with provided log. The log is passed in a file.

history_text

- \$1 = file

Query the entire changelog history for the specified file. The result is expected to be placed into an editor as plain text.

history

- \$1 = file

Query the entire changelog history for the specified file. The result is expected to be placed into a Revision View.

history_revision

- \$1 = revision
- \$2 = file

Query the history for corresponding revision of the specified file.

annotate

- \$1 = file

Query the annotations for a file.

add

- \$1 = log
- \$2- = list of files or dirs

Add files/dirs to the repository, with the provided revision log. The added files/dirs are committed.

add_no_commit

- \$1 = log
- \$2- = list of files or dirs

Add files/dirs to the repository, with the provided revision log. The added files/dirs are not committed.

remove

- \$1 = log
- \$2 = file or dir

Remove file/dir from the repository, with the provided revision log.

remove_no_commit

- \$1 = log
- \$2 = file or dir

Remove file/dir from the repository, with the provided revision log. The removed files/dirs are not committed.

revert

- \$* = files

Revert the local file to repository revision, cancelling all local changes, and close the file for editing if it was open.

diff_patch

- \$1 = file

Create a textual diff for the given file. This command is used to build the activity patch file.

diff_head

- \$1 = file

Display a visual comparison between the local file and the latest repository revision. The diff command must report a *normal* diff as opposed to *context* or *unified* ones.

diff_base_head

- \$1 = file

Display a visual comparison between the revision from which the file has been checked-out and the latest revision. The diff command must report a *normal* diff as opposed to *context* or *unified* ones.

diff_working

- \$1 = file

Display a visual comparison between the local file and the revision from which it was obtained. The diff command must report a *normal* diff as opposed to *context* or *unified* ones.

diff

- \$1 = rev

- \$2 = file

Display a visual comparison between the local file and the specified revision. The diff command must report a *normal* diff as opposed to *context* or *unified* ones.

diff2

- \$1 = revision 1
- \$2 = revision 2
- \$3 = file

Display a visual comparison between the two specified revisions of the file. The diff command must report a *normal* diff as opposed to *context* or *unified* ones.

13.9.4 Implementing VCS menus

GPS defines a standard set of Actions to interact with Version Control Systems. All these actions can be viewed, for instance, in the “VCS” of the Key Shortcuts dialog (see *The Key Manager Dialog*).

A Python facility exists in plugin *vcs.py* to associate menu items to VCS actions. This facility defines in one place the VCS menus that are to be displayed in the global VCS menu, in the contextual menus on contexts that contain files, and on the menus in the VCS explorer.

To use this facility, you must first define a list of associations in Python, and then register this list through a call to *vcs.register_vcs_actions*.

This function takes as parameter:

the name of the version control system as defined in the *name* attribute of the *vcs* node in the XML definition.

a list of dictionaries of the form *ACTION* : *<name of the vcs action>*, *LABEL*: *<menu label>*. The predefined *SEPARATOR* dictionary can be used to indicate a separator which will be displayed in the contextual menus on file and on the VCS Explorer.

If you have defined a custom VCS in a previous version of GPS, you will need to define your menus through this facility. The easiest is to simply copy one of the existing plugins (for instance *subversion.py* or *clearcase.py*) and simply change the first parameter in the call to *register_vcs_actions*.

13.10 The Server Mode

In order to give access to the GPS capabilities from external processes (e.g. *Emacs*), GPS can be launched in *server mode*.

The two relevant command line switches are *-server* and *-hide*.

-server will open a socket on the given port, allowing multiple clients to connect to a running GPS, and sending GPS shell or python commands.

-hide tells GPS not to display its main window when starting. note that under unix systems, you still need to have access to the current screen (as determined by the *DISPLAY* environment variable) in this mode.

Using the two switches together provides a way to launch GPS as a background process with no initial user interface.

Clients connecting through a standard socket have access to a simple shell using *GPS>>* as the separating prompt between each command. This is needed in order to determine when the output (result) of a command is terminated.

All the GPS shell commands (as defined in *The GPS Shell*) are available from this shell. In addition, the python interpreter, if enabled, is also available through the use of the *python* prefix before a python command.

For example, sending *pwd* through the socket will send the *pwd* command through the GPS shell and display the result on the socket; similarly, sending *python GPS.pwd()* will send the *GPS.help()* command through the python interpreter (see *The Python Interpreter* for more details).

The socket shell provides also additional commands:

- **logout** This command will inform the GPS server that the connection should now be closed.
- **id <string>** This command will register the current session with a given string. This string can then be used within GPS itself (for example via a .xml or python plug-in) to display extra information to the client via the socket, using the command `GPS.Socket().send`.

For example, let suppose that we start *gps* with the `-server=1234` command: this will bring up GPS as usual.

Now, on a separate terminal, create a simple client by typing the following:

```
telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
GPS>> id test-1
id set to 'test-1'
GPS>> pwd
c:\\working-dir\\
GPS>>
```

Then in the GPS Python Console:

```
>>> GPS.Socket ("test-1").send ("hello, it's time to logout\\n");
```

At this point, the following is received on the client (telnet) side:

```
GPS>> hello, it's time to logout
```

We can then close the client:

```
logout
Connection closed by foreign host.
```

13.11 Adding project templates

The Project template wizard lists a selection of templates. The default set is found automatically by GPS in the *share/gps/templates* directory of your GPS installation.

It is possible to register new directories in which GPS will look for templates, by using the Shell/Python command `GPS.ProjectTemplate.add_templates_dir`.

To create a new project template, first create a subdirectory in the *share/gps/templates/* directory, or in one of the directories which has been registered through `GPS.ProjectTemplate.add_templates_dir`. Then, in this directory, create one template description file.

A template description file is a text file with the *.gpt* extension, with the following syntax:

```
Name: <name>
Category: <category>
Project: <project file>
<optional_hook_line>

<variable_1>: <variable_1_default_value>: <variable_1_description>
<variable_2>: <variable_2_default_value>: <variable_3_description>
```

<etc>

```
[Description]
<the description>
```

Where the following should be defined:

- <name> The name of the template as it will appear in the template tree in the project template wizard.
- <category> The category in which the template will be inserted in the template tree. There can be multiple levels of categories, separated with /.
- <variable_1> A name which will be substituted in the template files when deploying the template, see below.
- <variable_1_default_value> The default value for variable 1, which will appear in the project template wizard.
- <variable_1_description> The description of variable 1.
- <optional_hook_line> An optional line of the form *post_hook: <python_file>* where *<python_file>* is the name of a Python file present in the same directory as the template description file. This Python file will be run by GPS once, right after the project template is deployed
- <description> A short paragraph describing the project template. This paragraph will be displayed in the project template wizard when the template is selected in the tree.

When deploying templates, GPS will copy in the destination directory chosen by the use all files and directories present in the directory that contains the template description file, except the Python file indicated as *post_hook*, and the template description file itself.

As it deploys templates, GPS will replace strings of the form `@code{@_<variable_name>_}` with the value of the variable. If *<variable_name>* is all lower case, the substitution will be transformed to lower-case. If *<variable_name>* is in *Mixed_Case*, the substitution will be transformed into *Mixed_Case* as well. If it is in upper case, then the substitution will contain the original value specified by the user.

ENVIRONMENT

14.1 Command Line Options

The command line options are:

Usage:

```
gps [options] [-Pproject-file] [[+line] source1] [[+line] source2] ...
```

Options:

```
--help           Show this help message and exit
--version        Show the GPS version and exit
--debug[=program] Start a debug session and optionally load the
                  program with the given arguments
--debugger debugger Specify the debugger's command line
--hide           Hide GPS main window
--host=tools_host Use tools_host to launch tools (e.g. gdb)
--target=TARG:PRO Load program on machine TARG using protocol PRO
--load=lang:file  Execute an external file written in the
                  language lang
--eval=lang:file  Execute an in-line script written in the
                  language lang
-XVAR=VALUE       Specify a value for a scenario variable
--readonly       Open all files in read-only mode
--server=port     Start GPS in server mode, opening a socket on the
                  given port
--tracelist       Output the current configuration for logs
--traceon=name    Activate the logs for a given module
--traceoff=name   Deactivate the logs for a given module
--tracefile=file  Parse an alternate configuration file for the logs
```

Source files can be absolute or relative pathnames.

If you prepend a file name with '=', this file will be searched anywhere on the project's source path

To open a file at a given line, use the '+line' prefix, e.g.

```
gps +40 source.adb
```

```
`tools_host` corresponds to a remote host's nickname as defined in
:ref:`Setup_the_remote_servers`.
```

By default, files specified on the command line are taken as is and can be absolute or relative pathnames. In addition, if you prepend a filename with the = character, then GPS will look for the file in the source search path of the project.

When no project is specified on the command line, GPS tries to find one and otherwise displays the *welcome dialog*.

14.2 Environment Variables

The following environment variables can be set to override some default settings in GPS:

GPS_HOME Override the variable HOME if present. All the configuration files and directories used by GPS are either relative to \$HOME/.gps (%HOME%.gps on Windows) if GPS_HOME is not set, or to \$GPS_HOME/.gps (respectively %GPS_HOME%.gps) if set.

GPS_DOC_PATH Set the search path for the documentation. *Adding documentation.*

GPS_CUSTOM_PATH Contains a list of directories to search for custom files. See *Customizing through XML and Python files* for more details.

GPS_CHANGELOG_USER Contains the user and e-mail to use in the global ChangeLog files. Note that the common usage is to have two spaces between the name and the e-mail. Ex: “John Does <john.doe@home.com>”

GPS_STARTUP_PATH Contains the value of the PATH environment variable just before GPS was started. This is used by GPS to restore the proper environment before spawning applications, no matter what particular directories it needed to set for its own purpose.

GPS_STARTUP_LD_LIBRARY_PATH Same as **GPS_STARTUP_LD_LIBRARY_PATH** but for the **LD_LIBRARY_PATH** variable.

GPS_PYTHONHOME If set, the Python interpreter will look for libraries in the subdirectory lib/python<version> of the directory contained in **GPS_PYTHONHOME**.

GNAT_CODE_PAGE This variable can be set to **CP_ACP** or **CP_UTF8** and is used to control the code page used on Windows platform. The default is **CP_UTF8** to support more languages. If file or directory names are using accents for example it may be necessary to set this variable to **CP_ACP** which is the default Windows ANSI code page.

GPS_ROOT Override and hardcode the default root installation directory. This variable should in general not be needed, except by GPS developers, in some rare circumstances. GPS will find all its resource files (e.g. images, plug-ins, xml files) from this root prefix, so setting **GPS_ROOT** to a wrong value will cause GPS to misbehave.

GPS_MEMORY_MONITOR If set, GPS will add special code on every allocation and deallocation, thus slowing things down a bit, that makes it possible to check where the biggest amount of memory is allocated, through the *GPS.debug_memory_usage* python command.

14.3 Files

\$HOME/.gps GPS state directory. Defaults to C:.\gps under Windows systems if HOME or USERPROFILE environment variables are not defined.

\$HOME/.gps/log Log file created automatically by GPS. When GPS is running, it will create a file named log.<pid>, where <pid> is the GPS process id, so that multiple GPS sessions do not clobber each other's log. In case of a successful session, this file is renamed log when exiting; in case of an unexpected exit (a bug box will be displayed), the log file is kept under its original name.

Note that the name of the log file is configured by the *traces.cfg* file.

\$HOME/.gps/aliases File containing the user-defined aliases (*Defining text aliases*).

\$HOME/.gps/plugin-ins Directory containing files with user-defined plug-ins. All xml and python files found under this directory are loaded by GPS during start up. You can create/edit these files to add your own menu/tool-bar entries in GPS, or define support for new languages. *Customizing through XML and Python files* and *Adding support for new languages*.

- \$HOME/.gps/keys.xml** Contains all the key bindings for the actions defined in GPS or in the custom files. This only contains the key bindings overridden through the key shortcuts editor (see *The Key Manager Dialog*).
- \$HOME/.gps/gps.css** Configuration and theme file for gtk. This file can be change specific aspects of the look of GPS. Its contents overrides any other style information set by your default gtk+ theme (as selected in the Preferences dialog) and GPS's `prefix/share/gps/gps.css` file.
- \$HOME/.gps/perspectives6.xml** Desktop file in XML format (using the menu *File* → *Save More* → *Desktop*), loaded automatically if found.
- \$HOME/.gps/locations.xml** This file contains the list of locations that GPS has previously edited. It corresponds to the history navigation (*Navigate* → *Back* and *Navigate* → *Forward*)
- \$HOME/.gps/properties.xml** This file is used to store file-specific properties across GPS sessions. In particular, it contains the encoding to use for various files when the default encoding isn't appropriate.
- \$HOME/.gps/histories.xml** Contains the state and history of combo boxes (for instance the *Build* → *Run* → *Custom...* dialog).
- \$HOME/.gps/targets.xml** Contains the build targets defined by the user.
- \$HOME/.gps/preferences.xml** Contains all the preferences in XML format, as specified in the preferences menu.
- \$HOME/.gps/traces.cfg** Default configuration for the system traces. These traces are used to analyze problems with GPS. By default, they are sent to the file `$HOME/.gps/log.<pid>`.
- This file is created automatically when the `$HOME/.gps/` directory is created. If you remove it manually, it won't be recreated the next time you start GPS.
- \$HOME/.gps/startup.xml** This file contains the list of scripts to load at startup, as well as additional code that need to be executed to setup the script.
- \$HOME/.gpe/activity_log.tmplt** Template file used to generate activities' group commit-log and patch file's header. If not present the system wide template (see below) is used. The set of configurable tags are described into this template.
- prefix** The prefix directory where GPS is installed, e.g `/opt/gps`.
- prefix/bin** The directory containing the GPS executables.
- prefix/etc/gps** The directory containing global configuration files for GPS.
- prefix/lib** This directory contains the shared libraries used by GPS.
- prefix/share/doc/gps/html** GPS will look for all the documentation files under this directory.
- prefix/share/examples/gps** This directory contains source code examples.
- prefix/share/examples/gps/language** This directory contains sources showing how to provide a shared library to dynamically define a new language. See *Adding support for new languages*.
- prefix/share/examples/gps/tutorial** This directory contains the sources used by the GPS tutorial.
- See `gps-tutorial.html`.
- prefix/share/gps/support** Directoring containing mandatory plug-ins for GPS, which are systematically loaded at startup.
- prefix/share/gps/plugin-ins** Directory containing files with system-wide plug-ins (xml and python files) loaded automatically at start-up.
- prefix/share/gps/library** Directory containing files with system-wide plug-ins (xml and python files) that are not loaded automatically at startup, but can be selected in the Plug-ins editor.
- prefix/share/gps/gps-splash.png** Splash screen displayed by default when GPS is started.

prefix/share/gps/perspectives6.xml This is the description of the default desktop that GPS uses when the user hasn't defined his own default desktop and no project specific desktop exists. You can modify this file if you want, knowing that this will impact all users of GPS sharing this installation. The format of this file is the same as `$HOME/.gps/perspectives6.xml`, which can be copied from your own directory if you wish.

prefix/share/gps/default.gpr Default project used by GPS. Can be modified after installation time to provide useful default for a given system or project.

prefix/share/gps/readonly.gpr Project used by GPS as the default project when working in a read-only directory.

prefix/share/gps/activity_log.tmplt Template file used by default to generate activities' group commit-log and patch file's header. This file can be copied into user home directory and customized (see above).

prefix/share/locale Directory used to retrieve the translation files, when relevant.

14.4 Reporting Suggestions and Bugs

If you would like to make suggestions about GPS, or if you encountered a bug, please report it to <mailto:report@gnat.com> if you are a supported user, and to <mailto:gps-devel@lists.act-europe.fr> otherwise.

Please try to include a detailed description of the problem, including sources to reproduce it if possible/needed, and/or a scenario describing the actions performed to reproduce the problem, as well as the tools (e.g *debugger*, *compiler*, *call graph*) involved.

The files `$HOME/.gps/log` may also bring some useful information when reporting a bug.

In case GPS generates a bug box, the log file will be kept under a separate name (`$HOME/.gps/log.<pid>`) so that it does not get erased by further sessions. Be sure to include the right log file when reporting a bug box.

14.5 Solving Problems

This section addresses some common problems that may arise when using or installing GPS.

GPS crashes on some GNU/Linux distributions at start up

Look at the `~/ .gps/log.xxx` file and if there is a message that looks like:

```
[GPS.MAIN_WINDOW] 1/16 loading gps-animation.png [UNEXPECTED_EXCEPTION]
1/17 Unexpected exception: Exception name: CONSTRAINT_ERROR _UNEX-
PECTED_EXCEPTION_ Message: gtk-image.adb:281 access check failed
```

Then it means either that there is a conflict with `~/ .local/share/mime/mime.cache`: removing this file will solve this conflict; or that you need to install the `shared-mime-info` package on your system.

Non-privileged users cannot start GPS Q: I have installed GPS originally as super user, and ran GPS successfully, but normal users can't.

A: You should check the permissions of the directory `$HOME/.gps` and its subdirectories, they should be owned by the user.

GPS crashes whenever I open a source editor This is usually due to font problems. Editing the file `$HOME/.gps/preferences` and changing the name of the fonts, e.g changing *Courier* by *Courier Medium*, and *Helvetica* by *Sans* should solve the problem.

GPS refuses to start the debugger If GPS cannot properly initialize the debugger (using the menu *Debug* → *Initialize*), it is usually because the underlying debugger (gdb) cannot be launched properly. To verify this, try to launch the ‘gdb’ command from a shell (i.e outside GPS). If gdb cannot be launched from a shell, it usually means that you are using a wrong version of gdb (e.g a version of gdb built for Solaris 8, but run on Solaris 2.6).

GPS is frozen during a debugging session If GPS is no longer responding while debugging an application you should first wait a little bit, since some communications between GPS and gdb can take a long time to finish. If GPS is still not responding after a few minutes, you can usually get the control back in GPS by either typing `Ctrl-C` in the shell where you’ve started GPS: this should unblock it; if it does not work, you can kill the gdb process launched by GPS using the *ps* and *kill*, or the *top* command under Unix,

and the *Task Manager* under Windows: this will terminate your debugging session, and will unblock GPS.

My Ada program fails during elaboration. How can I debug it ? If your program was compiled with GNAT, the main program is generated by the binder. This program is an ordinary Ada (or C if the *-C* switch was used) program, compiled in the usual manner, and fully debuggable provided that the *-g* switch is used on the *gnatlink* command (or *-g* is used in the *gnatmake* command itself).

The name of this package containing the main program is `b~xxx.ads/adb` where *xxx* is the name of the Ada main unit given in the *gnatbind* command, and you can edit and debug this file in the normal manner. You will see a series of calls to the elaboration routines of the packages, and you can debug these in the usual manner, just as if you were debugging code in your application.

How can I debug the Ada run-time library ?

The run time distributed in binary versions of GNAT hasn’t been compiled with debug information. Thus, it needs to be recompiled before you can actually debug it.

The simplest is to recompile your application by adding the switches *-a* and *-f* to the *gnatmake* command line. This extra step is then no longer required, assuming that you keep the generated object and ali files corresponding to the GNAT run time available.

Another possibility on Unix systems is to use the file `Makefile.adalib` that can be found in the `adalib` directory of your GNAT installation and specify e.g *-g -O2* for the *CFLAGS* switches.

The GPS main window is not displayed

If when launching GPS, nothing happens, you can try to rename the `.gps` directory (see *Files*) to start from a fresh set up.

My project have several files with the same name. How can I import it in GPS?

GPS’s projects do not allow implicit overriding of sources file, i.e. you cannot have multiple times the same file name in the project hierarchy. The reason is that GPS needs to know exactly where the file is, and cannot reliably guess which occurrence to use.

There are several solutions to handle this issue:

Put all duplicate files in the same project

There is one specific case where a project is allowed to have duplicate source files: if the list of source directories is specified explicitly. All duplicate files must be in the same project. With these conditions, there is no ambiguity for GPS and the GNAT tools which file to use, and the first file found on the source path is the one hiding all the others. GPS only shows the first file.

You can then have a scenario variable that changes the order of source directories to give visibility on one of the other duplicate files.

Use scenario variables in the project

The idea is that you define various scenarios in your project (For instance compiling in “debug” mode or “production” mode), and change the source directories depending on this setup. Such projects can be edited directly from GPS (in the project properties editor, this is the right part

of the window, as described in this documentation). On top of the project view (left part of the GPS main window), you have a combo box displayed for each of the variable, allowing a simple switch between scenarios depending on what you want to build.

Use extending projects

These projects cannot currently be created through GPS, so you will need to edit them by hand. See the GNAT user's guide for more information on extending projects.

The idea behind this approach is that you can have a local overriding of some source files from the common build/source setup (if you are working on a small part of the whole system, you may not want to have a complete copy of the code on your local machine).

GPS is very slow compared to previous versions under unix (GPS < 4.0.0)

GPS versions 4.x need the X RENDER extension when running under unix systems to perform at a reasonable speed, so you need to make sure your X server properly supports this extension.

Using the space key brings the smart completion window under Ubuntu

This is specific to the way GNOME is configured on Ubuntu distributions. To address this incompatibility, close GPS, then go to the GNOME menu *System->Preferences->Keyboard* (or launch *gnome-keyboard-properties*).

Select the *Layout* tab, click on *Layout Options*. Then click twice on *Using space key to input non-breakable space character* and then select *Usual space at any level* and then close the dialogs.

SCRIPTING API REFERENCE FOR *GPS*

This package groups all the classes and functions exported by the GNAT Programming System.

These functions are made available through various programming languages (Python and the GPS shell at the moment). The documentation in this package is mostly oriented towards Python, but it can also be used as a reference for the GPS shell

15.1 Function description

For all functions, the list of parameters is given. The first parameter will often be called “self”, and refers to the instance of the class to which the method applies. In Python, the parameter is generally put before the method’s name, as in:

```
self.method(arg1, arg2)
```

Although it could also be called as in:

```
method(self, arg1, arg2)
```

For all other parameters, their name and type are specified. An additional default value is given when the parameter is optional. If no default value is specified, the parameter is mandatory and should always be specified. The name of the parameter is relevant if you chose to use Python’s named parameters feature, as in:

```
self.method(arg1="value1", arg2="value2")
```

which makes the call slightly more readable. The method above would be defined with three parameters in this documentation (resp. “self”, “arg1” and “arg2”).

Some examples are also provides for several functions, to help clarify the use of the function.

15.2 User data in instances

A very useful feature of python is that all class instances can be associated with any number of user data fields. For example, if you create an instance of the class `GPS.EditorBuffer`, you can associate two fields “field1” and “field2” to it (the names and number are purely for demonstration purposes, and you can use your own), as in:

```
ed = GPS.EditorBuffer.get(GPS.File("a.adb"))
ed.field1 = "value1"
ed.field2 = 2
```

GPS takes great care for most classes of always returning the same python instance for a given GUI object. For instance, if you were to get another instance of `GPS.EditorBuffer` for the same file as above, you would in fact receive the same Python instance, and thus the two fields are available to you, as in:

```
ed = GPS.EditorBuffer.get(GPS.File("a.adb"))
# ed.field1 is still "value1"
```

This is a very convenient way to store your own data associated with the various objects exported by GPS. These data will cease to exist when the GPS object itself is destroyed (for instance when the editor is closed in the example above).

15.3 Hooks

In a lot of cases, you will need to connect to specific hooks exported by GPS to be aware of events happening in GPS (loading of a file, closing a file,...). These hooks and their use are described in the GPS manual (see also the `GPS.Hook` class).

Here is a small example, where the function `on_gps_started` is called when the GPS window is fully visible to the user:

```
import GPS
def on_gps_started(hook):
    pass
```

```
GPS.Hook("gps_started").add(on_gps_started)
```

The list of parameters for the hooks is described for each hook below. The first parameter is always the name of the hook, so that the same function can be used for multiple hooks if necessary.

There are two categories of hooks: the standard hooks and the action hooks. The former return nothing, the latter return a boolean indicating whether your callback was able to perform the requested action. They are used to override some of GPS's internal behavior.

15.4 Functions

`GPS.add_location_command(command)`

Add a command to the navigation buttons in the toolbar. When the user presses the back button, this command will be executed, and should put GPS in a previous state. This is for instance used while navigating in the HTML browsers to handle the back button

Parameters `command` – A string

`GPS.base_name(filename)`

Returns the base name for the given full path name

Parameters `filename` – A string

`GPS.cd(dir)`

Change the current directory to dir

Parameters `dir` – A string

`GPS.compute_xref()`

Update the cross-reference information stored in GPS. This needs to be called after major changes to the sources only, since GPS itself is able to work with partially up-to-date information

GPS.**contextual_context** ()

Returns the context at the time the contextual menu was open.

This function will only return a valid context while the menu is open, or while an action executed from that menu is executed. You can store your own data in the returned instance, so that for instance you can precompute some internal data in the filters for the contextual actions (see `<filter>` in the XML files), and reuse that precomputed data when the menu is executed. See also the documentation for “contextual_menu_open” hook.

Returns An instance of GPS.FileContext, GPS.AreaContext,...

See Also:

GPS.current_context ()

```
# Here is an example that shows how to precompute some data when we
# decide whether a menu entry should be displayed in a contextual menu,
# and reuse that data when the action executed through the menu is
# reused.
```

```
import GPS
```

```
def on_contextual_open(name):
    context = GPS.contextual_context ()
    context.private = 10
    GPS.Console().write("creating data " + `context.private` + `\n`)

def on_contextual_close(name):
    context = GPS.contextual_context ()
    GPS.Console().write("destroying data " + `context.private` + `\n`)

def my_own_filter():
    context = GPS.contextual_context ()
    context.private += 1
    GPS.Console().write("context.private=" + `context.private` + `\n`)
    return 1

def my_own_action():
    context = GPS.contextual_context ()
    GPS.Console().write("my_own_action " + `context.private` + `\n`)
```

```
GPS.parse_xml(''')
    <action name="myaction">
        <filter shell_lang="python"
            shell_cmd="contextual.my_own_filter()" />
        <shell lang="python">contextual.my_own_action()</shell>
    </action>

    <contextual action="myaction">
        <Title>Foo1</Title>
    </contextual>
    <contextual action="myaction">
        <Title>Foo2</Title>
    </contextual>
''')
```

```
GPS.Hook ("contextual_menu_open").add(on_contextual_open)
GPS.Hook ("contextual_menu_close").add(on_contextual_close)
```

```
# The following example does almost the same thing as the above, but
# without relying on the hooks to initialize the value. We set the value
```

```
# in the context the first time we need it, instead of every time the
# menu is open.

import GPS

def my_own_filter2():
    try:
        context = GPS.contextual_context()
        context.private2 += 1

    except AttributeError:
        context.private2 = 1

    GPS.Console().write("context.private2=" + `context.private2` + '\n')
    return 1

def my_own_action2():
    context = GPS.contextual_context()
    GPS.Console().write(
        "my_own_action, private2=" + `context.private2` + '\n')

GPS.parse_xml('''
<action name="myaction2">
  <filter shell_lang="python"
            shell_cmd="contextual.my_own_filter2()" />
  <shell lang="python">contextual.my_own_action2()</shell>
</action>
<contextual action="myaction2">
  <Title>Bar1</Title>
</contextual>
<contextual action="myaction2">
  <Title>Bar2</Title>
</contextual>
''')
```

GPS.**current_context**()

Returns the current context in GPS. This is the currently selected file, line, column, project,... depending on what window is currently active. From one call of this function to the next, a different instance is returned, and therefore you shouldn't store your own data in the instance, since you will not be able to recover it later on

Returns An instance of GPS.FileContext, GPS.AreaContext,...

See Also:

GPS.Editor.get_line()

GPS.MDI.current: () Access the current window

GPS.contextual_context()

GPS.**delete**(name)

Delete file/directory name from the file system

Parameters name – A string

GPS.**dir**(pattern='')

list files following pattern (all files by default)

Parameters pattern – A string

Returns A list of strings

GPS.**dir_name** (*filename*)

Returns the directory name for the given full path name

Parameters **filename** – A string

GPS.**dump** (*string*, *add_lf=False*)

Dump string to a temporary file. Return the name of the file. If *add_lf* is TRUE, append a line feed at end of file

Parameters

- **string** – A string
- **add_lf** – A boolean

Returns A string, the name of the output file

GPS.**dump_file** (*text*, *filename*)

Writes text to filename on the disk. This is mostly intended for poor shells like the GPS shell which do not have better solutions. In python, it is recommended to use python's own mechanisms

Parameters

- **text** – A string
- **filename** – A string

GPS.**exec_in_console** (*noname*)

This function is specific to python. It executes the string given in argument in the context of the GPS Python console. If you use the standard python's `exec()` function instead, the latter will only modify the current context, which generally will have no impact on the GPS console itself.

Parameters **noname** – A string

```
# Import a new module transparently in the console, so that users can
# immediately use it
GPS.exec_in_console("import time")
```

GPS.**execute_action** (*action*, **args*)

Execute one of the actions defined in GPS. Such actions are either predefined by GPS or defined by the users through customization files. See the GPS documentation for more information on how to create new actions. GPS will wait until the command completes to return the control to the caller, whether you execute a shell command, or an external process.

The action's name can start with a '/', and be a full menu path. As a result, the menu itself will be executed, just as if the user had pressed it.

The extra arguments must be strings, and are passed to the action, which can use them through \$1, \$2,...

The list of existing actions can be found through the Edit->Actions menu.

The action will not be executed if the current context is not appropriate for this action.

Parameters

- **action** – Name of the action to execute
- **args** – Any number of string parameters

See Also:

```
GPS.execute_asynchronous_action()

GPS.execute_action(action="Split vertically")
# will split the current window vertically
```

GPS.**execute_asynchronous_action** (*action*, **args*)

This command is similar to GPS.execute_action. However, commands that execute external applications or menus are executed asynchronously: GPS.execute_asynchronous_action will immediately return, although the external application might not have completed its execution

Parameters

- **action** – Name of the action to execute
- **args** – Any number of string parameters

See Also:

GPS.execute_action()

GPS.**exit** (*force=False*, *status='0'*)

Exit GPS, asking for confirmation if any file is currently modified and unsaved. If force is True, no check is done.

Status is the exit status to return to the calling shell. 0 will generally mean success on all architectures.

Parameters

- **force** – A boolean
- **status** – An integer

GPS.**freeze_prefs** ()

Prevents the signal “preferences_changed” from being emitted. One call to thaw_prefs should be made to unfreeze.

Freezing/thawing this signal is useful when about to modify a large number of preferences in one batch.

See Also:

GPS.thaw_prefs()

GPS.**freeze_xref** ()

Forces GPS to use the cross-reference information it already has in memory. GPS will no longer check on the disk whether more recent information is available. This can provide a significant speedup in complex scripts or scripts that need to analyze the cross-reference information for lots of files. In such cases, the script should generally call GPS.Project.update_xref to first load all the required information in memory.

You need to explicitly call GPS.thaw_xref to go back to the default GPS behavior. Note the use of the “finally” exception handling in the following example, which ensures that even if there is some unexpected exception, the script always restores properly the default behavior.

See Also:

GPS.Project.update_xref()

GPS.thaw_xref()

try:

```
GPS.Project.root().update_xref(recursive=True)
GPS.freeze_xref()
... complex computation
```

finally:

```
GPS.thaw_xref()
```

GPS.**get_build_mode** ()

Return the name of the current build mode. Return an empty string if no mode is registered.

`GPS.get_build_output(target_name, shadow, background, as_string)`
Return the result of the last compilation command

Parameters

- **target_name** – (optional) a string
- **shadow** – (optional) a Boolean, indicating whether we want the output of shadow builds
- **background** – (optional) a Boolean, indicating whether we want the output of background builds
- **as_string** – (optional) a Boolean, indicating whether the output should be returned as a single string. By default the output is returned as a list in script languages that support it.

Returns A string or list, the output of the latest build for the corresponding target.

See Also:

`GPS.File.make()`

`GPS.File.compile()`

`GPS.get_busy()`
Return the “busy” state

See Also:

`GPS.set_busy()`

`GPS.unset_busy()`

`GPS.get_home_dir()`
Return the directory that contains the user-specific files. This directory always ends with a directory separator

Returns The user’s GPS directory

See Also:

`GPS.get_system_dir()`

```
log = GPS.get_home_dir() + "log"
# will compute the name of the log file generated by GPS
```

`GPS.get_system_dir()`
Return the installation directory for GPS. This directory always ends with a directory separator

Returns The install directory for GPS

See Also:

`GPS.get_home_dir()`

```
html = GPS.get_system_dir() + "share/doc/gps/html/gps.html"
# will compute the location of GPS's documentation
```

`GPS.get_tmp_dir()`
Return the directory where gps creates temporary files. This directory always ends with a directory separator

Returns The install directory for GPS

`GPS.insmod(shared_lib, module)`
Dynamically register a new module, reading its code from `shared_lib`.

The library must define the following two symbols:

- `_init`: This is called by GPS to initialize the library itself

- `__register_module`: This is called to do the actual module registration, and should call the `Register_Module` function in the GPS source code

This is work in progress, and not fully supported on all systems.

Parameters

- **shared_lib** – Library containing the code of the module
- **module** – Name of the module

See Also:

`GPS.lsmod()`

`GPS.is_server_local(server)`

Tell if the specified server is the local machine.

Parameters `server` – The server. Possible values are “Build_Server”, “Debug_Server”, “Execution_Server” and “Tools_Server”.

Returns A boolean

`GPS.last_command()`

This function returns the name of the last action executed by GPS. This name is not ultra-precise: it will be accurate only when the action is executed through a key binding. Otherwise, an empty string is returned. However, the intent here is for a command to be able to check whether it is called multiple times in a row. For this, this command will return the command set by `GPS.set_last_command()` if it was set.

Returns A string

See Also:

`GPS.set_last_command()`

```
def kill_line():
    '''Emulates Emacs behavior: when called multiple times, the cut line must be
       appended to the previously cut one.'''

    # The name of the command below is unknown to GPS. This is just a
    # string we use in this implementation to detect multiple consecutive
    # calls to this function. Note that this works whether the function is
    # called from the same key binding or not, and from the same GPS action
    # or not

    append = GPS.last_command() == "my-kill-line":
    GPS.set_last_command("my-kill-line")
```

`GPS.lookup_actions()`

This command returns the list of all known GPS actions. This doesn't include menu names. All actions are lower-cased, but the order in the list is not significant.

Returns A list of strings

See Also:

`GPS.lookup_actions_from_key()`

`GPS.lookup_actions_from_key(key)`

Given a key binding, for instance “control-x control-b”, this function returns the list of actions that could be executed. Not all actions would be executed, though, since only the ones for which the filter matches are executed. The name of the actions is always in lower cases.

Parameters `key` – A string

Returns A list of strings

See Also:

`GPS.lookup_actions()`

`GPS.ls(pattern='')`
list files following pattern (all files by default)

Parameters `pattern` – A string

Returns A list of strings

`GPS.lsmod()`

Return the list of modules that are currently registered in GPS. Each facility in GPS is provided in a separate module, so that users can choose whether to activate specific modules or not. Some modules can also be dynamically loaded

Returns List of strings

See Also:

`GPS.insmod()`

`GPS.parse_xml(xml)`

Load an XML customization string. This string should contain one or more toplevel tags similar to what is normally found in custom files, such as `<key>`, `<alias>`, `<action>`,...

Optionally you can also pass the full contents of an XML file, starting from the `<?xml?>` header.

Parameters `xml` – The XML string to parse

```
GPS.parse_xml(
    '''<action name="A"><shell>my_action</shell></action>
    <menu action="A"><title>/Edit/A</title></menu>''')
```

Adds a new menu `in` GPS, which executes the command `my_action`

`GPS.pwd()`

Print name of current/working directory

Returns A string

This command will have the same return value as the standard Python command `os.getcwd()`. The current directory can also be changed through a call to `os.chdir("dir")`.

`GPS.repeat_next(count)`

This action will execute the next one `<count>` times.

Parameters `count` – An integer

`GPS.reset_xref_db()`

Empties the internal xref database for GPS. This is rarely useful, unless you want to force GPS to reload everything.

`GPS.save_persistent_properties()`

Forces an immediate save of the persistent properties that GPS maintains for files and projects (for instance the text encoding, the programming language, the debugger breakpoints,...).

You do not have to call this subprogram explicitly in general, since this is done automatically by GPS on exit.

`GPS.set_build_mode(mode='')`

Set the current build mode. If specified mode is not a registered mode, do nothing.

Parameters `mode` – Name of the mode to set

GPS.**set_busy**()

Activate the “busy” state in GPS by animating the GPS icon. This command can be called recursively, and GPS.unset_busy should be called a corresponding number of time to stop the animation.

See Also:

GPS.unset_busy()

GPS.get_busy()

GPS.**set_last_command**(*command*)

This function overrides the name of the last command executed by GPS. This new name will be the one returned by GPS.last_command() until the user performs a different action. Thus, multiple calls of the same action in a row will always return the value of the command parameter. See the example in GPS.last_command()

Parameters *command* – A string

See Also:

GPS.last_command()

GPS.**supported_languages**()

Return the list of languages for which GPS has special handling. Any file can be open in GPS, but some extensions are recognized specially by GPS to provide syntax highlighting, cross-references, or other special handling. See the GPS documentation on how to add support for new languages in GPS.

The returned list is sorted alphabetically, and the name of the language has been normalized (start with an upper case, and use lowercases for the rest except after an underscore character)

Returns List of strings

```
GPS.supported_languages()[0]
```

```
=> return the name of the first supported language
```

GPS.**thaw_prefs**()

Re-enables the emission of the “preferences_changed”

See Also:

GPS.freeze_prefs()

GPS.**thaw_xref**()

See GPS.freeze_xref for more information

See Also:

GPS.freeze_xref()

GPS.**unset_busy**()

Reset the “busy” state

See Also:

GPS.set_busy()

GPS.get_busy()

GPS.**version**()

Return GPS version as a string.

Returns A string

GPS.**xref_db**()

Returns the location of the xref database. This is a sqlite database that is created by GPS when it parses the .ali files generated by the compiler. Its location depends on both the location of the root project (the database is in

its object directory by default), and its optional IDE'Xref_Database attribute which can be used to specify an alternate location.

Returns a string

`GPS.xref_frozen()`

Return true if the xref database is frozen.

See Also:

`GPS.freeze_xref()`

15.5 Classes

15.5.1 `GPS.Action`

class `GPS.Action` (*name*)

This class gives access to the interactive commands in GPS. These are the commands to which the user can bind a key shortcut, or for which we can create a menu. Another way to manipulate those commands is through the XML tag `<action>`, but it might be more convenient to use python since you do not have to qualify the function name as a result

`__init__` (*name*)

Creates a new instance of Action. This is bound with either an existing action, or with an action that will be created through `GPS.Action.create()`. The name of the action can either be a simple name, or a path name to reference a menu, as in `/Edit/Copy` for instance.

Parameters *name* – A string

contextual (*path*, *ref*='', *add_before*=True)

Create a new contextual menu associated with the command. This function is somewhat a duplicate of `GPS.Contextual.create`, but with one major difference: the callback for the action is a python function that takes no argument, whereas the callback for `GPS.Contextual` receives one argument.

Parameters

- **path** – A string
- **ref** – A string
- **add_before** – A boolean

create (*on_activate*, *filter*='', *category*='General', *description*='')

Export the function `on_activate` and make it interactive so that users can bind keys and menus to it. The function should not require any argument, since it will be called with none.

The package `gps_utils.py` provides a somewhat more convenient python interface to make function interactives (see `gps_utils.interactive`).

Parameters

- **on_activate** – A subprogram
- **filter** – A string or subprogram This is either the name of a predefined filter (a string), or a subprogram that receives the context as a parameter, and should return True if the command can be executed within that context. This is used to disable menu items when they are not available.
- **category** – A string The category of the command in the `/Edit/Key Shortcuts` dialog.

- **description** – A string The description of the command that appears in that dialog. If you are using python, a convenient value is `on_activate.__doc__`, which avoids duplicating the comment.

execute_if_possible ()

Execute the action if its filter matches the current context. If it could be executed, True is returned, otherwise False is returned.

Returns A boolean

key (*key*)

Associate a default key binding with the action. This will be ignored if the user has defined his own key binding. Possible values for key can be experimented with by using the /Edit/Key Shortcuts dialog

Parameters **key** – A string

menu (*path*, *ref*='', *add_before*=True)

Create a new menu associated with the command. This function is somewhat a duplicate of `GPS.Menu.create()`, but with one major difference: the callback for the action is a python function that takes no argument, whereas the callback for `GPS.Menu()` receives one argument.

Parameters

- **path** – A string
- **ref** – A string
- **add_before** – A boolean

15.5.2 GPS.Activities

class `GPS.Activities` (*name*)

General interface to version control activities systems

__init__ (*name*)

Creates a new activity and returns its instance

Parameters **name** – Activity's name to be given to this instance

```
a=GPS.Activities("Fix loading order")
print a.id()
```

add_file (*file*)

Adds the file into the activity

Parameters **file** – An instance of `GPS.File`

commit ()

Commit the activity

files ()

Returns the activity's files list

Returns A list of files

static from_file (*file*)

Returns the activity containing the given file

Parameters **file** – An instance of `GPS.File`

Returns An instance of `GPS.Activities`

static get (*id*)
Returns the activity given its id

Parameters *id* – The unique activity’s id

Returns An instance of `GPS.Activities()`

See Also:

`GPS.Activities.list()`

group_commit ()
Returns true if the activity will be commit atomically

Returns A boolean

has_log ()
Returns true if the activity has a log present

Returns A boolean

id ()
Returns the activity’s unique id

Returns A string

is_closed ()
Returns true if the activity is closed

Returns A boolean

static list ()
Returns the list of all activities’s id

Returns A list of all activities’s id defined

log ()
Returns the activity’s log content

Returns A string

log_file ()
Returns the activity’s log file

Returns A file

name ()
Returns the activity’s name :return: A string

remove_file (*file*)
Removes the file into the activity

Parameters *file* – An instance of `GPS.File`

set_closed (*status*)
Set the activity’s closed status

Parameters *status* – A boolean

toggle_group_commit ()
Change the activity’s group commit status

vcs ()
Returns the activity’s VCS name

Returns A string

15.5.3 `GPS.Alias`

class `GPS.Alias`

This class represents a GPS Alias, that is, a code template to be expanded in an editor. This class allows you to manipulate them programmatically.

static `get` (*name*)

Get the alias instance corresponding to name

15.5.4 `GPS.AreaContext`

class `GPS.AreaContext`

Represents a context that contains file information and a range of lines currently selected

See Also:

`GPS.AreaContext.__init__()`

`__init__()`

Dummy function, whose goal is to prevent user-creation of a `GPS.AreaContext` instance. Such instances can only be created internally by `GPS`

`end_line()`

Return the last selected line in the context

Returns An integer

`start_line()`

Return the first selected line in the context

Returns An integer

15.5.5 `GPS.Bookmark`

class `GPS.Bookmark`

This class provides access to the bookmarks of `GPS`. These are special types of markers that are saved across sessions, and can be used to save a context within `GPS`. They are generally associated with a specific location in an editor, but can also be used to location special boxes in a graphical browser for instance.

`__init__()`

This function prevents the creation of a bookmark instance directly. You must use `GPS.Bookmark.get()` instead, which will always return the same instance for a given bookmark, thus allowing you to save your own custom data with the bookmark

See Also:

`GPS.Bookmark.get()`

static `create` (*name*)

This function creates a new bookmark at the current location in `GPS`. If the current window is an editor, it creates a bookmark that will save the exact line and column, so that the user can go back to them easily. Name is the string that appears in the bookmarks window, and that can be used later to query the same instance using `GPS.Bookmark.get()`. This function emits the hook `bookmark_added`.

Parameters *name* – A string

Returns An instance of `GPS.Bookmark`

See Also:

```
GPS.Bookmark.get ()

GPS.MDI.get ("file.adb").raise_window ()
bm = GPS.Bookmark.create ("name")
```

delete ()

Delete an existing bookmark. This emits the hook `bookmark_removed`

static get (name)

This function retrieves a bookmark by its name. If no such bookmark exists, an exception is raised. The same instance of `GPS.Bookmark` is always return for a given bookmark, so that you can store your own user data within the instance. Note however that this custom data will not be automatically preserved across GPS sessions, so you might want to save all your data when GPS exits

Parameters `name` – A string

Returns An instance of `GPS.Bookmark`

See Also:

```
GPS.Bookmark.create ()

GPS.Bookmark.get ("name").my_own_field = "GPS"
print GPS.Bookmark.get ("name").my_own_field # prints "GPS"
```

goto ()

Change the current context in GPS so that it matches the one saved in the bookmark. In particular, if the bookmark is inside an editor, this editor is raised, and the cursor moved to the correct line and column. You cannot query directly the line and column from the bookmark, since these might not exist, for instance when the editor points inside a browser.

static list ()

Return the list of all existing bookmarks

Returns A list of `GPS.Bookmark` instances

```
# The following command returns a list with the name of all
# existing purposes
names = [bm.name () for bm in GPS.Bookmark.list ()]
```

name ()

Return the current name of the bookmark. It might not be the same one that was used to create or get the bookmark, since the user might have used the bookmarks view to rename it

Returns A string

rename (name)

Rename an existing bookmark. This updates the bookmarks view automatically, and emits the hooks `bookmark_removed` and `bookmark_added`

Parameters `name` – A string

15.5.6 GPS.BuildTarget

class `GPS.BuildTarget` (*name*)

This class provides an interface to the GPS build targets. Build targets can be configured through XML or through the Target Configuration dialog.

`__init__(name)`

Initializes a new instance of the class `BuildTarget`. Name must correspond to an existing target.

Parameters `name` – Name of the target associated with this instance

```
compile_file_target=GPS.BuildTarget("Compile File")
compile_file_target.execute()
```

`clone(new_name, new_category)`

Clone the target to a new target. All the properties of the new target are copied from the target. Any graphical element corresponding to this new target is created.

Parameters

- `new_name` – The name of the new target
- `new_category` – The category in which to place the new target

`execute(main_name='', file='', force=False, extra_args='', build_mode='', synchronous=True, directory='', quiet=False)`

Launch the build target.

Parameters

- `main_name` – A String Indicates the base name of the main source to build, if this target acts on a main file.
- `file` – A `GPS.File` Indicates the file to build if this targets acts on a file.
- `force` – A Boolean If True, this means that the target should be launched directly, even if its parameters indicate that it should be launched through an intermediary dialog.
- `extra_args` – A String or a list of strings any extra parameters to pass to the command line. When a single string is passed, it is split into multiple arguments.
- `build_mode` – A String Indicates build mode to be used for build.
- `synchronous` – A Boolean if False, build target is launched asynchronously. `compilation_finished` hook will be called when build target execution is completed.
- `directory` – A String
- `quiet` – A Boolean

`hide()`

Hide target from menus and toolbar.

`remove()`

Remove target from the list of known targets. Any graphical element corresponding to this target is also removed.

`show()`

Show target in menus and toolbar where it was before hiding.

15.5.7 `GPS.Button`

`class GPS.Button(id, label, on_click)`

This class represents a button that can be pressed to trigger various actions

See Also:

```
GPS.Button.__init__()
```

`__init__` (*id*, *label*, *on_click*)

Initializes a new button. When the button is pressed by the user, `on_click` is called with the a single parameter, `self`.

Parameters

- **id** – A string, a unique identifier for the button
- **label** – A string, the text that appears on the button
- **on_click** – A subprogram, see the GPS documentation

```
def on_click (button):
    print "Button pressed"
button = GPS.Button ("my_id", label="Press me", on_click=on_click)
GPS.Toolbar().append (button)
```

`set_text` (*label*)

Change the text that appears on the button

Parameters *label* – A string

15.5.8 GPS.Clipboard

class `GPS.Clipboard`

This class provides an interface to the GPS clipboard. This clipboard contains the previous selections that were copied or cut from a text editor. Several older selections are also saved so that they can be pasted later on

static `contents` ()

This function returns the contents of the clipboard. Each item in the list corresponds to a past selection, the one at position 0 being the most recent. If you want to paste text in a buffer, you should paste the text at position `GPS.Clipboard.current` () rather than the first in the list

Returns A list of strings

static `copy` (*text*, *append=False*)

Copies a given static text into the clipboard. It is better in general to use `GPS.EditorBuffer.copy` (), but it might happen that you need to append text that doesn't exist in the buffer.

Parameters

- **text** – A string
- **append** – A boolean

See Also:

`GPS.EditorBuffer.copy` ()

static `current` ()

This function returns the index, in `GPS.Clipboard.contents` (), of the text that was last pasted by the user. If you were to select the menu /Edit/Paste, that would be the text pasted by GPS. If you select /Edit/Paste Previous, `current` will be incremented by 1, and the next selection in the clipboard will be pasted

Returns An integer

static `merge` (*index1*, *index2*)

This function merges two levels of the clipboard, so that the one at index `index1` now contains the concatenation of both. The one at `index2` is removed.

Parameters

- **index1** – A null or positive integer

- **index2** – A null or positive integer

15.5.9 GPS.CodeAnalysis

class `GPS.CodeAnalysis`

This class is a toolset that allows to handle CodeAnalysis instances.

`__init__()`

Raises an exception to prevent users from creating new instances.

`add_all_gcov_project_info()`

Adds coverage information of every source files referenced in the current project loaded in GPS, and every imported projects.

See Also:

`GPS.CodeAnalysis.add_gcov_project_info()`

`GPS.CodeAnalysis.add_gcov_file_info()`

`add_gcov_file_info(src, cov)`

Adds coverage information provided by a .gcov file parsing. The data is read from the cov parameter, that should have been created from the specified src file.

Parameters

- **src** – A `GPS.File` instance
- **cov** – A `GPS.File` instance

See Also:

`GPS.CodeAnalysis.add_all_gcov_project_info()`

`GPS.CodeAnalysis.add_gcov_project_info()`

```
a = GPS.CodeAnalysis.get ("Coverage Report")
```

```
a.add_gcov_file_info (src=GPS.File ("source_file.adb"), cov=GPS.File ("source_file.adb.gcov"))
```

`add_gcov_project_info(prj)`

Adds coverage information of every source files referenced in the given ‘prj’ gnat project file (.gpr).

Parameters **prj** – A `GPS.File` instance

See Also:

`GPS.CodeAnalysis.add_all_gcov_project_info()`

`GPS.CodeAnalysis.add_gcov_file_info()`

`clear()`

Removes all code analysis information from memory.

`dump_to_file(xml)`

Create an xml-formated file that contains a representation of the given code analysis.

Parameters **xml** – A `GPS.File` instance

See Also:

`GPS.CodeAnalysis.load_from_file()`

```
a = GPS.CodeAnalysis.get ("Coverage")
```

```
a.add_all_gcov_project_info ()
```

```
a.dump_to_file (xml=GPS.File ("new_file.xml"))
```

static get (*name*)

Creates an empty code analysis data structure. Data can be put in this structure by using one of the primitive operations.

Parameters *name* – The name of the code analysis data structure to get or create

Returns An instance of `GPS.CodeAnalysis` associated to a code analysis data structure in `GPS`.

```
a = GPS.CodeAnalysis.get ("Coverage")
a.add_all_gcov_project_info ()
a.show_coverage_information ()
```

hide_coverage_information ()

Removes from the Locations view any listed coverage locations, and remove from the source editors their annotation column if any.

See Also:

`GPS.CodeAnalysis.show_coverage_information()`

load_from_file (*xml*)

Replace the current coverage information in memory with the given xml-formated file one.

Parameters *xml* – A `GPS.File` instance

See Also:

`GPS.CodeAnalysis.dump_to_file()`

```
a = GPS.CodeAnalysis.get ("Coverage")
a.add_all_gcov_project_info ()
a.dump_to_file (xml=GPS.File ("new_file.xml"))
a.clear ()
a.load_from_file (xml=GPS.File ("new_file.xml"))
```

show_analysis_report ()

Displays the data stored in the `CodeAnalysis` instance into a new MDI window. This window contains a tree view that can be interactively manipulated to analyze the results of the code analysis (Coverage, ...).

show_coverage_information ()

Lists in the Locations view the lines that are not covered in the files loaded in the `CodeAnalysis` instance. The lines are also highlighted in the corresponding source file editors, and an annotation column is added to the source editors.

See Also:

`GPS.CodeAnalysis.hide_coverage_information()`

15.5.10 `GPS.Codefix`

class `GPS.Codefix` (*category*)

This class gives access to `GPS`'s features for automatically fixing compilation errors

See Also:

`GPS.CodefixError()`

`GPS.Codefix.__init__()`

`__init__` (*category*)

Return the instance of `codefix` associated with the given `category`

Parameters `category` – A string

error_at (*file, line, column, message=''*)

Return a specific error at a given location. If message is null, then the first matching error will be taken. None is returned if no such fixable error exists.

Parameters

- **file** – The file where the error is
- **line** – The line where the error is
- **column** – The column where the error is
- **message** – The message of the error

Returns An instance of `GPS.CodefixError`

errors ()

List the fixable errors in that session

Returns A list of instances of `GPS.CodefixError`

static parse (*category, output, regexp='', file_index=-1, line_index=-1, column_index=-1, style_index=-1, warning_index=-1*)

Parse the output of a tool, and suggests auto-fix possibilities whenever possible. This adds small icons in the location window, so that the user can click on it to fix compilation errors. You should call `Locations.parse` with the same output prior to calling this command.

The regular expression specifies how locations are recognized. By default, it matches `file:line:column`. The various indexes indicate the index of the opening parenthesis that contains the relevant information in the regular expression. Set it to 0 if that information is not available.

Access the various suggested fixes through the methods of the `Codefix` class

Parameters

- **category** – A string
- **output** – A string
- **regexp** – A string
- **file_index** – An integer
- **line_index** – An integer
- **column_index** – An integer
- **style_index** – An integer
- **warning_index** – An integer

See Also:

`GPS.Editor.register_highlighting()`

static sessions ()

List all the existing `Codefix` sessions. The returned values can all be used to create a new instance of `Codefix` through its constructor.

Returns A list of strings

```
# After a compilation failure:  
>>> GPS.Codefix.sessions()  
=> ['Builder results']
```

15.5.11 GPS.CodefixError

class `GPS.CodefixError` (*codefix, file, message=''*)

This class represents a fixable error in the compilation output

See Also:

`GPS.Codefix()`

`GPS.CodefixError.__init__()`

`__init__` (*codefix, file, message=''*)

Describe a new fixable error. If the message is not specified, the first error at that location is returned

Parameters

- **codefix** – An instance of `GPS.Codefix`
- **file** – An instance of `GPS.FileLocation`
- **message** – A string

fix (*choice='0'*)

Fix the error, using one of the possible fixes. The index given in parameter is the index in the list returned by `possible_fixes`. By default, the first choice is taken. Choices start at index 0.

Parameters *choice* – The index of the fix to apply, see output of `GPS.CodefixError.possible_fixes()`

```
for err in GPS.Codefix ("Builder results").errors():
    print err.fix()
```

```
# will automatically fix all fixable errors in the last compilation
# output
```

location ()

Return the location of the error

Returns An instance of `GPS.FileLocation`

message ()

Return the error message, as issues by the tool

Returns A string

possible_fixes ()

List the possible fixes for the specific error

Returns A list of strings

```
for err in GPS.Codefix ("Builder results").errors():
    print err.possible_fixes()
```

15.5.12 GPS.Combo

class `GPS.Combo` (*id, label='', on_changed=None*)

This class represents a combo box, ie a text entry widget with a number of predefined possible values. The user can interactively select one of multiple values through this widget

See Also:

`GPS.Toolbar`

`GPS.Combo.__init__()`

`__init__(id, label='', on_changed=None)`

Create a new combo. The combo will graphically be preceded by some text if label was specified. `on_changed` will be called every time the user selects a new value for the combo box. Its parameters are the following:

- \$1 = The instance of `GPS.Combo` (self)
- \$2 = The newly selected text (a string)

Parameters

- **id** – A string, the name of the combo to create
- **label** – A string, the label to add next to the entry
- **on_changed** – A subprogram, see the GPS documentaion on Subprogram parameters

See Also:

`GPS.Toolbar.append()`

`GPS.Toolbar.ge()`

`add(choice, on_selected=None)`

Add a choice to specified entry, `on_selected` will be executed whenever this choice is selected. It is called with the following parameters:

- \$1 = The instance of `GPS.Combo` (self)
- \$2 = The newly selected text (a string)

Parameters

- **choice** – A string
- **on_selected** – A subprogram, see the GPS documentation on Subprogram parameters

`clear()`

Remove all choices from specified entry

`get_text()`

Return the current selection in specified entry

Returns A string

`remove(choice)`

Remove a choice from specified entry

Parameters **choice** – A string

See Also:

`GPS.Combo.clear()`

`set_text(choice)`

Set the current selection in specified entry

Parameters **choice** – A string

15.5.13 GPS.Command

class GPS.Command

Interface to GPS command. This class is abstract, and shall be subclassed.

static get (*name*)

Return the list of commands of the name given in parameter, scheduled or running in the task manager

Parameters *name* – A string

Returns a list of GPS.Command

get_result ()

Return the result of the command, if any. Must be overridden by children

interrupt ()

Interrupt the current command

static list ()

Return the list of commands scheduled or running in the task manager

Returns a list of GPS.Command

name ()

Return The name of the command

progress ()

Return a list representing the current progress of the command. If current = total, then the command is finished.

Returns a list [current, total]

15.5.14 GPS.CommandWindow

class GPS.CommandWindow (*prompt=''*, *global_window=False*, *on_changed=None*, *on_activate=None*, *on_cancel=None*, *on_key=None*, *close_on_activate=True*)

This class gives access to a command-line window that pops up on the screen. This window is short-lived (in fact there can be only one such window at any given time), and any key press is redirected to that window. As a result, it should be used to interactively query a parameter for an action, for instance.

Among other things, it is used in the implementation of the interactive search facility, where each key pressed should be added to the search pattern instead of to the editor.

```
class Isearch(CommandWindow):
    def __init__(self):
        CommandWindow.__init__(
            self, prompt="Pattern",
            on_key=self.on_key,
            on_changed=self.on_changed)

    def on_key(self, input, key, cursor_pos):
        if key == "control-w":
            .... # Copy current word from editor into the window
            self.write(input[:cursor_pos + 1] + "FOO" + input[cursor_pos + 1:])
            return True ## No further processing needed
        return False

    def on_changed(self, input, cursor_pos):
        ## Search for next occurrence of input in buffer
        ....
```

`__init__` (*prompt*='', *global_window*=False, *on_changed*=None, *on_activate*=None, *on_cancel*=None, *on_key*=None, *close_on_activate*=True)

This function initializes an instance of a command window. An exception is raised if such a window is already active in GPS. Otherwise, the new window is popped up on the screen. Its location depends on the `global_window` parameter: if true, the command window is displayed at the bottom of the GPS window and occupies its whole width. If false, it is displayed at the bottom of the currently selected window.

The prompt is the short string displayed just before the command line itself. Its goal is to indicate to the user what he is entering.

The last four parameters are callbacks:

- `on_changed` is called when the user has entered one or more new characters in the command line. This function is given two parameters: the current input string, and the last cursor position in this string. See the example above on how to get the part of the input before and after the cursor.
- `on_activate` is called when the user has pressed enter. The command window has already been closed at that point if `close_on_activate` is True, and the focus given back to the initial MDI window that had it. This callback is given a single parameter, the final input string
- `on_cancel` is called when the user has pressed a key that closed the dialog, for instance Esc. It is given a single parameter, the final input string. This callback is also called when you explicitly destroy the window yourself by calling `self.destroy()`.
- `on_key` is called when the user has pressed a new key on his keyboard, but before the corresponding character has been added to the command line. This can be used to filter out some characters, or provide special behavior for some key combination (see the example above). It is given three parameters, the current input string, the key that was pressed, and the current cursor position.

param prompt A string
param global_window A boolean
param on_changed A subprogram
param on_activate A subprogram
param on_cancel A subprogram
param on_key A subprogram
param close_on_activate A boolean

read ()

This function returns the current contents of the command window

Returns A string

set_background (*color*='')

Change the background color of the command window. In most cases, this can be used to make the command window more obvious, or to point out errors by changing the color. If the color parameter is not specified, the color reverts to its default

Parameters color – A string

set_prompt (*prompt*)

Changes the prompt that is displayed before the text field

Parameters prompt – A string

write (*text*, *cursor*=-1)

This function replaces the current content of the command line. As a result, you should make sure to preserve the character you want, as in the `on_key` callback in the example above. Calling this function will

also result in several calls to the `on_changed` callback, one of them with an empty string (since `gtk` first deletes the contents and then writes the new contents).

The `cursor` parameter can be used to specify where the cursor should be left after the insertion. `-1` indicates the end of the string.

param text A string

param cursor An integer

15.5.15 GPS.Console

```
class GPS.Console(name, force=False, on_input=None, on_destroy=None, accept_input=True,
                 on_resize=None, on_interrupt=None, on_completion=None, on_key='', manage_prompt=True, ansi=False)
```

This class is used to create and interact with the interactive consoles in GPS. It can be used to redirect the output of scripts to various consoles in GPS, or to get input from the user has needed.

See Also:

`GPS.Process`

`GPS.Console.__init__()`

```
# The following example shows how to redirect the output of a script to
# a new console in GPS:
```

```
console = GPS.Console("My_Script")
console.write("Hello world") # Explicit redirection
```

```
# The usual python's standard output can also be redirected to this
# console:
```

```
sys.stdout = GPS.Console("My_Script")
print "Hello world, too" # Implicit redirection
sys.stdout = GPS.Console("Python") # Back to python's console
sys.stdout = GPS.Console() # Or back to GPS's console
```

```
# The following example shows an integration between the GPS.Console
# and GPS.Process classes, so that a window containing a shell can be
# added to GPS.
```

```
# Note that this class is in fact available directly through "from
# gps_utils.console_process import Console_Process" if you need it in
# your own scripts.
```

```
import GPS
```

```
class Console_Process(GPS.Console, GPS.Process):
```

```
    def on_output(self, matched, unmatched):
        self.write(unmatched + matched)
```

```
    def on_exit(self, status, unmatched_output):
```

```
        try:
            self.destroy()
        except:
            pass # Might already have been destroyed
```

```
    def on_input(self, input):
```

```
        self.send(input)
```

```
def on_destroy(self):
    self.kill() # Will call on_exit

def __init__(self, process, args=""):
    GPS.Console.__init__(
        self, process,
        on_input=Console_Process.on_input,
        on_destroy=Console_Process.on_destroy,
        force=True)
    GPS.Process.__init__(
        self, process + ' ' + args, "+",
        on_exit=Console_Process.on_exit,
        on_match=Console_Process.on_output)

bash = Console_Process("/bin/sh", "-i")

__init__(name, force=False, on_input=None, on_destroy=None, accept_input=True,
         on_resize=None, on_interrupt=None, on_completion=None, on_key=' ', man-
         age_prompt=True, ansi=False)
```

Create a new instance of `GPS.Console`. `GPS` will try to reuse any existing console with the same name. If none exists yet, or the parameter `force` is set to `True`, then `GPS` will create a new console.

You cannot create the Python and Shell consoles through this call. If you do, an exception is raised. Instead, use `GPS.execute_action` (“/Tools/Consoles/Python”), and then get a handle on the console through `GPS.Console`. This is because these two consoles are tightly associated with each of the scripting languages.

If `GPS` reuses an existing console, `on_input` overrides the callback that was already set on the console, whereas `on_destroy` will be called in addition to the one that was already set on the console.

If this is not the desired behavior, you can also call `destroy()` on the console, and call the constructor again.

- The subprogram `on_input` is called whenever the user has entered a new command in the console and pressed <enter> to execute it. It is called with the following parameters:

- \$1: The instance of the `GPS.Console`

- \$2: The command to execute

See the subprogram `GPS.Console.set_prompt_regexp` for proper handling of input in the console.

- The subprogram `on_destroy` is called whenever the user closes the console. It is called with a single parameter:

- \$1: The instance of the `GPS.Console`

- The subprogram `on_completion` is called whenever the user presses tab in the console. It is called with a single parameter:

- \$1: The instance of the `GPS.Console`

The default implementation is to insert a tab character, but you could choose to add some user input through `GPS.Console.add_input` for instance.

- The subprogram `on_resize` is called whenever the console is resized by the user. It is passed three parameters:

- \$1 is the instance of `GPS.Console`

- \$2 is the number of visible rows in the console,

- and \$3 is the number of visible columns.

This is mostly useful when a process is running in the console, in which case you can use `GPS.Process.set_size` to let the process know about the size. Note that the size passed to this callback is conservative: since all characters might not have the same size, GPS tries to compute the maximal number of visible characters and pass this to the callback, but the exact number of characters might depend on the font.

- The subprogram `on_interrupt` is called when the user presses control-c in the console. It receives a single parameter, which is the instance of `GPS.Console`. By default a control-c is handled by GPS itself and will kill the last process that was started.

As described above, GPS provides a high-level handling of consoles, where it manages histories, completion, command line editing and execution on its own through the callbacks described above. This is in general a good thing and provides advanced functionalities to some programs that lack them. However, there are cases where this gets in the way. For instance, if you want to run a Unix shell or a program that manipulates the console by moving the cursor around on its own, the high-level handling of GPS gets in the way. In such a case, the following parameters can be used: `on_key`, `manage_prompt` and `ansi`.

- `ansi` should be set to true if GPS should emulate an ANSI terminal. These are terminals that understand certain escape sequences that applications sent to move the cursor to specific positions on screen or to change the color and attributes of text.
- `manage_prompt` should be set to False to disable GPS's handling of prompt. In general, this is incompatible with using the `on_input` callback, since GPS no longer distinguishes what was typed by the user and what was written by the external application. This also means that the application is free to write anywhere on the screen. This should in general be set to True if you expect your application to send ANSI sequences.
- `on_key` is a subprogram that is called every time the user presses a key in the console. This is much lower-level than the other callbacks above, but if you are driving external applications you might have a need to send the keys as they happen, and not wait for a newline. `on_key` receives four parameters:

–\$1: the instance of `GPS.Console`

–\$2: **“keycode”**: this is the internal keycode for the key that the user pressed. All keys can be represented this way, but this will occasionally be left to 0 when the user input was simulated and no real key was pressed.

–\$3: **“key”**: this is the unicode character that the user entered. This will be 0 when the character is not printable (for instance return, tab, key up,...). In python, you can manipulate it with code like `unichr(key).encode("utf8")` to get a string representation that can be sent to an external process

–\$4: **“modifier”**: these are the state of the control, shift, mod1 and lock keys. This is a bit-mask, where shift is 1, lock is 2, control is 4 and mod1 is 8.

Parameters

- **name** – A string
- **force** – A boolean
- **on_input** – A subprogram, see the description below
- **on_destroy** – A subprogram
- **accept_input** – A boolean
- **on_resize** – A subprogram
- **on_interrupt** – A subprogram
- **on_completion** – A subprogram

- **on_key** – A subprogram
- **manage_prompt** – A boolean
- **ansi** – A boolean

accept_input ()

Return True if the console accepts input, False otherwise

Returns A boolean

add_input (*text*)

Add some extra text to the console as if the user had typed it. As opposed to text inserted with `GPS.Console.write`, this text remains editable by the user

Parameters *text* – A string

clear ()

Clear the current contents of the console

clear_input ()

Removes any user input that the user has started typing (ie since the last output inserted through `GPS.Console.write`)

copy_clipboard ()

Copy the selection to the clipboard

create_link (*regexp*, *on_click*)

Register a regular expression that should be highlight in this console to provide hyper links. These links are searched for when calling `GPS.Console.write_with_links`. The part of the text that matches any of the link registered in the console through `GPS.Console.create_link` gets highlighted in blue and underlined, just like an hyper link in a web browser. If the user clicks on that text, *on_click* gets called with one parameter, the text that was clicked on. This can for instance be used to jump to an editor, open a web browser,...

If the regular expression does not contain any parenthesis, the text that matches the whole *regexp* is highlighted as a link. Otherwise, only the part of the text that matches the first parenthesis group is highlighted (so that you can test for the presence of text before or after the actual hyper link).

Parameters

- **regexp** – A string
- **on_click** – A subprogram

See Also:

`GPS.Console.write_with_links()`

enable_input (*enable*)

Make the console accept / reject input according to the value of “enable”

Parameters *enable* – A boolean

flush ()

Do nothing, needed for compatibility with Python’s file class

get_text ()

Return the content of the console

Returns A string

isatty ()

Return True if the console behaves like a terminal. Mostly needed for compatibility with Python’s file class

Returns A boolean

read()

Read the available input in the console. Currently, this behaves exactly like `readline()`

Returns A String

readline()

Ask the user to enter a new line in the console, and returns that line. GPS is blocked until enter has been pressed in the console

Returns A String

select_all()

Select the complete contents of the console

write(text, mode="text")

Output some text on the console. This text is read-only. If the user had started typing some text, that text is temporarily remove, the next text is inserted (read-only), and the user text is put back afterward.

The optional parameter `mode` specifies the kind of the output text: "text" for ordinary messages (this is default), "log" for log messages, and "error" for error messages.

Parameters

- **text** – A utf8 string
- **mode** – A string, one of "text", "log", "error"

See Also:

`GPS.Console.write_with_links()`

`Console().write(u"\N{LATIN CAPITAL LETTER E WITH ACUTE}".encode("utf-8"))`

write_with_links(text)

Output some text on the console, highlight the parts of it that matches the regular expression registered by `GPS.Console.create_link`.

Parameters **text** – A utf8 string

import re

```
console = GPS.Console("myconsole")
console.create_link("([\\w-]+):(\\d+)", open_editor)
console.write_with_link("a file.adb:12 location in a file")

def open_editor(text):
    matched = re.match("[\\w+-]+):(\\d+)", text)
    buffer = GPS.EditorBuffer.get(GPS.File(matched.group(1)))
    buffer.current_view().goto(
        GPS.EditorLocation(buffer, int(matched.group(2)), 1))
```

15.5.16 GPS.Context

class GPS.Context

Represents a context in GPS. Depending on the currently selected window, an instance of one of the derived classes will be used.

module_name = None

The name (a string) of the GPS module which created the context.

15.5.17 GPS.Contextual

class `GPS.Contextual` (*name*)

This class is a general interface to the contextual menus in GPS. It gives you control over which menus should be displayed when the user right clicks in parts of GPS

See Also:

```
GPS.Contextual.__init__()
```

__init__ (*name*)

Initializes a new instance of `GPS.Contextual`. The *name* is the name that was given to the contextual menu when it was created, and is a static string independent of the actual label used when the menu is displayed (and which is dynamic, depending on the context). You can get the list of valid names by checking the list of names returned by `GPS.Contextual.list`

Parameters *name* – A string

See Also:

```
GPS.Contextual.list()
```

```
# You could for instance decide to always hide the "Goto  
# declaration" contextual menu with the following call:
```

```
GPS.Contextual ('Goto declaration of entity').hide()
```

```
# After this, the menu will never be displayed again.
```

create (*on_activate*, *label=None*, *ref=''*, *add_before=True*, *filter=None*, *group='0'*)

Create a new contextual menu entry. Whenever this menu entry is selected by the user, GPS will execute *on_activate*, passing one parameter which is the context for which the menu is displayed (this is generally the same as `GPS.current_contextual()`).

If *on_activate* is `None`, a separator will be created.

The *filter* parameter can be used to filter when the entry should be displayed in the menu. It is a subprogram that receives one parameter, an instance of `GPS.Context`, and returns a boolean. If it returns `True`, the entry will be displayed, otherwise it is hidden.

The *label* parameter can be used to control the text displayed in the contextual menu. By default, it is the same as the contextual name (used in the constructor to `GPS.Contextual.__init__`). If specified, it must be a subprogram that takes an instance of `GPS.Context` in parameter, and returns a string, which will be displayed in the menu.

The parameters *group*, *ref* and *add_before* can be used to control the location of the entry within the contextual menu. *group* allows you to create groups of contextual menus that will be put together. Items of the same group appear before all items with a greater group number. *ref* is the name of another contextual menu entry, and *add_before* indicates whether the new entry is put before or after that second entry.

Parameters

- **on_activate** – A subprogram with one parameter context
- **label** – A subprogram
- **ref** – A string
- **add_before** – A boolean
- **filter** – A subprogram

- **group** – An integer

```

## This example demonstrates how to create a contextual
## menu with global functions

def on_contextual(context):
    GPS.Console("Messages").write("You selected the custom entry")

def on_filter(context):
    return isinstance(context, GPS.EntityContext)

def on_label(context):
    global count
    count += 1
    return "Custom " + count

GPS.Contextual("Custom").create(
    on_activate=on_contextual, filter=on_filter, label=on_label)

## This example is similar to the one above, but uses a python
## class to encapsulate date.
## Note how the extra parameter self can be passed to the callbacks
## thanks to the call to self.create

class My_Context(GPS.Contextual):
    def on_contextual(self, context):
        GPS.Console("Messages").write(
            "You selected the custom entry " + self.data)

    def on_filter(self, context):
        return isinstance(context, GPS.EntityContext)

    def on_label(self, context):
        return self.data

    def __init__(self):
        GPS.Contextual.__init__(self, "Custom")
        self.data = "Menu Name"
        self.create(on_activate=self.on_contextual,
                    filter=self.on_filter,
                    label=self.label)

```

create_dynamic (*factory, on_activate, label='', filter=None, ref='', add_before=True, group='0'*)

Create a new dynamic contextual menu.

This is a submenu of a contextual menu, where the entries are generated by the *factory* parameter. This parameter should return a list of strings, which will be converted to menus by GPS. These strings can contain *'/'* characters to indicate submenus.

filter is a subprogram that takes the `GPS.Context` as a parameter, and returns a boolean indicating whether the submenu should be displayed.

label can be used to specify the label to use for the menu entry. It can include directory-like syntax to indicate submenus. This label can include standard macro substitution (see the GPS documentation), for instance *%e* for the current entity name.

on_activate is called whenever any of the entry of the menu is selected, and is passed three parameters, the context in which the contextual menu was displayed, the string representing the selected entry and the index of the selected entry within the array returned by *factory* (index starts at 0).

The parameters `ref` and `add_before` can be used to control the location of the entry within the contextual menu. `ref` is the name of another contextual menu entry, and `add_before` indicates whether the new entry is put before or after that second entry.

Parameters

- **factory** – A subprogram
- **on_activate** – A subprogram
- **label** – A string
- **filter** – A subprogram
- **ref** – A string
- **add_before** – A boolean
- **group** – A integer

```
## This example shows how to create a contextual menu
## through global functions

def build_contextual(context):
    return ["Choice1", "Choice2"]

def on_activate(context, choice, choice_index):
    GPS.Console("Messages").write("You selected " + choice)

def filter(context):
    return isinstance(context, GPS.EntityContext)

GPS.Contextual("My_Dynamic_Menu").create_dynamic(
    on_activate=on_activate, factory=build_contextual, filter=filter)

## This example is similar to the one above, but shows how
## to create the menu through a python class.
## Note how self can be passed to the callbacks thanks to the
## call to self.create_dynamic.

class Dynamic(GPS.Contextual):
    def __init__(self):
        GPS.Contextual.__init__(self, "My Dynamic Menu")
        self.create_dynamic(on_activate=self.on_activate,
                           label="References/My menu",
                           filter=self.filter,
                           factory=self.factory)

    def filter(self, context):
        return isinstance(context, GPS.EntityContext)

    def on_activate(self, context, choice, choice_index):
        GPS.Console("Messages").write("You selected " + choice)

    def factory(self, context):
        return ["Choice1", "Choice2"]
```

`hide()`

Make sure the contextual menu will never appear when the user right clicks anywhere in GPS. This is the standard way to disable contextual menus

See Also:

```
GPS.Contextual.show()
```

static list ()

Return the list of all registered contextual menus. This is a list of strings which are valid names that can be passed to the constructor of `GPS.Contextual`. These names were created when the contextual menu was registered in GPS.

Returns a list of strings

See Also:

```
GPS.Contextual.__init__()
```

set_sensitive (Sensitivity)

Control whether the contextual menu is grayed-out: False if it should be grayed-out, True otherwise.

Parameters **Sensitivity** – Boolean value

show ()

Make sure the contextual menu will be shown when appropriate. The entry might still be invisible if you right clicked on a context where it doesn't apply, but it will be checked

See Also:

```
GPS.Contextual.hide()
```

15.5.18 GPS.Debugger

class `GPS.Debugger`

Interface to debugger related commands. This class allows you to start a debugger and send commands to it. By connection to the various `debugger_*` hooks, you can also monitor the state of the debugger.

By connecting to the “`debugger_command_action_hook`”, you can also create your own debugger commands, that the user can then type in the debugger console. This is a nice way to implement debugger macros.

While developping such debugger interfaces, it might be useful to modify the file `$HOME/.gps/traces.cfg`, and add a line “`GVD.Out=yes`” in it. This will copy all input/output with the debuggers into the GPS log file.

See Also:

```
GPS.Debugger.__init__()
```

import GPS

```
def debugger_stopped(hook, debugger):
    GPS.Console("Messages").write(
        "hook=" + hook + " on debugger="
        + `debugger.get_num()` + "\n")

def start():
    d = GPS.Debugger.spawn(GPS.File("../obj/parse"))
    d.send("begin")
    d.send("next")
    d.send("next")
    d.send("graph display A")
```

```
GPS.Hook("debugger_process_stopped").add(debugger_stopped)
```

__init__ ()

It is an error to create a `Debugger` instance directly. Instead, use `GPS.Debugger.get()` or `GPS.Debugger.spawn()`

See Also:

`GPS.Debugger.get()`

`GPS.Debugger.spawn()`

close()

Closes the given debugger. This also closes all associated windows (call stack, console,...)

command()

Return the command that is being executed in the debugger. This is often only available when called from the `debugger_state_changed` hook, where it might also indicate the command that just finished

Returns A string

static get (*id=None*)

This command gives access to an already running debugger, and will return an instance of `GPS.Debugger` attached to it. The parameter can be null, in which case the current debugger is returned; it can be an integer, in which case the corresponding debugger is returned (starting at 1); or it can be a file, in which case this function returns the debugger currently debugging that file.

Parameters *id* – Either an integer or an instance of `GPS.File`

Returns An instance of `GPS.Debugger`

get_executable()

Returns the name of the executable currently debugged in that debugger

Returns An instance of `GPS.File`

See Also:

`GPS.Debugger.get_num()`

get_num()

Returns the index of the debugger. This can be used later on to retrieve the debugger from `GPS.Debugger.get()`, or to get access to other windows associated with that debugger

Returns An integer

See Also:

`GPS.Debugger.get_file()`

is_break_command()

Return true if the command returned by `GPS.Debugger.command` is likely to modify the list of breakpoints after it has finished executing

Returns A boolean

is_busy()

Returns true if the debugger is currently executing a command. In this case, it is an error to send a new command to it

Returns A boolean

is_context_command()

Return true if the command returned by `GPS.Debugger.command` is likely to modify the current context (current task, thread,...) after it has finished executing

Returns A boolean

is_exec_command()

Return true if the command returned by `GPS.Debugger.command` is likely to modify the stack trace in the debugger (“next”, “cont”, ...)

Returns A boolean

static list ()

This command returns the list of currently running debuggers

Returns A list of `GPS.Debugger` instances

non_blocking_send (*cmd*, *output=True*)

This command works like `send`, but is not blocking, and does not return the result.

Parameters

- **cmd** – A string
- **output** – A boolean

See Also:

`GPS.Debugger.send()`

send (*cmd*, *output=True*, *show_in_console=False*)

This command executes *cmd* in the debugger. GPS is blocked while *cmd* is executing on the debugger. If *output* is true, the command is displayed in the console.

If *show_in_console* is True, the output of the command is displayed in the debugger console, but is not returned by this function. If *show_in_console* is False, the result is not displayed in the console, but is returned by this function

Parameters

- **cmd** – A string
- **output** – A boolean
- **show_in_console** – A boolean

Returns A string

See Also:

`GPS.Debugger.non_blocking_send()`

static spawn (*executable*, *args=''*)

This command starts anew debugger. It will debug file. When file is executed, the extra arguments *args* are passed

Parameters

- **executable** – An instance of `GPS.File`
- **args** – A string

Returns An instance of `GPS.Debugger`

15.5.19 GPS.Docgen

class `GPS.Docgen`

Interface for handling customized documentation generation. This class is used in conjunction with `GPS.DocgenTagHandler`. You cannot create directly this class, but use the ones furnished in `GPS.DocgenTagHandler` callbacks.

See Also:

`GPS.DocgenTagHandler()`

generate_index_file (*name, filename, content*)

Create a new Index file. The file 'filename' will be titled 'name', and will contain the general decoration along with 'content'.

All other generated documentation file will have a link to it for convenience.

Parameters

- **name** – The name of the new index file.
- **filename** – The created file name.
- **content** – The content of the created file.

get_current_file ()

Retrieves the current analysed source file. You should call this method only from a `GPS.DocgenTagHandler.on_match()` callback.

Returns A `GPS.File` instance

get_doc_dir ()

Retrieves the directory that will contain the documentation. You should call this method only from a `GPS.DocgenTagHandler.on_match()` callback.

Returns A `GPS.File` instance

static register_css (*filename*)

Registers a new CSS file to use when generating the documentation. This allows either to override a default style, or add new ones for custom tags handling

Parameters filename – A file name

static register_main_index (*filename*)

Registers the file to be used as main page (e.g. `index.html`). By default, the first page generated in the Table of Contents is used.

Parameters filename – A file name

static register_tag_handler (*handler*)

Registers a new tag handler. This handler will be used each time a new documentation is generated and the corresponding tag is found

Parameters handler – The handler to register

```
# register a default handler for tag <description>
# that is, -- <description>sth</description>
# will be translated as <div class="description">sth</div>
GPS.Docgen.register_tag_handler(GPS.DocgenTagHandler ("description"))
```

15.5.20 `GPS.DocgenTagHandler`

class `GPS.DocgenTagHandler` (*tag, on_start=None, on_match=None, on_exit=None*)

This class is used to handle user-defined documentation tags. This allows custom handling of comments such as

```
-- <summary>This fn does something</summary>
```

See Also:

`GPS.Docgen` ()

```

import GPS

class ScreenshotTagHandler(GPS.DocgenTagHandler):
    "Handling for <screenshot>screen.jpg</screenshot>"

    def __init__(self):
        GPS.DocgenTagHandler.__init__(
            self, "screenshot",
            on_match=self.on_match, on_start=self.on_start, on_exit=self.on_exit)

    def on_start(self, docgen):
        self.list = {}

    def on_match(self, docgen, attrs, value, entity_name, entity_href):
        # In this examples, images are in the directory _project_root_/doc/imgs/

        dir = docgen.get_current_file().project().file().directory()+"doc/imgs/"
        img = '' % (dir, value, value)
        self.list[entity_name] = [entity_href, img]
        return "<h3>Screenshot</h3><p>%s</p>" % (img)

    def on_exit(self, docgen):
        content=""

        for pict in sorted(self.list.keys()):
            content += "<div class='subprograms'>"
            content += "  <div class='class'>"
            content += "    <h3>%s</h3>" % (pict)
            content += "    <div class='comment'>"
            content += "      <a href=\"%s\">%s</a>" % (self.list[pict][0], self.list[pict][1])
            content += "    </div>"
            content += "  </div>"
            content += "</div>"

        if content != "":
            docgen.generate_index_file("Screenshots", "screenshots.html", content)

    def on_gps_start(hook):
        GPS.Docgen.register_css(GPS.get_system_dir() + "share/mycustomfiles/custom.css")
        GPS.Docgen.register_tag_handler(ScreenshotTagHandler())

GPS.Hook("gps_started").add(on_gps_start)

__init__(tag, on_start=None, on_match=None, on_exit=None)

```

Create a new `GPS.DocgenTagHandler` instance handling the tag “tag”. You need to register it afterwards using `GPS.Docgen.register_tag_handler`.

`on_match` is a callback that is called each time a tag corresponding to the `GPS.DocgenTagHandler` is analysed. It takes the following parameters:

- \$1 = the instance of `GPS.Docgen`.
- \$2 = the eventual attributes of the tag.
- \$3 = the value of the tag.
- \$4 = the entity name linked to the analysed tag.
- \$5 = the href to the entity documentation location.

`on_start` is a callback that is called each time a documentation generation starts. It takes the following parameters:

- `$1` = the instance of `GPS.Docgen`.

`on_exit` is a callback that is called each time a documentation generation finishes. It takes the following parameters:

- `$1` = the instance of `GPS.Docgen`.

Using the default values of the callbacks (e.g. `None`), the `GPS.DocgenTagHandler` handler will translate comments of the form “- <tag>value</tag>” by “<div class=“tag”>value</div>”.

Parameters

- **tag** – The tag that is handled
- **on_start** – A subprogram
- **on_match** – A subprogram
- **on_exit** – A subprogram

15.5.21 `GPS.Editor`

class `GPS.Editor`

Deprecated interface to all editor-related commands

static `add_blank_lines` (*file, start_line, number_of_lines, category=''*)
OBSOLESCENT.

Adds `number_of_lines` non-editable lines to the buffer editing file, starting at line `start_line`. If `category` is specified, use it for highlighting. Create a mark at beginning of block and return its ID

Parameters

- **file** – A string
- **start_line** – An integer
- **number_of_lines** – An integer
- **category** – A string

static `add_case_exception` (*name*)
OBSOLESCENT.

Add `name` into the case exception dictionary

Parameters `name` – A string

static `block_fold` (*file, line=None*)
OBSOLESCENT.

Fold the block around line. If `line` is not specified, fold all blocks in the file.

Parameters

- **file** – A string
- **line** – An integer

static `block_get_end` (*file, line*)
OBSOLESCENT.

Returns ending line number for block enclosing line

Parameters

- **file** – A string
- **line** – An integer

Returns An integer

static block_get_level (*file, line*)
OBSOLESCECENT.

Returns nested level for block enclosing line

Parameters

- **file** – A string
- **line** – An integer

Returns An integer

static block_get_name (*file, line*)
OBSOLESCECENT.

Returns name for block enclosing line

Parameters

- **file** – A string
- **line** – An integer

Returns A string

static block_get_start (*file, line*)
OBSOLESCECENT.

Returns ending line number for block enclosing line

Parameters

- **file** – A string
- **line** – An integer

Returns An integer

static block_get_type (*file, line*)
OBSOLESCECENT.

Returns type for block enclosing line

Parameters

- **file** – A string
- **line** – An integer

Returns A string

static block_unfold (*file, line=None*)
OBSOLESCECENT.

Unfold the block around line. If line is not specified, unfold all blocks in the file.

Parameters

- **file** – A string
- **line** – An integer

static close (*file*)

OBSOLESCENT.

Close all file editors for file

Parameters **file** – A string

static copy ()

OBSOLESCENT.

Copy the selection in the current editor

static create_mark (*filename, line=1, column=1, length=0*)

Create a mark for file_name, at position given by line and column. Length corresponds to the text length to highlight after the mark. The identifier of the mark is returned. Use the command goto_mark to jump to this mark

Parameters

- **filename** – A string
- **line** – An integer
- **column** – An integer
- **length** – An integer

Returns A string

See Also:

`GPS.Editor.goto_mark()`

`GPS.Editor.delete_mark()`

static cursor_center (*file*)

OBSOLESCENT.

Scroll the view to center cursor

Parameters **file** – A string

static cursor_get_column (*file*)

OBSOLESCENT.

Returns current cursor column number

Parameters **file** – A string

Returns An integer

static cursor_get_line (*file*)

OBSOLESCENT.

Returns current cursor line number

Parameters **file** – A string

Returns An integer

static cursor_set_position (*file, line, column=1*)

OBSOLESCENT.

Set cursor to position line/column in buffer file

Parameters

- **file** – A string

- **line** – An integer
- **column** – An integer

static cut ()

OBSOLESCENT.

Cut the selection in the current editor

static delete_mark (*identifier*)

OBSOLESCENT.

Delete the mark corresponding to identifier

Parameters **identifier** – A string

See Also:

`GPS.Editor.create_mark()`

static edit (*filename*, *line=1*, *column=1*, *length=0*, *force=False*, *position=5*)

OBSOLESCENT.

Open a file editor for file_name. Length is the number of characters to select after the cursor. If line and column are set to 0, then the location of the cursor is not changed if the file is already opened in an editor. If force is set to true, a reload is forced in case the file is already open. Position indicates the MDI position to open the child in (5 for default, 1 for bottom).

The filename can be a network file name, with the following general format:

```
protocol://username@host:port/full/path
```

where protocol is one of the recognized protocols (http, ftp,... see the GPS documentation), and the username and port are optional.

Parameters

- **filename** – A string
- **line** – An integer
- **column** – An integer
- **length** – An integer
- **force** – A boolean
- **position** – An integer

static get_buffer (*file*)

OBSOLESCENT.

Returns the text contained in the current buffer for file

Parameters **file** – A string

static get_chars (*filename*, *line=0*, *column=1*, *before=-1*, *after=-1*)

OBSOLESCENT.

Get the characters around a certain position. Returns string between “before” characters before the mark and “after” characters after the position. If “before” or “after” is omitted, the bounds will be at the beginning and/or the end of the line.

If the line and column are not specified, then the current selection is returned, or the empty string if there is no selection

Parameters

- **filename** – A string
- **line** – An integer
- **column** – An integer
- **before** – An integer
- **after** – An integer

Returns A string

static get_column (*mark*)
OBSOLESCE

Returns the current column of mark

Parameters mark – An identifier

Returns An integer

static get_file (*mark*)
OBSOLESCE

Returns the current file of mark

Parameters mark – An identifier

Returns A file

static get_last_line (*file*)
OBSOLESCE

Returns the number of the last line in file

Parameters file – A string

Returns An integer

static get_line (*mark*)
OBSOLESCE

Returns the current line of mark

Parameters mark – An identifier

Returns An integer

static goto_mark (*identifier*)
OBSOLESCE

Jump to the location of the mark corresponding to identifier

Parameters identifier – A string

See Also:

`GPS.Editor.create_mark()`

static highlight (*file, category, line=0*)
OBSOLESCE

Marks a line as belonging to a highlighting category. If line is not specified, mark all lines in file.

Parameters

- **file** – A string
- **category** – A string

- **line** – An integer

See Also:

`GPS.Editor.unhighlight()`

static highlight_range (*file, category, line=0, start_column=0, end_column=-1*)
OBSOLESCEMENT>

Highlights a portion of a line in a file with the given category

Parameters

- **file** – A string
- **category** – A string
- **line** – An integer
- **start_column** – An integer
- **end_column** – An integer

static indent (*current_line_only=False*)
OBSOLESCEMENT.

Indent the selection (or the current line if requested) in current editor. Do nothing if the current GPS window is not an editor

Parameters **current_line_only** – A boolean

static indent_buffer ()
OBSOLESCEMENT.

Indent the current editor. Do nothing if the current GPS window is not an editor

static insert_text (*text*)
OBSOLESCEMENT.

Insert a text in the current editor at the cursor position

Parameters **text** – A string

static mark_current_location ()
OBSOLESCEMENT.

Push the location in the current editor in the history of locations. This should be called before jumping to a new location on a user's request, so that he can easily choose to go back to the previous location.

static paste ()
OBSOLESCEMENT.

Paste the selection in the current editor

static print_line_info (*file, line*)
OBSOLESCEMENT.

Print the contents of the items attached to the side of a line. This is used mainly for debugging and testing purposes.

Parameters

- **file** – A string
- **line** – An integer

static redo (*file*)

OBSOLESCENT.

Redo the last undone edition command for file

Parameters file – A string

static refill ()

OBSOLESCENT.

Refill selected (or current) editor lines. Do nothing if the current GPS window is not an editor

static register_highlighting (*category, color, speedbar=False*)

OBSOLESCENT.

Create a new highlighting category with the given color. The format for color is “#RRGGBB”. If speedbar is true, then a mark will be inserted in the speedbar to the left of the editor to give a fast overview to the user of where the highlighted lines are.

Parameters

- **category** – A string
- **color** – A string
- **speedbar** – A boolean

static remove_blank_lines (*mark, number=0*)

OBSOLESCENT

Remove blank lines located at mark. If number is specified, remove only the number first lines

Parameters

- **mark** – A string
- **number** – An integer

static remove_case_exception (*name*)

OBSOLESCENT.

Remove name from the case exception dictionary

Parameters name – A string

static replace_text (*file, line, column, text, before=-1, after=-1*)

OBSOLESCENT.

Replace the characters around a certain position. “before” characters before (line, column), and up to “after” characters after are removed, and the new text is inserted instead. If “before” or “after” is omitted, the bounds will be at the beginning and/or the end of the line

Parameters

- **file** – A string
- **line** – An integer
- **column** – An integer
- **text** – A string
- **before** – An integer
- **after** – An integer

static save (*interactive=True, all=True*)
OBSOLESCECENT.

Save current or all files. If interactive is true, then prompt before each save. If all is true, then all files are saved

Parameters

- **interactive** – A boolean
- **all** – A boolean

static save_buffer (*file, to_file=None*)
OBSOLESCECENT.

Saves the text contained in the current buffer for file. If to_file is specified, the file will be saved as to_file, and the buffer status will not be modified

Parameters

- **file** – A string
- **to_file** – A string

static select_all ()
OBSOLESCECENT.

Select the whole editor contents

static select_text (*first_line, last_line, start_column=1, end_column=0*)
OBSOLESCECENT.

Select a block in the current editor

Parameters

- **first_line** – An integer
- **last_line** – An integer
- **start_column** – An integer
- **end_column** – An integer

static set_background_color (*file, color*)
OBSOLESCECENT.

Set the background color for the editors for file

Parameters

- **file** – A string
- **color** – A string

static set_synchronized_scrolling (*file1, file2, file3=''*)
OBSOLESCECENT.

Synchronize the scrolling between multiple editors

Parameters

- **file1** – A string
- **file2** – A string
- **file3** – A string

static set_title (*file, title, filename*)
OBSOLESCENT.

Change the title of the buffer containing the given file

Parameters

- **file** – A string
- **title** – A string
- **filename** – A string

static set_writable (*file, writable*)
OBSOLESCENT.

Change the Writable status for the editors for file

Parameters

- **file** – A string
- **writable** – A boolean

static subprogram_name (*file, line*)
OBSOLESCENT.

Returns the name of the subprogram enclosing line

Parameters

- **file** – A string
- **line** – An integer

Returns A string

static undo (*file*)
OBSOLESCENT.

Undo the last edition command for file

Parameters **file** – A string

static unhighlight (*file, category, line=0*)
OBSOLESCENT.

Unmarks the line for the specified category. If line is not specified, unmark all lines in file

Parameters

- **file** – A string
- **category** – A string
- **line** – An integer

See Also:

`GPS.Editor.highlight()`

static unhighlight_range (*file, category, line=0, start_column=0, end_column=-1*)
OBSOLESCENT.

Remove highlights for a portion of a line in a file

Parameters

- **file** – A string

- **category** – A string
- **line** – An integer
- **start_column** – An integer
- **end_column** – An integer

15.5.22 `GPS.EditorBuffer`

`class GPS.EditorBuffer`

This class represents the physical contents of a file. It is always associated with at least one view (a `GPS.EditorView` instance), which makes it visible to the user. The contents of the file can be manipulated through this class

`__init__()`

This function prevents the direct creation of instances of `EditorBuffer`. Use `GPS.EditorBuffer.get()` instead

`add_multi_cursor(location)`

Adds a new multi cursor at the given location.

`add_special_line(start_line, text, category='', name='')`

Adds one non-editable line to the buffer, starting at line `start_line` and contains string `text`. If `category` is specified, use it for highlighting. Create a mark at beginning of block and return it. If `name` is specified, returned mark will have this name

Parameters

- **start_line** – An integer
- **text** – A string
- **category** – A string
- **name** – A string

Returns An instance of `GPS.EditorMark`

See Also:

`GPS.EditorBuffer.get_mark()`

`apply_overlay(overlay, frm='beginning of buffer', to='end of buffer')`

Applies the overlay to the given range of text. This immediately changes the rendering of the text based on the properties of the overlay

Parameters

- **overlay** – An instance of `GPS.EditorOverlay`
- **frm** – An instance of `GPS.EditorLocation`
- **to** – An instance of `GPS.EditorLocation`

See Also:

`GPS.EditorBuffer.remove_overlay()`

`beginning_of_buffer()`

Returns a location pointing to the first character in the buffer

Returns An instance of `GPS.EditorLocation`

blocks_fold()

Folds all the blocks in all the views of the buffer. Block folding is a language-dependent feature, whereby one can hide part of the source code temporarily, by keeping only the first line of the block (for instance the first line of a subprogram body, the rest is hidden). A small icon is displayed to the left of the first line so that it can be unfolded later on

See Also:

`GPS.EditorBuffer.blocks_unfold()`

`GPS.EditorLocation.block_fold()`

blocks_unfold()

Unfolds all the blocks that were previously folded in the buffer, ie make the whole source code visible. This is a language dependent feature

See Also:

`GPS.EditorBuffer.blocks_fold()`

`GPS.EditorLocation.block_unfold()`

characters_count()

Returns the total number of characters in the buffer

Returns An integer

close (*force=False*)

Closes the editor and all its views. If the buffer has been modified and not saved, a dialog is open asking the user whether to save. If *force* is True, do not save and do not ask the user. All changes are lost

Parameters *force* – A boolean

copy (*frm='beginning of buffer', to='end of buffer', append=False*)

Copy the given range of text into the clipboard, so that it can be further pasted into other applications or other parts of GPS. If *append* is True, the text is appended to the last clipboard entry instead of generating a new one

Parameters

- **frm** – An instance of `GPS.EditorLocation`
- **to** – An instance of `GPS.EditorLocation`
- **append** – A boolean

See Also:

`GPS.Clipboard.copy()`

create_overlay (*name=''*)

Create a new overlay. Properties can be set on this overlay, which can then be applied to one or more ranges of text to changes its visual rendering or to associate user data with it. If *name* is specified, this function will return an existing overlay with the same name in this buffer if any can be found. If the name is not specified, a new overlay is created. Changing the properties of an existing overlay results in an immediate graphical update of the views associated with the buffer.

A number of predefined overlay exists. Among these are the ones used for syntax highlighting by GPS itself, which are “keyword”, “comment”, “string”, “character”. You can use these to navigate from one comment section to the next for instance.

Parameters *name* – A string

Returns An instance of `GPS.EditorOverlay`

current_view ()

Returns the last view used for this buffer, ie the last view that had the focus and through which the user might have edited the buffer's contents

Returns An instance of `GPS.EditorView`

cut (*frm='beginning of buffer', to='end of buffer', append=False*)

Copy the given range of text into the clipboard, so that it can be further pasted into other applications or other parts of GPS. The text is removed from the edited buffer. If `append` is `True`, the text is appended to the last clipboard entry instead of generating a new one

Parameters

- **frm** – An instance of `GPS.EditorLocation`
- **to** – An instance of `GPS.EditorLocation`
- **append** – A boolean

delete (*frm='beginning of buffer', to='end of buffer'*)

Delete the given range of text from the buffer

Parameters

- **frm** – An instance of `GPS.EditorLocation`
- **to** – An instance of `GPS.EditorLocation`

end_of_buffer ()

Returns a location pointing to the last character in the buffer

Returns An instance of `GPS.EditorLocation`

expand_alias (*alias*)

Expand given alias in the editor buffer at the point where the cursor is.

file ()

Returns the name of the file edited in this buffer

Returns An instance of `GPS.File`

finish_undo_group ()

Cancels the grouping of commands on the editor. See `GPS.EditorBuffer.start_undo_group`

static get (*file='current editor', force=False, open=True*)

If file is already opened in an editor, get a handle on its buffer. This instance is then shared with all other buffers referencing the same file. As a result, you can for instance associate your own data with the buffer, and retrieve it at any time until the buffer is closed. If the file is not opened yet, it is loaded in a new editor, and a new view is opened at the same time (and thus the editor becomes visible to the user). If file is not specified, the current editor is returned, ie the last one that had the keyboard focus.

If the file is not currently open, the behavior depends on the `open` parameter: if `true`, a new editor is created for that file, otherwise `None` is returned.

When a new file is open, it has received the focus. But if the editor already existed, it is not raised explicitly, and you need to do it yourself through a call to `GPS.MDIWindow.raise_window` (see the example below).

If `force` is set to `true`, a reload is forced in case the file is already open.

Parameters

- **file** – An instance of `GPS.File`
- **force** – A boolean
- **open** – A boolean

Returns An instance of `GPS.EditorBuffer`

```
ed = GPS.EditorBuffer.get(GPS.File("a.adb"))
GPS.MDI.get_by_child(ed.current_view()).raise_window()
ed.data = "whatever"

# ... Whatever, including modifying ed

ed = GPS.EditorBuffer.get(GPS.File("a.adb"))
ed.data # => "whatever"
```

get_chars (*frm='beginning of buffer', to='end of buffer'*)

Returns the contents of the buffer between the two locations given in parameter. Modifying the returned value has no effect on the buffer

Parameters

- **frm** – An instance of `GPS.EditorLocation`
- **to** – An instance of `GPS.EditorLocation`

Returns A string

get_mark (*name*)

Check whether there is a mark with that name in the buffer, and return it. An exception is raised if there is no such mark

Parameters *name* – A string

Returns An instance of `GPS.EditorMark`

See Also:

```
GPS.EditorLocation.create_mark()

ed = GPS.EditorBuffer.get(GPS.File("a.adb"))
loc = GPS.EditorLocation(ed, 4, 5)
mark = loc.create_mark("name")
mark.data = "whatever"

# .. anything else

mark = ed.get_mark("name")
# mark.data is still "whatever"
```

get_multi_cursors_marks ()

Returns the list of all marks corresponding to existing multi cursors in that buffer. Note that if you intend to perform actions with them (in particular deletions/insertions), you should call `set_multi_cursors_manual_sync`, with the cursor mark as argument. See `set_multi_cursors_*` functions for more details

Returns A list of `GPS.EditorMark` instances

static get_new ()

Open a new editor on a blank file. This file has no name, and you'll have to provide one when you save it

Returns An instance of `GPS.EditorBuffer`

indent (*frm='beginning of buffer', to='end of buffer'*)

Recompute the indentation of the given range of text. This feature is language-dependent

Parameters

- **frm** – An instance of `GPS.EditorLocation`

- **to** – An instance of `GPS.EditorLocation`

insert (*location*, *text*)

Inserts some text in the buffer

Parameters

- **location** – An instance of `GPS.EditorLocation`
- **text** – A string

See Also:

`GPS.EditorBuffer.delete()`

is_modified ()

Tests whether the buffer has been modified since it was last open or saved

Returns A boolean

is_read_only ()

Whether the buffer is editable or not.

Returns A boolean

See Also:

`GPS.EditorBuffer.set_read_only()`

lines_count ()

Returns the total number of lines in the buffer

Returns An integer

static list ()

This function returns the list of all editors that are currently open in GPS.

Returns A list of instances of `GPS.EditorBuffer`

It is possible to close all editors at once using a command like

```
for ed in GPS.EditorBuffer.list():
    ed.close()
```

paste (*location*)

Paste the contents of the clipboard at the given location in the buffer

Parameters **location** – An instance of `GPS.EditorLocation`

redo ()

Redo the last undone command on the editor

refill (*frm='beginning of buffer'*, *to='end of buffer'*)

Refill the given range of text, ie cut long lines if necessary so that they fit in the limit specified in the GPS preferences

Parameters

- **frm** – An instance of `GPS.EditorLocation`
- **to** – An instance of `GPS.EditorLocation`

remove_all_multi_cursors ()

Removes all active multi-cursors from the buffer

remove_overlay (*overlay, frm='beginning of buffer', to='end of buffer'*)

Removes all instances of the overlay in the given range of text. It isn't an error if the overlay is not applied to any of the character in the range, it just has no effect in that case.

Parameters

- **overlay** – An instance of `GPS.EditorOverlay`
- **frm** – An instance of `GPS.EditorLocation`
- **to** – An instance of `GPS.EditorLocation`

See Also:

`GPS.EditorBuffer.apply_overlay()`

remove_special_lines (*mark, lines*)

Removes specified number of special lines at the specified mark. It doesn't delete the mark

Parameters

- **mark** – An instance of `GPS.EditorMark`
- **lines** – An integer

save (*interactive=True, file='Same file as edited by the buffer'*)

Saves the buffer to the given file. If `interactive` is true, a dialog is open to ask for confirmation from the user first, which gives him a chance to cancel the saving. “interactive” is ignored if file is specified.

Parameters

- **interactive** – A boolean
- **file** – An instance of `GPS.File`

select (*frm='beginning of buffer', to='end of buffer'*)

Selects an area in the buffer. The boundaries are included in the selection. The order of the boundaries is irrelevant, but the cursor will be left on to

Parameters

- **frm** – An instance of `GPS.EditorLocation`
- **to** – An instance of `GPS.EditorLocation`

selection_end ()

Return the character after the end of the selection. This will always be located after the start of the selection, no matter the order of parameters given to `GPS.EditorBuffer.select`. If the selection is empty, `EditorBuffer.selection_start` and `EditorBuffer.selection_end` will be equal.

Returns An instance of `GPS.EditorLocation`

To get the contents of the current selection, one would use:

```
buffer = GPS.EditorBuffer.get()
selection = buffer.get_chars(
    buffer.selection_start(), buffer.selection_end() - 1)
```

selection_start ()

Return the start of the selection. This will always be located before the end of the selection, no matter the order of parameters given to `GPS.EditorBuffer.select`

Returns An instance of `GPS.EditorLocation`

set_multi_cursors_auto_sync()

Set the buffer in auto sync mode regarding multi cursors. This means that any insertion/deletion will be propagated in a ‘naive’ way on all multi cursors. Cursor movements won’t be propagated.

set_multi_cursors_manual_sync (*multi_cursor_mark=None*)

Set the buffer in manual sync mode regarding multi cursors. *multi_cursor_mark* should be the mark corresponding to the multi cursor that is gonna be affected, or *None* if the action is from the main cursor. This info is useful to provide correct undo/redo actions for custom multi cursors actions.

set_read_only (*read_only=True*)

Indicates whether the user should be able to edit the buffer interactively (through any view).

Parameters *read_only* – A boolean

See Also:

`GPS.EditorBuffer.is_read_only()`

start_undo_group()

Starts grouping commands on the editor. All future editions will be considered as belonging to the same group. `finish_undo_group` should be called once for every call to `start_undo_group`.

undo()

Undo the last command on the editor

unselect()

Cancel the current selection in the buffer

views()

Returns the list of all views currently editing the buffer. There is always at least one such view. When the last view is destroyed, the buffer itself is destroyed

Returns A list of `GPS.EditorView` instances

15.5.23 `GPS.EditorHighlighter`

class `GPS.EditorHighlighter` (*pattern, action, index=0, secondary_action=None*)

This class can be used to transform source editor text into hyperlinks when the Control key is pressed. Two actions can then be associated with this hyperlink: clicking with the left mouse button on the hyperlink triggers the primary action, and clicking with the middle mouse button on the hyperlink triggers the alternate action.

__init__ (*pattern, action, index=0, secondary_action=None*)

Register a highlighter. The action is a Python function that takes a string as a parameter: the string being passed is the section of text which is highlighted.

Parameters

- **pattern** – A regular expression representing the patterns on which we want to create hyperlinks.
- **action** – The primary action for this hyperlink
- **index** – This indicate the number of the parenthesized group in pattern that needs to be highlighted.
- **secondary_action** – The alternate action for this hyperlink

```
# Define an action
def view_html(url):
    GPS.HTML.browse (url)

def wget_url(url):
```

```
def on_exit_cb(self, code, output):
    GPS.Editor.edit (GPS.dump (output))
    p=GPS.Process("wget %s -O -" % url, on_exit=on_exit_cb)

# Register a highlighter to launch a browser on any URL
# left-clicking on an URL will open the default browser to this URL
# middle-clicking will call "wget" to get the source of this URL and
# open the output in a new editor

h=GPS.EditorHighlighter ("http(s)?://[^\s:]*", view_html, 0, wget_url)

# Remove the highlighter
h.remove()
```

remove()

Unregister the highlighter. This cannot be called while the hyper-mode is active.

15.5.24 GPS.EditorLocation

class `GPS.EditorLocation` (*buffer, line, column*)

This class represents a location in a specific editor buffer. This location is not updated when the buffer changes, but will keep pointing to the same line/column even if new lines are added in the buffer. This location is no longer valid when the buffer itself is destroyed, and the use of any of these subprograms will raise an exception.

See Also:

`GPS.EditorMark()`

__init__ (*buffer, line, column*)

Initializes a new instance. Creating two instances at the same location will not return the same instance of `GPS.EditorLocation`, and therefore any user data you have stored in the location will not be available in the second instance

Parameters

- **buffer** – The instance of `GPS.EditorBuffer`
- **line** – An integer
- **column** – An integer

```
ed = GPS.EditorBuffer.get(GPS.File("a.adb"))
loc = GPS.EditorLocation(ed, line=4, column=5)
loc.data = "MY OWN DATA"
loc2 = GPS.EditorLocation(ed, line=4, column=5)
# loc2.data is not defined at this point
```

backward_overlay (*overlay=None*)

Same as `GPS.EditorLocation.forward_overlay`, but moves backward instead. If there are no more changes, the location is left at the beginning of the buffer.

Parameters *overlay* – An instance of `GPS.EditorOverlay`

Returns An instance of `GPS.EditorLocation`

beginning_of_line ()

Return a location located at the beginning of the line on which self is.

Returns A new instance of `GPS.EditorLocation`

block_end()

Return the location of the end of the current block

Returns An instance of `GPS.EditorLocation`

block_end_line()

Return the last line of the block surrounding the location. The definition of a block depends on the specific language of the source file

Returns An integer

block_fold()

Fold the block containing the location, ie make it invisible on the screen, except for its first line. Clicking on the icon next to this first line will unfold the block and make it visible to the user

See Also:

`GPS.EditorLocation.block_unfold()`

block_level()

Return the nesting level of the block surrounding the location. The definition of a block depends on the specific programming language

Returns An integer

block_name()

Return the name of the block surrounding the location. The definition of a block depends on the specific language of the source file

Returns A string

block_start()

Return the location of the beginning of the current block

Returns An instance of `GPS.EditorLocation`

block_start_line()

Return the first line of the block surrounding the location. The definition of a block depends on the programming language

Returns An integer

block_type()

Return the type of the block surrounding the location. This type indicates whether the block is a subprogram, an if statement,...

Returns A string

block_unfold()

Unfold the block containing the location, ie make it visible any information that was hidden as a result of running `GPS.EditorLocation.block_fold`

See Also:

`GPS.EditorLocation.block_fold()`

buffer()

Return the buffer in which the location is found

Returns An instance of `GPS.EditorBuffer`

column()

Return the column of the location

Returns An integer

create_mark (*name*='')

Create a mark at that location in the buffer. The mark will stay permanently at that location, and follows if the buffer is modified. If the name is specified, this creates a named mark, which can be retrieved through a call to `GPS.EditorBuffer.get_mark`. If a mark with the same name already exists, it is moved to the new location, and then returned

Parameters *name* – A string

Returns An instance of `GPS.EditorMark`

See Also:

`GPS.EditorBuffer.get_mark()`

```
buffer = GPS.EditorBuffer.get(GPS.File("a.adb"))
loc = GPS.EditorLocation(buffer, 3, 4)
mark = loc.create_mark()
buffer.insert(loc, "text")
loc = mark.location()
# loc.column() is now 8
```

end_of_line ()

Return a location located at the end of the line on which self is.

Returns A new instance of `GPS.EditorLocation`

ends_word ()

Return true if self is currently at the end of a word. The definition of a word depends on the language used

Returns A boolean

forward_char (*count*)

Return a new location located count characters after self. If count is negative, the location is moved backward instead

Parameters *count* – An integer

Returns A new instance of `GPS.EditorLocation`

forward_line (*count*)

Return a new location located count lines after self. The location is moved back to the beginning of the line. In case self is on the last line, the beginning of the last line is returned.

Parameters *count* – An integer

Returns A new instance of `GPS.EditorLocation`

forward_overlay (*overlay*='')

Moves to the next change in the list of overlays applying to the character. If overlay is specified, go to the next change for this specific overlay (ie the next beginning or end of range where it applies). If there are no more changes, the location is left at the end of the buffer.

Parameters *overlay* – An instance of `GPS.EditorOverlay`

Returns An instance of `GPS.EditorLocation`

See Also:

`GPS.EditorLocation.backward_overlay()`

forward_word (*count*)

Return a new location located count words after self. If count is negative, the location is moved backward instead. The definition of a word depends on the language used

Parameters *count* – An integer

Returns A new instance of `GPS.EditorLocation`

`get_char()`

Return the character at that location in the buffer. An exception is raised when trying to read past the end of the buffer. The character might be encoded on several bytes, since it is a UTF8 string.

Returns A UTF8 string

```
char = buffer.beginning_of_buffer().get_char()
GPS.Console().write(char)  ## Prints the character
# To manipulate in python, convert the string to a unicode string:
unicode = char.decode("utf-8")
```

`get_overlays()`

This function returns the list of all the overlays that apply at this specific location. The color and font of the text is composed through the contents of these overlays.

Returns A list of `GPS.EditorOverlay` instances

`has_overlay(overlay)`

This function returns True if the given overlay applies to the character at that location

Parameters `overlay` – An instance of `GPS.EditorOverlay`

Returns A boolean

`inside_word()`

Return true if self is currently inside a word. The definition of a word depends on the language used

Returns A boolean

`line()`

Return the line of the location

Returns An integer

`offset()`

Return the offset of the location in the buffer, ie the number of characters from the beginning of the buffer to the location

Returns An integer

`search(pattern, backward=False, case_sensitive=False, regexp=False, whole_word=False, scope='Whole', dialog_on_failure=True)`

This function searches for the next occurrence of `Pattern` in the editor, starting at the given location. If there is such a match, this function returns the two locations for the beginning of the match and the end of the match. Typically, these would be used to highlight the match in the editor.

When no match is found, this function returns null. Additionally, if `dialog_on_failure` is true then a dialog is displayed to the user asking whether the search should restart at the beginning of the buffer.

Parameters

- **pattern** – A string
- **backward** – A boolean
- **case_sensitive** – A boolean
- **regexp** – A boolean
- **whole_word** – A boolean
- **scope** – A string
- **dialog_on_failure** – A boolean

Returns A list of two `GPS.EditorLocation`

See Also:

`GPS.File.search()`

starts_word()

Return true if self is currently at the start of a word. The definition of a word depends on the language used

Returns A boolean

subprogram_name()

Return the name of the subprogram enclosing the location

Returns A string

15.5.25 `GPS.EditorMark`

class `GPS.EditorMark`

This class represents a specific location in an open editor. As opposed to the `GPS.EditorLocation` class, the exact location is updated whenever the buffer is modified. For instance, if you add a line before the mark, then the mark is moved one line forward as well, so that it still points to the same character in the buffer.

The mark remains valid even if you close the buffer; or if you reopen it and modify it. It will always point to the same location in the file, while you have kept the python object.

`GPS.EditorLocation.create_mark()` allows you to create named marks which you can then retrieve through `GPS.EditorBuffer.get_mark`. Such named marks are only valid while the editor exists. As soon as you close the editor, you can no longer use `get_mark` to retrieve it (but the mark is still valid if you have kept a python object referencing it).

See Also:

`GPS.EditorLocation()`

__init__()

This subprogram will always raise an exception, thus preventing the direct creation of a mark. Instead, you should use `GPS.EditorLocation.create_mark()` to create such a mark

delete()

Deletes the physical mark from the buffer. All instances referencing the same mark will no longer be valid. If you haven't given a name to the mark in the call to `GPS.EditorLocation.create_mark()`, it will automatically be destroyed when the last instance referencing it goes out of scope. Therefore, calling `delete()` is not mandatory in the case of unnamed marks, although it is still recommended

is_present()

Returns True if mark's location is still present in the buffer

location()

Returns the current location of the mark. This location will vary depending on the changes that take place in the buffer

Returns An instance of `GPS.EditorLocation`

```
ed = GPS.EditorBuffer.get(GPS.File("a.adb"))
loc = GPS.EditorLocation(ed, 3, 5)
mark = loc.create_mark()
# ...
loc = mark.location()
```

move (*location*)

Moves the mark to a new location in the buffer. This is slightly less expensive than destroying the mark and creating a new one through `GPS.EditorLocation.create_mark()`, although the result is the same

Parameters *location* – An instance of `GPS.EditorLocation`

15.5.26 `GPS.EditorOverlay`

class `GPS.EditorOverlay`

This class represents properties that can be applied to one or more ranges of text. This can be used to change the display properties of the text (colors, fonts,...) or store any user-specific attributes that can be retrieved later. GPS itself uses overlays to do syntax highlighting. If two or more overlays are applied to the same range of text, the final colors and fonts of the text depends on the priorities of these overlays and the order in which they were applied to the buffer.

This class is fairly low-level, and we recommend using the class `gps_utils.highlighter.OverlayStyle()` instead. That class provides similar support for specifying attributes, but makes it easier to highlight sections of an editor with that style, or to remove the highlighting.

In fact, if your goal is to highlight parts of editors, it might be simpler to use `gps_utils.highlighter.Background_Highlighter()` or one of the classes derived from it. These classes provide convenient support for highlighting editors in the background, i.e. without interfering with the user or slowing things down.

__init__ ()

This subprogram is used to prevent the direct creation of overlays. Overlays need to be created through `GPS.EditorBuffer.create_overlay()`

See Also:

`GPS.EditorBuffer.create_overlay()`

get_property (*name*)

This subprogram is used to retrieve one of the predefined properties of the overlay. This list of these properties is described for `GPS.EditorOverlay.set_property`

Parameters *name* – A string

Returns A string or a boolean, depending on the property

name ()

Return the name associated with this overlay, as given to `GPS.EditorBuffer.create_overlay()`

Returns A string

See Also:

`GPS.EditorBuffer.create_overlay()`

set_property (*name, value*)

This function is used to change some of the predefined properties of the overlay. These are mostly used to change the visual rendering of the text,... The following attribute names are currently recognized:

- *foreground* (value is a string with the color name)
 - Change the foreground color of the text.
- *background* (value is a string with the color name)
 - Change the background color of the text.

- paragraph-background* (value is a string with the color name)
Change the background color of the entire lines. Contrary to the “background” property, this highlights the entire line, including the space after the end of the text, regardless of which characters are actually covered by the overlay.
- font* (value is a string with the font name)
Changes the font of the text
- weight* (value is a string, one of “light”, “normal” and “bold”)
- style* (value is a string, one of “normal”, “oblique” and “italic”)
- editable*** (value is a boolean): Indicates whether this range of text is editable or not
- variant* (one of 0 (“normal”) or 1 (“small_caps”))
- stretch* (from 0 (“ultra-condensed”) to 8 (“ultra-expanded”))
- underline* (one of 0 (“none”), 1 (“single”), 2 (“double”), 3 (“low”))
- size-points* (an integer) Font size in points
- rise* (an integer)
Offset of text above the baseline (below the baseline if rise is negative, in Pango units).
- pixels-above-lines* (an integer)
Pixels of blank space above paragraphs.
- pixels-below-lines* (an integer)
Pixels of blank space below paragraphs.
- pixels-inside-wrap* (an integer)
Pixels of blank space between wrapped lines in a paragraph.
- invisible* (a boolean)
Whether this text is hidden
- strikethrough* (a boolean)
Whether to strike through the text.
- background-full-height* (a boolean)
Whether the background color fills the entire line height or only the height of the tagged characters.

The set of predefined attributes is fixed. However, overlays are especially useful to store your own user data in the usual python manner, which you can retrieve later. This can be used to mark specially specific ranges of text which you want to be able to find easily later on, even if the buffer has been modified since then (see `GPS.EditorLocation.forward_overlay`)

param name A string

param value A string or a boolean, depending on the property

15.5.27 `GPS.EditorView`

class `GPS.EditorView` (*buffer*)

One view of an editor, ie the visible part through which users can modify text files. A given `GPS.EditorBuffer` can be associated with multiple views. Closing the last view associated with a buffer will also close the buffer

```
# To get a handle on the current editor, use the following code:
view = GPS.EditorBuffer.get().current_view()
```

__init__ (*buffer*)

This constructor is called implicitly whenever you create a new view. It creates a new view for the given buffer, and is automatically inserted into the GPS MDI

Parameters *buffer* – An instance of `GPS.EditorBuffer`

buffer ()

Returns the buffer to which the view is attached. Editing the text of the file should be done through this instance

Returns An instance of `GPS.EditorBuffer`

center (*location='location of cursor'*)

Scrolls the view so that the location is centered

Parameters *location* – An instance of `GPS.EditorLocation`

cursor ()

Return the current location of the cursor in this view

Returns An instance of `GPS.EditorLocation`

get_extend_selection ()

Returns A boolean, whether the user is currently performing a selection. This should impact cursor movements (since moving the cursor should extend the selection).

goto (*location, extend_selection*)

Moves the cursor at the given location. Each view of a particular buffer has its own cursor position, which is where characters typed by the user will be inserted. If `extend_selection` is `True`, extend the selection from the current bound to the new location.

Parameters

- **location** – An instance of `GPS.EditorLocation`
- **extend_selection** – A Boolean

is_read_only ()

Whether the view is editable or not. This property is in fact shared by all views of the same buffer.

Returns A boolean

See Also:

`GPS.EditorBuffer.is_read_only()`

set_extend_selection (*extend*)

Sets the mode for cursor movement. When the parameter is true, moving the cursor extends the selection (for instance shift+cursor keys, or the Emacs mode of setting the mark with ctrl-space and then moving the cursor). When the parameter is false, cursor movement cancels the selection.

Parameters *extend* – A boolean

set_read_only (*read_only=True*)

Indicates whether the user should be able to edit interactively through this view. Setting a view Writable/Read Only will also modify the status of the other views of the same buffer.xx

Parameters *read_only* – A boolean

See Also:

```
GPS.EditorBuffer.get_read_only()
```

title (*short=False*)

Returns the view's title, the short title is returned if short is set to True

Parameters **short** – A boolean

15.5.28 GPS.Entity

class `GPS.Entity` (*name, file=None, line=1, column=1, approximate_search_fallback=True*)

Represents an entity from the source, based on the location of its declaration

See Also:

```
GPS.Entity.__init__()
```

```
__init__ (name, file=None, line=1, column=1, approximate_search_fallback=True)
```

Initializes a new instance of the Entity class, from any reference to the entity. The file parameter should only be omitted for a predefined entity of the language. This will only work for languages for which a cross-reference engine has been defined

Parameters

- **name** – A string, the name of the entity
- **file** – An instance of `GPS.File`, in which the entity is referenced
- **line** – An integer, the line at which the entity is referenced
- **column** – An integer, the column at which the entity is referenced
- **approximate_search_fallback** – If True, when the line and column are not exact, this parameter will trigger approximate search in the database (eg. see if there are similar entities in the surrounding lines)

```
>>> GPS.Entity("foo", GPS.File("a.adb"), 10, 23).declaration().file().name()
=> will return the full path name of the file in which the entity "foo",
referenced in a.adb at line 10, column 23, is defined.
```

attributes ()

Return various boolean attributes of the entity: is the entity global, static,...

Returns A `htable`

body (*nth='1'*)

Return the location at which the implementation of the entity is found. For Ada subprograms and packages, this corresponds to the body of the entity. For Ada private types, this is the location of the full declaration for the type. For entities which do not have a notion of body, this returns the location of the declaration for the entity. Some entities have several bodies. This is for instance the case of a separate subprogram in Ada, where the first body just indicates the subprogram is separate, and the second body provides the actual implementation. The `nth` parameter gives access to the other bodies. An exception is raised when there are not at least `nth` bodies.

Parameters **nth** – An integer

Returns An instance of `GPS.FileLocation`

```
entity = GPS.Entity("bar", GPS.File("a.adb"), 10, 23)
body = entity.body()
print "The subprogram bar's implementation is found at " + body.file.name() +
```

called_by (*dispatching_calls=False*)

Display the list of entities that call the entity. The returned value is a dictionary whose keys are instances of Entity calling this entity, and whose value is a list of FileLocation instances where the entity is referenced. This command might take a while to execute, since GPS needs to get the cross-reference information for lots of source files. If `dispatching_calls` is true, then calls to self that might occur through dispatching are also listed.

Parameters `dispatching_calls` – A boolean

Returns A dictionary, see below

called_by_browser ()

Open the call graph browser to show what entities call self

calls (*dispatching_calls=False*)

Display the list of entities called by the entity. The returned value is a dictionary whose keys are instances of Entity called by this entity, and whose value is a list of FileLocation instances where the entity is referenced. If `dispatching_calls` is true, then calls done through dispatching will result in multiple entities being listed (ie all the possible subprograms that are called at that location)

Parameters `dispatching_calls` – A boolean

Returns A dictionary, see below

See Also:

`GPS.Entity.is_called_by()`

category ()

Return the category of a given entity. Possible values include: label, literal, object, subprogram, package, namespace, type, unknown. The exact list of strings is not hard-coded in GPS and depends on the programming language of the corresponding source.

See instead `is_access`, `is_array`, `is_subprogram`,...

Returns A string

declaration ()

Return the location of the declaration for the entity. The file's name is "`<predefined>`" for predefined entities

Returns An instance of `GPS.FileLocation`, where the entity is declared

```
entity=GPS.Entity("integer")
if entity.declaration().file().name() == "<predefined>":
    print "This is a predefined entity"
```

derived_types ()

Return a list of all the entities that are derived from self. For object-oriented languages, this includes types that extend self. In Ada, this also includes subtypes of self.

Returns List of `GPS.Entity`

discriminants ()

Return the list of discriminants for entity. This is a list of entities, empty if the type has no discriminant or if this notion doesn't apply to that language

Returns List of instances of `GPS.Entity`

documentation (*extended=False*)

Return the documentation for the entity. This is the comment block found just before or just after the declaration of the entity (if any such block exists). This is also the documentation string displayed in the

tooltips when you leave the mouse cursor over an entity for a while. If `extended` is true, then the returned documentation will include formatting and full entity description.

Parameters `extended` – A boolean

Returns A string

`end_of_scope()`

Return the location at which the end of the entity is found.

Returns An instance of `GPS.FileLocation`

`fields()`

Return the list of fields for entity. This is a list of entities. This applies to Ada record and tagged types, or C structs for instance.

In older versions of GPS, this used to return the literals for enumeration types, but these should now be queried through `self.literals()` instead.

Returns List of instances of `GPS.Entity`

`find_all_refs(include_implicit=False)`

Display in the location window all the references to the entity. If `include_implicit` is true, then implicit uses of the entity will also be referenced, for instance when the entity appears as an implicit parameter to a generic instantiation in Ada

Parameters `include_implicit` – A boolean

See Also:

`GPS.Entity.references()`

`full_name()`

Return the full name of the entity that it to say the name of the entity prefixed with its callers and parent packages names. The casing of the name has been normalized to lower-cases for case-insensitive languages

Returns A string, the full name of the entity

`is_access()`

Whether `self` is a pointer or access (variable or type) :return: A boolean

`is_array()`

Whether `self` is an array type or variable. :return: A boolean

`is_container()`

Whether `self` contains other entities (like a package or a record for instance). :return: A boolean

`is_generic()`

Whether the entity is a generic. :return: A boolean

`is_global()`

Whether `self` is a global entity. :return: A boolean

`is_predefined()`

Whether `self` is a predefined entity, i.e. an entity for which there is no explicit declaration (like an 'int' in C or an 'Integer' in Ada)

Returns A boolean

`is_subprogram()`

Whether the entity is a subprogram, procedure or function. :return: A boolean

`is_type()`

Whether `self` is a type declaration (as opposed to a variable) :return: A boolean

literals ()

Return the list of literals for an enumeration type.

Returns List of instances of `GPS.Entity`

methods (include_inherited=False)

Return the list of primitive operations (aka methods) for self. This list is not sorted

Parameters `include_inherited` – A boolean

Returns A list of instances of `GPS.Entity`

name ()

Return the name of the entity. The casing of the name has been normalized to lower-cases for case-insensitive languages

Returns A string, the name of the entity

name_parameters (location)

Refactor the code at the location, to add named parameters. This only work if the language has support for such parameters, namely Ada for the time being

Parameters `location` – An instance of `GPS.FileLocation`

```
GPS.Entity("foo", GPS.File("decl.ads")).rename_parameters(
    GPS.FileLocation(GPS.File("file.adb"), 23, 34))
```

parameters ()

Return the list of parameters for entity. This is a list of entities. This applies to subprograms.

Returns List of instances of `GPS.Entity`

parent_types ()

Return the list of parent types when self is a type. For instance, if we have the following Ada code:

```
type T is new Integer;
type T1 is new T;
```

then the list of parent types for T1 is [T].

Returns a list of `GPS.Entity`

pointed_type ()

Return the type pointed to by entity. If self is not a pointer (or an Ada access type), None is returned. This function also applies to variables, and returns the same information as their type would

Returns An instance of `GPS.Entity`

```
## Given the following Ada code:
##   type Int is new Integer;
##   type Ptr is access Int;
##   P : Ptr;
## the following requests would apply:

f = GPS.File("file.adb")
GPS.Entity("P", f).type()           # Ptr
GPS.Entity("P", f).pointed_type()  # Int
GPS.Entity("Ptr", f).pointed_type() # Int
```

primitive_of ()

Return the type for which self is a primitive operation (or a method, in other languages than Ada)

Returns An instance of `GPS.Entity` or None

references (*include_implicit=False, synchronous=True, show_kind=False, in_file=None, kind_in=''*)

List all references to the entity in the project sources. If `include_implicit` is true, then implicit uses of the entity will also be referenced, for instance when the entity appears as an implicit parameter to a generic instantiation in Ada.

If `synchronous` is True, then the result will be directly returned, otherwise a command will be returned and its result will be accessible with `get_result()`. The result, then, is either a list of locations (if `show_kind` is False), or a `htable` indexed by location, and whose value is a string indicating the kind of the reference (declaration, body, label, end-of-spec,...). The parameter `in_file` can be used to limit the search to references in a particular file. This is a lot faster. The parameter `kind_in` is a list of comma-separated list of reference kinds (as would be returned when `show_kind` is True). Only such references are returned, as opposed to all references.

Parameters

- **include_implicit** – A boolean
- **synchronous** – A boolean
- **show_kind** – A boolean
- **in_file** – An instance of `GPS.File`
- **kind_in** – A string

Returns List of `GPS.FileLocation`, `htable` or `GPS.Command`

See Also:

```
GPS.Entity.find_all_refs()
```

```
for r in GPS.Entity("GPS", GPS.File("gps.adb")).references():
    print "One reference in " + r.file().name()
```

rename (*name, include_overriding=True, make_writable=False, auto_save=False*)

Rename the entity every where in the application. The source files should have been compiled first, since this operation relies on the cross-reference information which have been generated by the compiler. If `include_overriding` is true, then subprograms that override or are overridden by self are also renamed. Likewise, if self is a parameter to a subprogram then parameters with the same name in overriding or overridden subprograms are also renamed.

If some renaming should be performed in a read-only file, the behavior depends on the `make_writable` parameter: if true, the file is made writable and the renaming is performed; if false, no renaming is performed in that file, and a dialog is displayed asking whether you want to do the other renamings.

The files will be saved automatically if `auto_save` is true, otherwise they are left edited.

Parameters

- **name** – A string
- **include_overriding** – A boolean
- **make_writable** – A boolean
- **auto_save** – A boolean

return_type ()

Return the return type for entity. This applies to subprograms.

Returns An instance of `GPS.Entity`

show()

Display in the type browser the informations known about the entity: list of fields for records, list of primitive subprograms or methods, list of parameters, ...

type()

Return the type of the entity. For a variable, it is its type. This function used to return the parent types when self is itself a type, but this usage is deprecated and you should be using `self.parent_types()` instead.

Returns An instance of `GPS.Entity`

15.5.29 `GPS.EntityContext`

class `GPS.EntityContext`

Represents a context that contains entity information

See Also:

`GPS.EntityContext.__init__()`

`__init__()`

Dummy function, whose goal is to prevent user-creation of a `GPS.EntityContext` instance. Such instances can only be created internally by GPS

entity (*approximate_search_fallback=True*)

Return the entity stored in the context

Parameters `approximate_search_fallback` – If True, when the line and column are not exact, this parameter will trigger approximate search in the database (eg. see if there are similar entities in the surrounding lines)

Returns An instance of `GPS.Entity`

15.5.30 `GPS.Exception`

class `GPS.Exception`

One of the exceptions that can be raised by GPS. It is a general error message, and its semantic depends on what subprogram raised the exception.

15.5.31 `GPS.File`

class `GPS.File` (*name, local=False*)

Represents a source file of your application

See Also:

`GPS.File.__init__()`

`__init__(name, local=False)`

Initializes a new instance of the class `File`. This doesn't need to be called explicitly, since GPS will call it automatically when you create such an instance. If `name` is a base file name (no directory is specified), then GPS will attempt to search for this file in the list of source directories of the project. If a directory is specified, or the base file name wasn't found in the source directories, then the file name is considered as relative to the current directory. If `local` is "true" the specified file name is to be considered as local to the current directory.

Parameters

- `name` – Name of the file associated with this instance

- **local** – A boolean

See Also:

```
GPS.File.name()

file=GPS.File("/tmp/work")
print file.name()
```

compile (*extra_args*='')

Compile current file. This call will return only once the compilation is completed. Additional arguments can be added to the command line.

Parameters *extra_args* – A string

See Also:

```
GPS.File.make()

GPS.File("a.adb").compile()
```

directory ()

Return the directory in which the file is found

Returns A string

```
## Sorting files by TN is easily done with a loop like
dirs={}
for s in GPS.Project.root().sources():
    if dirs.has_key(s.directory()):
        dirs[s.directory()].append(s)
    else:
        dirs[s.directory()] = [s]
```

entities (*local=True*)

Return the list of entities that are either referenced (if *local* is false) or declared (if *local* is true) in self.

Parameters *local* – A boolean

Returns A list of `GPS.Entity`

generate_doc ()

Generate the documentation of the file, and display it with the default browser

See Also:

```
GPS.Project.generate_doc()
```

get_property (*name*)

Return the value of the property associated with the file. This property might have been set in a previous GPS session if it is persistent. An exception is raised if no such property already exists for the file

Parameters *name* – A string

Returns A string

See Also:

```
GPS.File.set_property()
```

imported_by (*include_implicit=False, include_system=True*)

Return the list of files that depends on *file_name*. This command might take some time to execute since GPS needs to parse the cross-reference information for multiple source files. If *include_implicit* is true, then implicit dependencies are also returned. If *include_system* is true, then system files from the compiler runtime are also returned.

Parameters

- **include_implicit** – A boolean. This is now ignored, and only explicit dependencies corresponding to actual ‘with’ or ‘#include’ lines will be returned.
- **include_system** – A boolean

Returns A list of files

See Also:

`GPS.File.imports()`

imports (*include_implicit=False, include_system=True*)

Return the the list of files that self depends on. If `include_implicit` is true, then implicit dependencies are also returned. If `include_system` is true, then system files from the compiler runtime are also returned.

Parameters

- **include_implicit** – A boolean
- **include_system** – A boolean

Returns A list of files

See Also:

`GPS.File.imported_by()`

language ()

Return the name of the language this file is written in. This is based on the file extension and the naming scheme defined in the project files or the XML files. The empty string is returned when the language is unknown

Returns A string

make (*extra_args=''*)

Compile and link the file and all its dependencies. This call will return only once the compilation is completed. Additional arguments can be added to the command line.

Parameters `extra_args` – A string

See Also:

`GPS.File.compile()`

name (*remote_server='GPS_Server'*)

Return the name of the file associated with self. This is an absolute file name, including directories from the root of the filesystem.

If `remote_server` is set, then the function returns the equivalent path on the specified server. `GPS_Server` (default) is always the local machine.

Parameters `remote_server` – A string. Possible values are “GPS_Server” (or empty string), “Build_Server”, “Debug_Server”, “Execution_Server” and “Tools_Server”.

Returns A string, the name of the file

other_file ()

Return the name of the other file semantically associated with this one. In Ada this is the spec or body of the same package depending on the type of this file. In C, this will generally be the `.c` or `.h` file with the same base name.

Returns An instance of `GPS.File`

```
GPS.File("tokens.ads").other_file().name()
=> will print "/full/path/to/tokens.adb" in the context of the project
=> file used for the GPS tutorial.
```

project (*default_to_root=True*)

Return the project to which file belongs. If file is not one of the sources of the project, the returned value depends on `default_to_none`: if false, None is returned. Otherwise, the root project is returned.

Parameters `default_to_root` – A boolean

Returns An instance of `GPS.Project`

```
GPS.File("tokens.ads").project().name()
=> will print "/full/path/to/sdc.gpr" in the context of the project file
=> used for the GPS tutorial
```

references (*kind=''*, *sortby=0*)

Returns all references (to any entity) within the file. The acceptable values for `kind` can currently be retrieved directly from the cross-references database by using a slightly convoluted approach:

```
sqlite3 obj/gnatinspect.db
> select display from reference_kinds;
```

Parameters

- **kind** (*string*) – this can be used to filter the references, and is more efficient than traversing the list afterward. For instance, you can get access to the list of dispatching calls by passing “dispatching call” for `kind`. The list of kinds is defined in the cross-reference database, and new values can be added at any time. See above on how to retrieve the list of possible values.
- **sortby** (*integer*) – how the returned list should be sorted. 0 indicates that they are sorted in the order in which they appear in the file; 1 indicates that they are sorted first by entity, and then in file order.

Returns a list of tuples (`GPS.Entity`, `GPS.FileLocation`)

remove_property (*name*)

Removes a property associated with a file

Parameters `name` – A string

See Also:

```
GPS.File.set_property()
```

search (*pattern*, *case_sensitive=False*, *regexp=False*, *scope='whole'*)

Return the list of matches for `pattern` in the file. Default values are False for `case_sensitive` and `regexp`. `Scope` is a string, and should be any of ‘whole’, ‘comments’, ‘strings’, ‘code’. The latter will match only for text outside of comments

Parameters

- **pattern** – A string
- **case_sensitive** – A boolean
- **regexp** – A boolean
- **scope** – One of (“whole”, “comments”, “strings”, “code”)

Returns List of `GPS.FileLocation` instances

See Also:

```
GPS.EditorLocation.search()
```

```
GPS.File.search_next()
```

search_next (*pattern, case_sensitive=False, regexp=False*)

Return the next match for *pattern* in the file. Default values are `False` for `case_sensitive` and `regexp`. Scope is a string, and should be any of 'whole', 'comments', 'strings', 'code'. The latter will match only for text outside of comments

Parameters

- **pattern** – A string
- **case_sensitive** – A boolean
- **regexp** – A boolean

Returns An instance of `GPS.FileLocation`

See Also:

```
GPS.File.search_next()
```

set_property (*name, value, persistent=False*)

Associates a string property with the file. This property is retrievable during the whole GPS session, or across GPS sessions if `persistent` is set to `True`.

This is different than setting instance properties through Python's standard mechanism in that there is no guarantee that the same instance of `GPS.File` will be created for each physical file on the disk, and therefore you would not be able to associate a property with the physical file itself

Parameters

- **name** – A string
- **value** – A string
- **persistent** – A boolean

See Also:

```
GPS.File.get_property()
```

```
GPS.Project.set_property()
```

used_by ()

Display in the dependency browser the list of files that depends on *file_name*. This command might take some time to execute since GPS needs to parse the cross-reference information for multiple source files

See Also:

```
GPS.File.uses()
```

uses ()

Display in the dependency browser the list of files that *file_name* depends on.

See Also:

```
GPS.File.used_by()
```

15.5.32 GPS.FileContext

class `GPS.FileContext`

Represents a context that contains file information

See Also:

`GPS.FileContext.__init__()`

`__init__()`

Dummy function, whose goal is to prevent user-creation of a `GPS.FileContext` instance. Such instances can only be created internally by `GPS`

directory ()

Return the current directory in the context

Returns A string

file ()

Return the name of the file in the context

Returns An instance of `GPS.File`

files ()

Return the list of selected files in the context

Returns A list of `GPS.File`

location ()

Return the file location stored in the context

Returns An instance of `GPS.FileLocation`

project ()

Return the project in the context, or the root project if none was specified in the context. Return an error if no project can be determined from the context

Returns An instance of `GPS.Project`

15.5.33 GPS.FileLocation

class `GPS.FileLocation` (*filename, line, column*)

Represents a location in a file

See Also:

`GPS.FileLocation.__init__()`

`__init__(filename, line, column)`

Initializes a new instance of `GPS.FileLocation`.

Parameters

- **filename** – An instance of `GPS.File`
- **line** – An integer
- **column** – An integer

```
location = GPS.FileLocation(GPS.File("a.adb"), 1, 2)
```

column ()

Return the column of the location

Returns An integer, the column of the location

See Also:

`GPS.FileLocation.file()`

`GPS.FileLocation.line()`

file()

Return the file of the location

Returns An instance of `GPS.File`, the file of the location

See Also:

`GPS.FileLocation.line()`

`GPS.FileLocation.column()`

line()

Return the line of the location

Returns An integer, the line of the location

See Also:

`GPS.FileLocation.file()`

`GPS.FileLocation.column()`

15.5.34 GPS.GUI

class `GPS.GUI`

This is an abstract class (ie no instances of it can be created from your code, which represents a graphical element of the GPS interface

See Also:

`GPS.GUI.__init__()`

__init__()

Prevents the creation of instances of `GPS.GUI`. Such instances are created automatically by `GPS` as a result of calling other functions

See Also:

`GPS.Toolbar.append()`

See Also:

`GPS.Toolbar.entry()`

See Also:

`GPS.Menu.get()`

destroy()

Destroy the graphical element. It will disappear from the interface, and cannot necessarily be recreated later on

hide()

Temporarily hide the graphical element. It can be shown again through a call to `GPS.GUI.show()`

See Also:

`GPS.GUI.show()`

is_sensitive()

Return False if the widget is currently greyed out, and is not clickable by users

Returns A boolean

See Also:

`GPS.GUI.set_sensitive()`

pywidget()

This function is only available if GPS was compiled with support for pygobject, and the latter was found at run time. It returns a widget that can be manipulated through the usual PyGtk functions. PyGObject is a binding to the gtk+ toolkit, and allows you to create your own windows easily, or manipulate the entire GPS GUI from python

Returns An instance of PyWidget

See Also:

`GPS.MDI.add()`

```
# The following example makes the project view inactive. One could easily
# change the contents of the project view as well
widget = GPS.MDI.get("Project View")
widget.pywidget().set_sensitive False
```

set_sensitive(sensitive=True)

Indicate whether the associated graphical element should respond to user interaction or not. If the element is not sensitive, then the user will not be able to click on it

Parameters sensitive – A boolean

See Also:

`GPS.GUI.is_sensitive()`

show()

Show again the graphical element that was hidden by hide()

See Also:

`GPS.GUI.hide()`

15.5.35 GPS.HTML

class GPS.HTML

This class gives access to the help system of GPS, as well as to the integrated browser

static add_doc_directory(directory)

Add a new directory to the GPS_DOC_PATH environment variable. This directory is searched for documentation files. If this directory contains a `gps_index.xml` file, it is parsed to find the list of documentation files to add to the Help menu. See the GPS documentation for more information on the format of the `gps_index.xml` files

Parameters directory – Directory that contains the documentation

static browse(URL, anchor='', navigation=True)

Open the GPS html viewer, and load the given URL. If anchor matches a `<a>` tag in this file, GPS will jump to it. If URL isn't an absolute file name, it is searched in the path set by the environment variable `GPS_DOC_PATH`.

If navigation is True, then the URL is saved in the navigation list, so that users can move back and forward from and to this location later on.

The URL can be a network file name, with the following general format:

```
protocol://username@host:port/full/path
```

where protocol is one of the recognized protocols (http, ftp,.. see the GPS documentation), and the user-name and port are optional.

Parameters

- **URL** – Name of the file to browse
- **anchor** – Location in the file where to jump to
- **navigation** – A boolean

See Also:

```
GPS.HTML.add_doc_directory()
```

```
GPS.HTML.browse("gps.html")
```

=> will **open** the GPS documentation **in** the internal browser

```
GPS.HTML.browse("http://host.com/my/document")
```

=> will download documentation **from the web**

15.5.36 GPS.Help

class GPS.Help

This class gives access to the external documentation for shell commands. This external documentation is stored in the file shell_commands.xml, part of the GPS installation, and is what you are currently seeing.

You almost never need to use this class yourself, since it is used implicitly by Python when you call the help(object) command at the GPS prompt.

The help browser understands the standard http urls, with links to specific parts of the document. For instance:

```
"http://remote.com/my_document"
or "#link"
```

As a special case, it also supports links starting with '%'. These are shell commands to execute within GPS, instead of a standard html file. For instance:

```
<a href="%shell:Editor.edit g-os_lib.ads">GNAT.OS_Lib</a>;
```

The first word after '%' is the language of the shell command, the rest of the text is the command to execute

See Also:

```
GPS.Help.__init__()
```

```
__init__()
```

Initializes the instance of the Help class. This parses the XML file that contains the description of all the commands. With python, the memory occupied by this XML tree will be automatically freed. However, with the GPS shell you need to explicitly call GPS.Help.reset()

See Also:

```
GPS.Help.reset()
```

file()

Return the name of the file that contains the description of the shell commands. You shouldn't have to access it yourself, since you can do so through `GPS.Help().getdoc()` instead

Returns A string

See Also:

`GPS.Help.getdoc()`

getdoc (*name*, *html=False*)

Search, into the XML file `shell_commands.xml`, the documentation for this specific command or entity. If no documentation is found, an error is raised. If `html` is true, the documentation is formatted in HTML

Parameters

- **name** – The fully qualified name of the command
- **html** – A boolean

Returns A string, containing the help for the command

```
print GPS.Help().getdoc("GPS.Help.getdoc")
```

```
Help
Help.getdoc %1 "GPS.Help.getdoc"
Help.reset %2
```

reset()

Free the memory occupied by this instance. This frees the XML tree that is kept in memory. As a result, you can no longer call `GPS.Help.getdoc()` afterward.

15.5.37 GPS.Hook

class `GPS.Hook` (*name*)

General interface to hooks. Hooks are commands executed when some specific events occur in GPS, and allow you to customize some of the aspects of GPS

See Also:

`GPS.Hook.__init__()`

The available hooks are:

- `activity_checked_hook(hookname)`

Hook called when an activity has been checked, this is the last step done after the activity has been committed. It is at this point that the activity closed status is updated.

- `after_character_added(hookname, file, character)`

Hook called when a character has been added in the editor. This hook is also called for the backspace key.

param file An instance of `GPS.File`

param character A character

See Also:

Hook: `character_added`

Hook: `word_added`

- `annotation_parsed_hook(hookname)`

Hook called when the last file annotation has been parsed after the corresponding VCS action.

- before_exit_action_hook(hookname)

This hook is called when GPS is about to exit. If it returns 0, this exit will be prevented (it is recommended that you display a dialog to explain why, in such a case)

return A boolean

- before_file_saved(hookname, file)

Hook called right before a file is saved

param file An instance of GPS.File

- bookmark_added(hookname, bookmark_name)

Hook called when a new bookmark has been created by the user

param bookmark_name A string, the name of the bookmark that has been added

- bookmark_removed(hookname, bookmark_name)

Hook called when a new bookmark has been removed by the user

param bookmark_name A string, the name of the bookmark that has been removed

- buffer_edited(hookname, file)

Hook called after the user has stopped modifying the contents of an editor

param file An instance of GPS.File

- build_server_connected_hook(hookname)

Hook called when GPS connects to the build server in remote mode

- character_added(hookname, file, character)

Hook called when a character is going to be added in the editor. It is also called when a character is going to be removed, in which case the last parameter is 8 (control-h)

param file An instance of GPS.File

param character A character

See Also:

Hook after_character_added

Hook word_added

- clipboard_changed(hookname)

Hook called when the contents of the clipboard has changed, either because the user has done a Copy or Cut operation, or because he called Paste Previous which changes the current entry in the multi-level clipboard.

- commit_done_hook(hookname)

Hook called when a commit has been done.

- compilation_finished(hookname, category, target_name, mode_name, status)

Hook called when a compile operation has finished.

Among the various tasks that GPS connects to this hook are the automatic reparsing of all xref information, and the activation of the automatic-error fixes

See also the hook “xref_updated”.

param category A string, the location/highlighting category that contains the compilation output.

param target_name A string, name of the executed build target.

param mode_name A string, name of the executed build mode.

param status An integer, exit status of the executed program.

•`compilation_starting(hookname, category, quiet, shadow)`

Hook called when a compile operation is about to start.

Among the various tasks that GPS connects to this hook are: check whether unsaved editors should be saved (asking the user), and stop the background task that parses all xref info. If quiet is True, then no visible modification should be done in the MDI, like raising consoles, clearing their content,..., since the compilation should happen in background mode.

Functions connected to this hook should return False if the compilation should not occur for some reason, True if it is OK to start the compilation. Typically, the reason to reject a compilation would be because the user has explicitly cancelled it through a graphical dialog, or because running a background compilation is not suitable at this time.

param category A string, the location/highlighting category that contains the compilation output.

param quiet A boolean, if True then the GUI should advertise the compilation, otherwise nothing should be reported to the user, unless there is an error.

param shadow A boolean, indicates whether the build launched was a Shadow build, ie a “secondary” build launched automatically by GPS after a “real” build. For instance, when the multiple toolchains mode is activated, the builds generating cross-references are Shadow builds.

return A boolean

```
# The following code adds a confirmation dialog to all
# compilation commands.
def on_compilation_started(hook, category, quiet, shadow):
    if not quiet:
        return MDI.yes_no_dialog("Confirm compilation ?")
    else:
        return True
```

```
Hook("compilation_starting").add(on_compilation_started)
```

```
# If you create a script to execute your own build script, you
# should always do the following as part of your script. This
# ensures a better integration in GPS (saving unsaved editors,
# reloading xref information automatically in the end, raising
# the GPS console, parsing error messages for automatically
# fixable errors,...)
```

```
if notHook("compilation_starting").run_until_failure(
    "Builder results", False, False):
    return
```

```
# ... spawn your command
```

```
Hook("compilation_finished").run("Builder results")
```

•`compute_build_targets(hookname, name)`

Hook called whenever GPS needs to compute a list of subtargets for a given build target. The handler should check whether name is a known build target, and if so, return a list of tuples, where each tuple corresponds to one target and contains a display name (used in the menus, for instance) and the name of the target. If name is not known, it should return an empty list.

param name A string, the target type

return A string

```
def compute_targets(hook, name):
    if name == "my_target":
        return [(display_name_1, target_1),
                (display_name_2, target_2)]
    return ""
GPS.Hook("compute_build_targets").add(compute_targets)
```

•context_changed(hookname, context)

Hook called when the current context changes in GPS, ie a new file is selected, or a new entity, or a new window,...

param context An instance of `GPS.Context`

•contextual_menu_close(hookname)

Hook called just before a contextual menu is destroyed. At this time, the value returned by `GPS.contextual_context()` is still the one used in the hook `contextual_menu_open`, and therefore you can still reference the data you stored in the context. This hook is called even if no action was selected by the user. However, it is always called before the action is executed, since the menu itself is closed first.

See Also:

`contextual_menu_open hook()`

•contextual_menu_open(hookname)

Hook called just before a contextual menu is created. It is called before any of the filters is evaluated, and can be used to precomputed data shared by multiple filters to speed up the computation. Use `GPS.contextual_context()` to get the context of the contextual menu and store precomputed data in it.

See Also:

`contextual_menu_close hook()`

•debugger_breakpoints_changed(hookname, debugger)

Hook called when the list of breakpoints has been refreshed. This might occur whether or not the list has changed, but is a good time to refresh any view that might depend on an up-to-date list

param debugger An instance of `GPS.Debugger`

•debugger_command_action_hook(hookname, debugger, command)

This hook is emitted when the user types a command in the debugger console, or emits the console through the `GPS.Debugger` API. It gives you a chance to override the behavior for the command, or even define your own commands. Note that you must ensure that any debugger command you execute this way does finish with a prompt. The function should return the output of your custom command

param debugger An instance of `GPS.Debugger`

param command A string, the command the user wants to execute

return A boolean

```
## The following example implements a new gdb command, "hello". When the
## user types this command in the console, we end up executing "print A"
## instead. This can be used for instance to implement convenient
## macros
```

```
def debugger_commands(hook, debugger, command):
    if command == "hello":
        return 'A=' + debugger.send("print A", False)
    else:
        return ""
```

```
GPS.Hook("debugger_command_action_hook").add(debugger_commands)
```

- `debugger_context_changed(hookname, debugger)`

Called when the debugger context has changed, for instance after the user has switched the current thread, has selected a new frame,...

param debugger An instance of `GPS.Debugger`

- `debugger_executable_changed(hookname, debugger)`

Called when the file being debugged has changed

param debugger An instance of `GPS.Debugger`

- `debugger_process_stopped(hookname, debugger)`

Called when the debugger ran and has stopped, for instance when hitting a breakpoint, or after a next command. If you need to know when the debugger just started processing a command, you can connect to the `debugger_state_changed` hook instead. Conceptually, you could connect to `debugger_state_changed` at all times instead of `debugger_process_stopped` and check when the state is now “idle”

param debugger An instance of `GPS.Debugger`

See Also:

Hook `debugger_state_changed`

- `debugger_process_terminated(hookname, debugger)`

Called when the program being debugged has terminated

param debugger An instance of `GPS.Debugger`

- `debugger_question_action_hook(hookname, debugger, question)`

Action hook called just before displaying an interactive dialog, when the debugger is asking a question to the user. This hook can be used to disable the dialog (and send the reply directly to the debugger instead). It should return a non-empty string to pass to the debugger if the dialog should not be displayed. You cannot send commands to the debugger when inside this hook, since the debugger is blocked waiting for an answer

param debugger An instance of `GPS.Debugger`

param question A string

return A string

```
def gps_question(hook, debugger, str):
    return "1"    ## Always choose choice 1
```

```
GPS.Hook("debugger_question_action_hook").add(gps_question)
```

```
debug=GPS.Debugger.get ()
deubg.send("print &foo")
```

- `debugger_started(hookname, debugger)`

Hook called when a new debugger has been started

param debugger An instance of `GPS.Debugger`

See Also:

Hook `debugger_state_changed`

- `debugger_state_changed(hookname, debugger, new_state)`

Indicates a change in the status of the debugger: `new_state` can be one of “none” (the debugger is now terminated), “idle” (the debugger is now waiting for user input) or “busy” (the debugger is now processing a command, and the process is running). As opposed to `debugger_process_stopped`, this hook is called when the command is just starting its executing (hence the debugger is busy while this hook is called, unless the process immediately stopped).

This hook is in fact emitted also when internal commands are sent to the debugger, and thus much more often than if it was just reacting to user input. It is therefore recommended that the callback does the minimal amount of work, possibly doing the rest of the work in an idle callback to be executed when GPS is no longer busy.

If the new state is “busy”, you cannot send additional commands to the debugger.

When the state is either “busy” or “idle”, `GPS.Debugger.command` will return the command that is about to be executed or the command that was just executed and just completed.

param debugger An instance of `GPS.Debugger`

param new_state A string

- `debugger_terminated(hookname, debugger)`

Hook called when the debugger session has been terminated. It is now recommended that you connect to the `debugger_state_changed` hook and test whether the new state is “none”.

param debugger An instance of `GPS.Debugger`

See Also:

Hook `debugger_state_changed`

- `diff_action_hook(hookname, vcs_file, orig_file, ref_file, diff_file, title)`

Hook called to request the display of the comparison window

param vcs_file An instance of `GPS.File`

param orig_file An instance of `GPS.File`

param ref_file An instance of `GPS.File`

param diff_file An instance of `GPS.File`

param title Buffer title

return A boolean

- `file_changed_detected(hookname, file)`

Hook called whenever GPS detects that an opened file changed on the disk. You can connect to this hook if you want to change the default behavior, which is asking if the user wants to reload the file. Your function should return 1 if the action is handled by the function, and return 0 if the default behavior is desired.

param file An instance of `GPS.File`

return A boolean

```
import GPS
```

```
def on_file_changed(hook, file):  
    # automatically reload the file without prompting the user  
    ed = GPS.EditorBuffer.get(file, force = 1)  
    return 1
```

```
# install a handler on "file_changed_detected" hook  
GPS.Hook("file_changed_detected").add(on_file_changed)
```

•`file_changed_on_disk(hookname, file)`

Hook called when some external action has changed the contents of a file on the disk, such as a VCS operation. The parameter might be a directory instead of a file, indicating that any file in that directory might have changed

param file An instance of `GPS.File`

•`file_closed(hookname, file)`

Hook called just before the last editor for a file is closed. You can still use `EditorBuffer.get()` and `current_view()` to access the last editor for file.

param file An instance of `GPS.File`

•`file_deleted(hookname, file)`

Hook called whenever GPS detects that a file was deleted on the disk. The parameter might be a directory instead of a file, indicating that any file within that directory has been deleted.

param file An instance of `GPS.File`

•`file_edited(hookname, file)`

Hook called when a file editor has been opened for a file that wasn't already opened before. Do not confuse with the hook `open_file_action`, which is used to request the opening of a file.

param file An instance of `GPS.File`

See Also:

`open_file_action hook()`

•`file_line_action_hook(hookname, identifier, file, every_line, normalize)`

Hook called to request the display of new information on the side of the editors. It isn't expected that you connect to this hook, but you might want to run it yourself to ask GPS to display some information on the side of its editors

param identifier A string

param file An instance of `GPS.File`

param every_line A boolean

param normalize A boolean

return A boolean

•`file_renamed(hookname, file, renamed)`

Hook called whenever a GPS action renamed a file on the disk. The file parameter indicates the initial location of the file, while the renamed parameter indicates the new location. The parameters might be

directories instead of files, indicating that the directory has been renamed, and thus any file within that directory have their path changed.

param file An instance of `GPS.File`

param renamed An instance of `GPS.File`

•`file_saved(hookname, file)`

Hook called whenever a file has been saved

param file An instance of `GPS.File`

•`file_status_changed_action_hook(hookname, file, status)`

Hook called when a file status has changed

param file An instance of `GPS.File`

param status A string, the new status for the file. This is the status has displayed into the GPS status line. The value is either Unmodified, Modified or Saved.

return A boolean

•`gps_started(hookname)`

Hook called when GPS is fully loaded, and its window is visible to the user.

It isn't recommended to do any direct graphical action before this hook has been called, so it is recommended that in most cases your start scripts connect to this hook.

•`html_action_hook(hookname, url_or_file, enable_navigation, anchor)`

Hook called to request the display of HTML files. It is generally useful if you want to open an HTML file, and let GPS handle it in the usual manner

param url_or_file A string

param enable_navigation A boolean

param anchor A string

return A boolean

•`location_action_hook(hookname, identifier, category, file, line, column, message)`

Hook called to request the display of new information on the side of the location window

param identifier A string

param category A string

param file An instance of `GPS.File`

param line An integer

param column An integer

param message A string

return A boolean

•`location_changed(hookname, file, line, column)`

Hook called when the location in the current editor has changed, and the cursor has stopped moving.

param file An instance of `GPS.File`

param line An integer

param column An integer

•**log_parsed_hook**(hookname)

Hook called when the last file log has been parsed after the corresponding VCS action.

•**marker_added_to_history**(hookname)

Hook called when a new marker is added to the history list of previous locations, where the user can navigate back and forward

•**open_file_action_hook**(hookname, file, line, column, column_end, enable_navigation, new_file, force_reload, focus=False)

This hook is called when GPS needs to open a file. You can connect to this hook if you want to have your own editor open, instead of the internal editor of GPS. Your function should return 1 if it did open the file, 0 if the next function connected to this hook should be called.

The file should be opened directly at line and column. If column_end is not 0, the given range should be highlighted if possible. The enable_navigation parameter is set to True if the new location should be added to the history list, so that the user can navigate forward and backward across previous locations. new_file is set to True if a new file should be created when file is not found. If set to False, nothing should be done. force_reload is set to true if the file should be reloaded from the disk, discarding any change the user might have done. focus is set to true if the open editor should be given the keyboard focus

param file An instance of `GPS.File`

param line An integer

param column An integer

param column_end An integer

param enable_navigation A boolean

param new_file A boolean

param force_reload A boolean

param focus A boolean

return A boolean

See Also:

`file_edited hook()`

```
GPS.Hook('open_file_action_hook').run(  
    GPS.File("gps-kernel.ads"),  
    322, # line  
    5, # column  
    9, # column_end  
    1, # enable_navigation  
    1, # new_file  
    0) # force_reload
```

•**preferences_changed**(hookname)

Hook called when the value of some of the preferences changes. Modules should refresh themselves dynamically

•**project_changed**(hookname)

Hook called when the project has changed. A new project has been loaded, and all previous settings and caches are now obsolete. In the callbacks for this hook, the attribute values have not been computed from

the project yet, and will only return the default values. Connect to the `project_view_changed` hook instead to query the actual values

See Also:

Hook `project_view_changed`

- `project_changing(hookname, file)`

Hook called just before a new project is loaded.

param file An instance of `GPS.File`

- `project_editor(hookname)`

Hook called before the Project Editor is opened. This allows a custom module to perform specific actions before the actual creation of this dialog.

- `project_saved(hookname, project)`

Hook called when a project is saved to disk. It is called for each project in the hierarchy

param project An instance of `GPS.Project`

- `project_view_changed(hookname)`

Hook called when the project view has been changed, for instance because one of the environment variables has changed. This means that the list of directories, files or switches might now be different. In the callbacks for this hook, you can safely query the new attribute values.

- `revision_parsed_hook(hookname)`

Hook called when the last file revision has been parsed after the corresponding VCS action.

- `rsync_action_hook(hookname)`

For internal use only

- `search_functions_changed(hookname)`

Hook called when the list of registered search functions changes.

- `search_regexps_changed(hookname)`

Hook called when a new regexp has been added to the list of predefined search patterns

- `search_reset(hookname)`

Hook called when the current search pattern is reset or changed by the user, or when the current search is no longer possible because the setup of GPS has changed.

- `server_config_hook(hookname, server_type, nickname)`

Hook called when a server is assigned to a server operations category.

param server_type A string, the server operations category. Can take the values “`BUILD_SERVER`”, “`EXECUTION_SERVER`” or “`DEBUG_SERVER`”

param nickname A string, the server’s nickname

- `server_list_hook(hookname)`

Hook called when the list of configured servers changed.

- `source_lines_revealed(hookname, context)`

Hook called when a range of line becomes visible on the screen

param context An instance of `GPS.Context`

- `status_parsed_hook(hookname)`

Hook called when the last file status has been parsed after the corresponding VCS action.

- `stop_macro_action_hook(hookname)`

You should run this hook to request that the macro currently being replayed be stopped. No more events should be processed as part of this macro

- `task_started(task)`

- `task_changed(task)`

- `task_terminated(task)`

These hooks are called when a new background task is started, when an existing task's status is changed (either it has made progressed, was paused,...), or when a task is terminated. `task_changed` might be called asynchronously; for efficiency, GPS will not necessary emit it for every progress made by the task when this progress is fast.

Each of these hooks receives an instance of `GPS.Task` as parameter.

- `variable_changed(hookname)`

Hook called when one of the scenario variables has been renamed, removed or when one of its possible values has changed.

- `xref_updated(hookname)`

Hook called when the cross-reference information have been updated.

- `word_added(hookname, file)`

Hook called when a word has been added in the editor

param file An instance of `GPS.File`

See Also:

Hook `character_added`

`__init__` (*name*)

Create a new hook instance, which refers to one of the already defined hooks

Parameters *name* – A string, the name of the hook

add (*function_name*, *last=True*)

Connect a new function to a specific hook. Any time this hook is run through `run_hook`, this function will be called with the same parameters passed to `run_hook`. If `Last` is `True`, then this function will be called after all functions currently added to this hook. If `Last` is `False`, it will be called before.

Parameters

- **function_name** – A subprogram, see the “Subprogram Parameters” section in the GPS documentation
- **last** – A boolean

See Also:

`GPS.Hook.remove()`

An example using the GPS shell:

```
# in the GPS shell:
```

```
parse_xml '<action name="edited"><shell>echo "File edited hook=$1 file=$2"</shell></action>'
```

```

Hook "file_edited"
Hook.add %1 "edited"

def filed_edited(hook_name, file):
    print "File edited (hook=" + hook_name + " file=" + file.name()
GPS.Hook("file_edited").add(file_edited)

```

describe_functions ()

List all the functions that are executed when the hook is executed. The returned list might contain <<internal> strings, which indicate that some Ada function is connected to this hook

Returns A list of strings

static list ()

List all defined hooks. See also `run_hook`, `register_hook` and `add_hook`

Returns A list of strings

See Also:

```
GPS.Hook.list_types()
```

static list_types ()

List all defined type hooks

Returns A list of strings

See Also:

```
GPS.Hook.register()
```

static register (name, type='')

Defines a new hook. This hook can take any number of parameters, the default is none. The type and number of parameters is called the type of the hook, and this is described by the optional second parameter. The value of this parameter should be either the empty string for a hook that doesn't take any parameter. Or it could be one of the predefined types exported by GPS itself (see `list_hook_types`). Finally, it could be the word "generic" if this is a new type of hook purely defined for this scripting language

Parameters

- **name** – A string, the name of the hook to create
- **type** – A string, the type of the hook. See `GPS.Hook.list_types()`

remove (function_name)

Remove `function_name` from the list of functions executed when the hook is run. This is the reverse of `GPS.Hook.add`

Parameters `function_name` – A subprogram, see the "Subprogram Parameters" section in the GPS documentation

See Also:

```
GPS.Hook.add()
```

run (*args)

Run the hook. This will call all the functions that attached to that hook, and return the return value of the last callback (this depends on the type of the hook, most often this is always None). When the callbacks for this hook are expected to return a boolean, this command stops as soon as one the callbacks returns True

Parameters `args` – Any number of parameters to pass to the hook.

See Also:`GPS.Hook.run_until_success()``GPS.Hook.run_until_failure()`**run_until_failure** (*args)

This only applies to hooks returning a boolean. This executes all functions attached to this hook, until one returns False, in which case no further function is called. This returns the returned value of the last executed function.

Parameters `args` – Any number of parameters to pass to the hook.

Returns A boolean

See Also:`GPS.Hook.run_until_success()``GPS.Hook.run()`**run_until_success** (*args)

This only applies to hooks returning a boolean. This executes all functions attached to this hook, until one returns True, in which case no further function is called. This returns the returned value of the last executed function. This is mostly the same as `GPS.Hook.run`, but makes the halt condition more explicit.

Parameters `args` – Any number of parameters to pass to the hook.

Returns A boolean

See Also:`GPS.Hook.run_until_failure()``GPS.Hook.run()`

15.5.38 `GPS.Invalid_Argument`

class `GPS.Invalid_Argument`

An exception raised by GPS. Raised when calling a subprogram from the GPS module with an invalid argument type (passing an integer when a string is expected, for instance)

15.5.39 `GPS.Locations`

class `GPS.Locations`

General interface to the locations window

static add (*category, file, line, column, message, highlight='', length='0', look_for_secondary=False*)

Add a new entry in the location window. Nodes are created as needed for the category or file. If `Highlight` is specified to a non-empty string, the whole line is highlighted in the file, with a color given by that highlight category (see `register_highlighting` for more information). `Length` is the length of the highlighting. The default value of 0 indicates that the whole line should be highlighted

Parameters

- **category** – A string
- **file** – An instance of `GPS.File`
- **line** – An integer
- **column** – An integer

- **message** – A string
- **highlight** – A string, the name of the highlight category
- **length** – An integer
- **look_for_secondary** – A boolean

```
GPS.Editor.register_highlighting("My_Category", "blue")
GPS.Locations.add(category="Name in location window",
                  file=GPS.File("foo.c"),
                  line=320,
                  column=2,
                  message="message",
                  highlight="My_Category")
```

static dump (*file*)

Dump the contents of the Locations View to the specified file, in XML format.

Parameters **file** – A string

static list_categories ()

Return the list of all categories currently displayed in the Locations window. These are the top-level nodes used to group information generally related to one command, like the result of a compilation.

Returns A list of strings

See Also:

`GPS.Locations.remove_category()`

static list_locations (*category*, *file*)

Return the list of all file locations currently listed in the given category and file.

Parameters

- **category** – A string
- **file** – A string

Returns A list of EditorLocation

See Also:

`GPS.Locations.remove_category()`

static parse (*output*, *category*, *regexp=''*, *file_index=-1*, *line_index=-1*, *column_index=-1*, *msg_index=-1*, *style_index=-1*, *warning_index=-1*, *highlight_category='Builder results'*, *style_category='Style errors'*, *warning_category='Builder warnings'*)

Parse the contents of the string, which is supposedly the output of some tool, and add the errors and warnings to the locations window. A new category is created in the locations window if it doesn't exist. Preexisting contents for that category is not removed, see `locations_remove_category`.

The regular expression specifies how locations are recognized. By default, it matches `file:line:column`. The various indexes indicate the index of the opening parenthesis that contains the relevant information in the regular expression. Set it to 0 if that information is not available. `style_index` and `warning_index`, if they match, force the error message in a specific category.

`highlight_category`, `style_category` and `warning_category` reference the colors to use in the editor to highlight the messages when the regexp has matched. If they are set to the empty string, no highlighting is done in the editor. The default values match those by GPS itself to highlight the error messages. Create these categories with `GPS.Editor.register_highlighting()`.

Parameters

- **output** – A string
- **category** – A string
- **regexp** – A string
- **file_index** – An integer
- **line_index** – An integer
- **column_index** – An integer
- **msg_index** – An integer
- **style_index** – An integer
- **warning_index** – An integer
- **highlight_category** – A string
- **style_category** – A string
- **warning_category** – A string

See Also:

`GPS.Editor.register_highlighting()`

static remove_category (*category*)

Remove a category from the location window. This removes all associated files

Parameters *category* – A string

See Also:

`GPS.Locations.list_categories()`

static set_sort_order_hint (*category*)

Sets desired sorting order for file nodes of the category. Actual sort order can be overridden by user.

Parameters *category* – A string (“Chronological” or “Alphabetical”)

15.5.40 `GPS.Logger`

class `GPS.Logger` (*name*)

This class provides an interface to the GPS logging mechanism. This can be used when debugging scripts, or even be left in production scripts for post-mortem analysis for instance. All output through this class is done in the GPS log file, in `$HOME/.gps/log`.

GPS comes with some predefined logging streams, which can be used to configure the format of the log file, such as whether colors should be used, whether timestamps should be logged with each message,...

active = True

Whether this logging stream is active

count = None

__init__ (*name*)

Create a new logging stream. Each stream is associated with a name, which is displayed before each line in the GPS log file, and is used to distinguish between various parts of GPS. Calling this constructor with the same name multiple times will create a new class instance.

Parameters *name* – A string

```
log = GPS.Logger("my_script")
log.log("A message")
```

check (*condition*, *error_message*, *success_message*='')

If condition evaluates to False, then error_message will be logged in the log file. If the condition evaluates to True, then success_message is logged if it was specified

Parameters

- **condition** – A boolean
- **error_message** – A string
- **success_message** – A string

```
log=GPS.Logger("my_script")
log.check(1 == 2, "Invalid addition")
```

log (*message*)

Logs a message in the GPS log file

Parameters *message* – A string

set_active (*active*)

Activate or deactivate a logging stream. The default for a stream depends on the file \$HOME/.gps/traces.cfg, and will generally be active. When a stream is inactive, no message is sent to the log file

Use self.active to test whether a log stream is active.

Parameters *active* – A boolean

15.5.41 GPS.MDI

class `GPS.MDI`

Represents GPS's Multiple Document Interface. This gives access to general graphical commands for GPS, as well as control over the current layout of the windows within GPS

See Also:

`GPS.MDIWindow`

If you have installed the pygobject package (see GPS's documentation), GPS will export a few more functions to python so that it is easier to interact with GPS itself. In particular, the `GPS.MDI.add` function allows you to put a widget created by pygobject under control of GPS's MDI, so that users can interact with it as with all other GPS windows.

```
import GPS
```

```
## The following line is the usual way to make pygobject visible
from gi.repository import Gtk, GLib, Gdk, GObject
```

```
def on_clicked(*args):
    GPS.Console().write("button was pressed\n")
```

```
def create():
    button=Gtk.Button('press')
    button.connect('clicked', on_clicked)
    GPS.MDI.add(button, "From testgtk", "testgtk")
    win = GPS.MDI.get('testgtk')
```

```
win.split()

create()

GROUP_CONSOLES = 0
GROUP_DEBUGGER_DATA = 0
GROUP_DEBUGGER_STACK = 0
GROUP_DEFAULT = 0
GROUP_GRAPHS = 0
GROUP_VCS_ACTIVITIES = 0
GROUP_VCS_EXPLORER = 0
GROUP_VIEW = 0
POSITION_AUTOMATIC = 0
POSITION_BOTTOM = 0
POSITION_LEFT = 0
POSITION_RIGHT = 0
POSITION_TOP = 0

static add (widget, title='', short='', group=0, position=0, save_desktop=None)
```

This function is only available if pygobject could be loaded in the python shell. You must install this library first, see the documentation for GPS.MDI itself.

This function adds a widget inside the MDI of GPS. The resulting window can then be manipulated by the user like any other standard GPS window. It can be split, floated, resized,... Title is the string used in the title bar of the window, short is the string used in the notebook tabs. You can immediately retrieve a handle to the created window by calling GPS.MDI.get (short).

Parameters

- **widget** – A widget, created by pygobject
- **title** – A string
- **short** – A string
- **group** – An integer, see the constants MDI.GROUP_*. This indicates to which logical group the widget belongs (the default group should be reserved for editors). You can create new groups as you see fit.
- **position** – An integer, see the constants MDI.POSITION_*. It is used when no other widget of the same group exists, to specify the initial location of the newly created notebook. When other widgets of the same group exist, the widget is put on top of them.
- **save_desktop** – A function that should be called when GPS saves the desktop into XML. This function receives the GPS.MDIWindow as a parameter, and should return a tuple of two elements (name, data) where name is a unique identifier for this window, and data is a string containing additional data to be saved (and later restored). One suggestion is to encode any python data through json and send the resulting string as data. An easier alternative is to use the modules.py support script in GPS, which handles this parameter automatically on your behalf.

Returns the instance of GPS.MDIWindow that was created

```

from gi.repository import Gtk
b = Gtk.Button("Press Me")
GPS.MDI.add(b)

```

See Also:

`GPS.MDI.get()`

`GPS.GUI.pywidget()`

`GPS.MDI()`

static children()

Return all the windows currently in the MDI

Returns A list of `GPS.MDIWindow`

static current()

Return the window that currently has the focus, or raise an error if there is none

Returns An instance of `GPS.MDIWindow`

static dialog(msg)

Display a modal dialog to report information to a user. This blocks the interpreter until the dialog is closed

Parameters `msg` – A string

static file_selector(file_filter='empty')

Display a modal file selector. The user selected file is returned, or a file with an empty name if 'Cancel' is pressed.

A file filter can be defined (such as `"*.ads"`) to show only a category of files.

Parameters `file_filter` – A string

Returns An instance of `GPS.File`

static get(name)

Return the window whose name is name. If there is no such window, None is returned

Parameters `name` – A string

Returns An instance of `GPS.MDIWindow`

static get_by_child(child)

Return the window that contains child, or raise an error if there is none

Parameters `child` – An instance of `GPS.GUI`

Returns An instance of `GPS.MDIWindow`

static hide()

Hides the graphical interface of GPS.

static input_dialog(msg, *args)

Display a modal dialog and request some input from the user. The message is displayed at the top, and one input field is displayed for each remaining argument. The arguments can take the form `"label=value"`, in which case `"value"` is used as default for this entry. If argument is preprend with 'multiline:' prefix field is edited as multi-line text. The return value is the value that the user has input for each of these parameters.

An empty list is returned if the user presses Cancel

Parameters

- `msg` – A string

- **args** – Any number of strings

Returns A list of strings

```
a, b = GPS.MDI.input_dialog("Please enter values", "a", "b")
print a, b
```

static save_all (*force=False*)

Save all currently unsaved windows. This includes open editors, the project, and any other window that has registered some save callbacks.

If the force parameter is false, then a confirmation dialog is displayed so that the user can select which windows to save

Parameters force – A boolean

static show ()

Shows the graphical interface of GPS.

static yes_no_dialog (*msg*)

Display a modal dialog to ask a question to the user. This blocks the interpreter until the dialog is closed. The dialog has two buttons Yes and No, and the selected button is returned to the caller

Parameters msg – A string

Returns A boolean

```
if GPS.MDI.yes_no_dialog("Do you want to print?"):
    print "You pressed yes"
```

15.5.42 GPS.MDIWindow

class `GPS.MDIWindow`

This class represents one of the windows currently displayed in GPS. This includes both the windows currently visible to the user, and the ones that are temporarily hidden, for instance because they are displayed below another window. Windows acts as containers for other widgets

__init__ ()

Prevents the creation of instances of `GPS.MDIWindow`. This is done by calling the various subprograms in the `GPS.MDI` class

float (*float=True*)

Float the window, ie create a new toplevel window to display it. It is then under control of the user's operating system or window manager. If float is False, the window is reintegrated within the GPS MDI instead

Parameters float – A boolean

get_child ()

Return the child contained in the window. The returned value might be an instance of a subclass of `GPS.GUI`, if that window was created from a shell command

Returns An instance of `GPS.GUI`

```
# Accessing the GPS.Console instance used for python can be done with:
GPS.MDI.get("Python").get_child()
```

is_floating ()

Return whether the window is currently floating (ie in its own toplevel window), or False if the window is integrated into the main GPS window

Returns A boolean

name (*short=False*)

Return the name of the window. If *short* is *False*, the long name is returned, ie the one that appears in the title bar. If *short* is *True*, the short name is returned, ie the one that appears in the notebook tabs.

Parameters *short* – A boolean

Returns A string

next (*visible_only=True*)

Return the next window in the MDI, or window itself if there is no other window. If *visible_only* is *true*, then only the windows currently visible to the user are visible. This always returns floating windows

Parameters *visible_only* – A boolean

Returns An instance of `GPS.MDIWindow`

raise_window ()

Raise the window so that it becomes visible to the user. The window also gains the focus

rename (*name, short=''*)

Change the title used for a window. *name* is the long title, as it appears in the title bar for instance, and *short*, if specified, is the name that appears in the notebook tabs.

Using this function might be dangerous in some contexts, since GPS keeps track of editors through their name.

Parameters

- **name** – A string
- **short** – A string

split (*vertically=True, reuse=False*)

Split the window in two parts, either horizontally (side by side), or vertically (one below the other). If *reuse* is *true*, attempt to reuse an existing space rather than splitting the current window. This should be used to avoid ending up with too small windows

Parameters

- **vertically** – A boolean
- **reuse** – A boolean

See Also:

`GPS.MDIWindow.single()`

15.5.43 GPS.Menu

class `GPS.Menu`

This class is a general interface to the menu system in GPS. It gives you control over which menus should be active, what should be executed when the menu is selected by the user,...

See Also:

`GPS.Menu.__init__()`

`__init__()`

Prevents the creation of a menu instance. Such instances can only be created internally by GPS as a result of calling `GPS.Menu.get` or `GPS.Menu.create`. This is so that you always get the same instance of `GPS.Menu` when you are referring to a given menu in GPS, and so that you can store your own specific data with the menu

static create (*path*, *on_activate*='', *ref*='', *add_before*=True, *filter*=None, *group*='')

Create a new menu in the GPS system. The menu is added at the given location (see `GPS.Menu.get` for more information on the `path` parameter). Submenus are created as necessary so that path is valid.

If `on_activate` is specified, it will be executed every time the user selects that menu. It is called with only one parameter, the instance of `GPS.Menu` that was just created.

If `ref` and `add_before` are specified, they specify the name of another item in the parent menu (and not a full path) before or after which the new menu should be added.

If the name of the menu starts with a '-' sign, as in `"/Edit/"`, then a menu separator is inserted instead. In this case, `on_activate` is ignored.

Underscore characters ('_') need to be duplicated in the path. A single underscore indicates the mnemonic to be used for that menu. For instance, if you create the menu `"/_File"`, then the user can open the menu by pressing alt-F. But the underscore itself will not be displayed in the name of the menu.

If `group` is specified, create a radio menu item in given group.

Parameters

- **path** – A string
- **on_activate** – A subprogram, see the GPS documentation on subprogram parameters
- **ref** – A string
- **add_before** – A boolean
- **filter** – A subprogram
- **group** – A string

Returns The instance of `GPS.Menu`

```
def on_activate(self):  
    print "A menu was selected: " + self.data
```

```
menu = GPS.Menu.create("/Edit/My Company/My Action", on_activate)  
menu.data = "my own data"    ## Store your own data in the instance
```

static get (*path*)

Return the menu found at the given path. Path is similar to what one finds on a hard disk, starting with the main GPS menu ('/'), down to each submenus. For instance, `"/VCS/Directory/Update Directory"` refers to the submenu `'Update Directory'` of the submenu `'Directory'` of the menu `'VCS'`. Path is case-sensitive

Parameters **path** – A string

Returns The instance of `GPS.Menu`

```
# The following example will prevent the user from using the VCS  
# menu and all its entries:
```

```
GPS.Menu.get('/VCS').set_sensitive (False)
```

get_active ()

Return True if the widget is a currently active radio menu item

Returns A boolean

rename (*name*)

Change the name of a menu. The first underscore character seen in `name` will be used as the keyboard shortcut to access this menu from now on. If you actually want to insert an underscore in the name, you need to double it

Parameters name – A string

set_active (*is_active=True*)
Set the active state of a radio menu item

Parameters is_active – A boolean

15.5.44 GPS.Message

class `GPS.Message` (*category, file, line, column, text, flags*)

This class is used to manipulate GPS messages: build errors, editor annotations, etc.

MESSAGE_INVISIBLE = 0

MESSAGE_IN_LOCATIONS = 2

MESSAGE_IN_SIDEBAR = 1

MESSAGE_IN_SIDEBAR_AND_LOCATIONS = 3

The flags returned by `GPS.Message.get_flags()`

__init__ (*category, file, line, column, text, flags*)

Add a Message in GPS.

Parameters

- **category** – A String indicating the message category
- **file** – A File indicating the file
- **line** – An integer indicating the line
- **column** – An integer indicating the column
- **text** – A pango markup String containing the message text
- **flags** – An integer representing the location of the message

```
# Create a message
m=GPS.Message("default", GPS.File("gps-main.adb"),
              1841, 20, "test message", 0)

# Remove the message
m.remove()
```

execute_action ()

If the message has an associated action, execute it.

get_category ()

Return the message's category.

get_column ()

Return the message's column.

get_file ()

Return the message's file.

get_flags ()

Return an integer which represents the location of the message: should it be displayed in locations view and source editor's sidebar. Message is displayed in source editor's sidebar when zero bit is set, and is displayed in locations view when first bit is set, so here is possible values:

- `GPS.Message.MESSAGE_INVISIBLE`: message is invisible

- GPS.Message.MESSAGE_IN_SIDEBAR: message is visible in source editor’s sidebar only
- GPS.Message.MESSAGE_IN_LOCATIONS: message is visible in locations view only
- GPS.Message.MESSAGE_IN_SIDEBAR_AND_LOCATIONS: message is visible in source editor and locations view

Note, set of flags can be extended in the future, thus it is better to handle them as bit set.

get_line ()

Return the message’s line.

get_mark ()

Return an EditorMark which was created with the message and keeps track of the location when the file is edited.

get_text ()

Return the message’s text.

static list (*file=None, category=None*)

Return a list of all messages currently stored in GPS.

Parameters

- **file** – a GPS File. Specifying this parameter restricts the output to messages to this file only.
- **category** – a String. Specifying this parameter restricts the output to messages of this category only

Returns a list of GMS.Message

remove ()

Remove the message from GPS.

set_action (*action, image, tooltip=None*)

Add an action item to the message. This will add an icon to the message, and clicking on this icon will execute action.

Parameters

- **action** – A String corresponding to a registered GPS action.
- **image** – A String corresponding to the id of a registered GPS image. See icons.xml for an example of how to register icons in GPS.
- **tooltip** – A string which contains the tooltip to display when the mouse is on the icon.

static set_sort_order_hint (*category, hint*)

Sets default sorting method for files in Locations view.

Parameters

- **category** – Name of messages category
- **hint** – Default sorting method (“chronological” or “alphabetical”)

set_style (*style, len*)

Set the style of the message. The second parameter indicates the length in number of characters to highlight. If 0, then highlight the whole line. If left out, this means the length of the message highlighting is not modified.

Parameters

- **style** – An integer

- **len** – An integer

set_subprogram (*subprogram, image, tooltip=None*)

Add an action item to the message. This will add an icon to the message, and clicking on this icon will execute the subprogram, with the message passed as parameter of the subprogram.

Parameters

- **subprogram** – A subprogram in the scripting language. This subprogram takes as a parameter one message.
- **image** – A String corresponding to the id of a registered GPS image. See icons.xml for an example of how to register icons in GPS.
- **tooltip** – A string which contains the tooltip to display when the mouse is on the icon.

```
# This adds a "close" button to all the messages
[msg.set_subprogram(lambda m : m.remove(), "gtk-close", "")
 for msg in GPS.Message.list()]
```

15.5.45 GPS.Missing_Arguments

class `GPS.Missing_Arguments`

An exception raised by GPS. Raised when calling a subprogram from the GPS module with missing arguments

15.5.46 GPS.OutputParserWrapper

class `GPS.OutputParserWrapper` (*child=None*)

This class is used to handle user-defined tool output parsers. Parsers are organized in chain. Output of one parser is passed as input to next one. Chain of parser could be attached to a build target. This class is for internal use only. Instead user should inherit custom parser from `OutputParser` defined in `tool_output.py`, but their methods are match.

```
# Here is an example of custom parser:
#
import GPS, tool_output

class PopupParser(tool_output.OutputParser):
    def on_stdout(self, text, command):
        GPS.MDI.dialog(text)
        if self.child != None:
            self.child.on_stdout(text, command)
```

You can attach custom parser to a build target by specify it in XML file

```
<target model="myTarget" category="_Run" name="My Target">
  <output-parsers>[default] popupparses</output-parsers>
</target>
```

Where [default] abbreviates names of all parsers predefined in GPS.

__init__ (*child=None*)

Create a new parser and initialize its child reference if provided.

on_exit (*status, command*)

This method is called when all output is parsed. Its purpose is to flush any buffered data at end of stream.

on_stderr (*text, command*)

This is like `on_stdout`, but concerns error stream.

`on_stdout` (*text, command*)

This method is called each time a portion of output text is ready to parse. It takes the portion of text as parameter and pass filtered portion to its child.

15.5.47 `GPS.Preference`

class `GPS.Preference` (*name*)

Interface to the GPS preferences, as set in the Edit/Preferences dialog. New preferences are created through XML customization files (or calls to `GPS.parse_xml()`, see the GPS documentation)

See Also:

```
GPS.Preference.__init__()
```

```
GPS.parse_xml('''
    <preference name="custom-adb-file-color"
        label="Background color for .adb files"
        page="Editor:Fonts & Colors"
        default="yellow"
        type="color" />''')
print "color is " + GPS.Preference("custom-adb-file-color").get()
```

`__init__` (*name*)

Initializes an instance of the `GPS.Preference` class, associating it with the preference given in parameter. The name is the one that can be found in the `$HOME/.gps/preferences` file. When you are creating a new preference, this name can include `/` characters, which will result in subpages created in the Preferences dialog. The name after the last `/` should only include letters and `-` characters. If the name starts with `/` and contains no other `/`, then the preference will not be visible in the Preferences dialog, although it can be manipulated as usual and will be loaded automatically by GPS on startup.

Parameters *name* – A string

create (*label, type, doc='', default='', *args*)

This function creates a new preference, and makes it visible in the preferences dialog. In the dialog, the preference appears in the page given by the name used when creating the instance of `GPS.Preference`. The label is used to qualify the preference, and *doc* will appear as a tooltip to explain the preference to users. The type describes the type of preference, and therefore how it should be edited by users.

The parameters to this function cannot be named (since it uses a variable number of parameters, see the documentation below).

The additional parameters depend on the type of preference you are creating:

- For an “integer”, the default value is 0, and the two additional parameters are the minimum and maximum possible values. These are integers.
- For a “boolean”, the default is True.
- For a “string”, the default is the empty string.
- A “multiline” behaves the same as a string except it is edited on multiple lines in the Preferences dialog.
- For a “color”, the default is “black”.
- For a “font”, the default is “sans 9”.
- For a “enum”, any number of additional parameters can be specified. They are all the possible values of the preference. The default is the index in the list of possible values, starting at 0.

Parameters

- **label** – A string
- **type** – A string, one of “integer”, “boolean”, “string”, “color”, “font”, “enum”, “multiline”
- **doc** – A string
- **default** – Depends on the type
- **args** – Additional parameters depending on the type

get ()

Get value for the given preference. The exact returned type depends on the type of the preference. Note that boolean values are returned as integers, for compatibility with older versions of Python

Returns A string or an integer

```
if GPS.Preference("MDI-All-Floating") :
    print "We are in all-floating mode"
```

set (value, save=True)

Set value for the given preference. The type of the parameter depends on the type of the preference.

Parameters

- **value** – A string, boolean or integer
- **save** – no longer used, kept for backward compatibility only.

15.5.48 GPS.Process

```
class GPS.Process (command, regexp='', on_match=None, on_exit=None, task_manager=True,
                  progress_regexp='', progress_current=1, progress_total=1, before_kill=None,
                  remote_server='', show_command=False, single_line_regexp=False,
                  case_sensitive_regexp=True, strip_cr=True)
```

Interface to expect-related commands. This class can be used to spawn new processes and communicate with them later on. It is similar to what GPS uses to communicate with gdb. This class is a subclass of GPS.Command.

See Also:

GPS.Process.__init__()

GPS.Command()

```
# The following example launches a gdb process, let it print its welcome
# message, and kills it as soon as a prompt is seen in the output. In
# addition, it displays debugging messages in a new GPS window. As you
# might note, some instance-specific data is stored in the instance of
# the process, and can be retrieve in each callback.
```

```
import GPS, sys
```

```
def my_print(msg):
```

```
    sys.stdout.set_console("My gdb")
```

```
    print(msg)
```

```
    sys.stdout.set_console()
```

```
def on_match(self, matched, unmatched):
```

```
    my_print("on_match (" + self.id + ")=" + matched)
```

```
    self.kill()
```

```
def on_exit(self, status, remaining_output):
    my_print "on_exit (" + self.id + ")"

def run():
    proc = GPS.Process("gdb", "^\\(gdb\\)", on_match=on_match,
                      on_exit=on_exit)
    proc.id = "first session"

run()

# A similar example can be implemented by using a new class. This is
# slightly cleaner, since it doesn't pollute the global namespace.

class My_Gdb(GPS.Process):
    def matched(self, matched, unmatched):
        my_print("matched " + self.id)
        self.kill()

    def exited(self, status, output):
        my_print("exited " + self.id)

    def __init__(self):
        self.id = "from class"
        GPS.Process.__init__(self, "gdb",
                             "^\\(gdb\\)",
                             on_match=My_Gdb.matched,
                             on_exit=My_Gdb.exited)
```

My_Gdb ()

```
__init__(command, regexp=', on_match=None, on_exit=None, task_manager=True,
          progress_regexp=', progress_current=1, progress_total=1, before_kill=None,
          remote_server=', show_command=False, single_line_regexp=False,
          case_sensitive_regexp=True, strip_cr=True)
```

Spawn specified command. Command can include triple-quoted strings, similar to python, which will always be preserved as one argument.

If `regexp` is not-empty and `on_match_action` is specified, launch `on_match_action` when `regexp` is found in the process output. If `on_exit_action` is specified, execute it when the process terminates. Return the ID of the spawned process.

`regexp` is always compiled with the `multi_line` option, so that “^” and “\$” also match at the beginning and end of each line, not just the whole output. You can optionally compile it with the `single_line` option whereby ”.” also matches the newline character. Likewise you can set the `regexp` to be case insensitive by setting `case_sensitive_regexp` to `False`.

`on_match` is a subprogram called with the parameters:

- \$1 = the instance of `GPS.Process`
- \$2 = the string which matched the `regexp`
- \$3 = the string since the last match

`before_kill` is a subprogram called just before the process is about to be killed. It is called when the user is interrupting the process through the task manager, or when `GPS` exits. It is not called when the process terminates normally. When it is called, the process is still valid and can be send commands. Its parameters are:

- \$1 = the instance of `GPS.Process`

- \$2 = the entire output of the process

`on_exit` is a subprogram called when the process has exited. You can no longer send input to it at this stage. Its parameters are:

- \$1 = the instance of `GPS.Process`
- \$2 = the exit status
- \$3 = the output of the process since the last call to `on_match`

If `task_manager` is set to `True`, the process will be visible in the GPS task manager, and can be interrupted or paused by users. Otherwise, it will simply be running in the background, and never visible to the user. If `progress_regexp` is specified, then the output of the process will be scanned for this regexp. The part that match will not be returned to `on_match`. Instead, they will be used to guess the current progress of the command. Two groups of parenthesis are parsed, the one at `progress_current`, and the one at `progress_total`. The number returned for each of these groups indicate the current progress of the command, and the total that must be reached for this command to complete. For instance, if your process outputs lines like “done 2 out of 5”, you should create a regular expression that matches the 2 and the 5 to guess the current progress. As a result, a progress bar is displayed in the task manager of GPS, and will allow users to monitor commands.

`remote_server` represents the server used to spawn the process. By default, the `GPS_Server` is used, which is always the local machine. See the section “Using GPS for Remote Development” in the GPS documentation for more information on this field.

If `show_command` is set, then the command line used to spawn the new Process is displayed in the “Messages” console.

If `strip_cr` is true, the output of the process will have all its ASCII.CR removed before the string is passed on to GPS and your script. This in general provides better portability to Windows systems, but might not be suitable for applications for which CR is relevant (for instance those that drive an ANSI terminal).

An exception is raised if the process could not be spawned.

Parameters

- **command** – A string
- **regexp** – A string
- **on_match** – A subprogram, see the section “Subprogram parameters” in the GPS documentation
- **on_exit** – A subprogram
- **task_manager** – A boolean
- **progress_regexp** – A string
- **progress_current** – An integer
- **progress_total** – An integer
- **before_kill** – A subprogram
- **remote_server** – A string. Possible values are “GPS_Server”, the empty string (equivalent to “GPS_Server”), “Build_Server”, “Debug_Server”, “Execution_Server” and “Tools_Server”.
- **show_command** – A boolean
- **single_line_regexp** – A boolean

- **case_sensitive_regexp** – A boolean
- **strip_cr** – A boolean

See Also:`GPS.Process()`**expect** (*regexp, timeout=-1*)

Block the execution of the script until either `regexp` has been seen in the output of the command, or the timeout has expired. If the timeout is negative, wait forever until we see the `regexp` or the process finishes its execution.

While in such a call, the usual `on_match` callback is called as usual, so you might need to add an explicit test in your `on_match` callback not to do anything in this case.

This command returns the output of the process since the start of the call to `expect` and up to the end of the text that matched `regexp`. Note that it will also include the output that was sent to the `on_match` callback while `expect` was running. It will not however include output already returned by a previous call to `expect` (nor does it guarantee that two successive calls to `expect` will return the full output of the process, since some output might have been matched by `on_match` between the two calls, and would not be returned by the second `expect`).

If a timeout occurred or the process terminated, an exception is raised

Parameters

- **regexp** – A string
- **timeout** – An integer, in milliseconds

Returns A string

```
proc = GPS.Process("/bin/sh")
print("Output till prompt=" + proc.expect(">"))
proc.send("ls")
```

get_result ()

Wait till the process terminates, and return its output. This is the output since the call to `get_result`, ie if you call `get_result` after performing some calls to `expect`, the returned string does not return the output that was already returned by `expect`.

Returns A string**interrupt** ()

Interrupt a process controlled by GPS

kill ()

Terminate a process controlled by GPS

send (*command, add_lf=True*)

Send a line of text to the process. If you need to close the input stream to an external process, it often works to send the character ASCII 4, for instance through the python command `chr(4)`.

Parameters

- **command** – A string
- **add_lf** – A boolean

set_size (*rows, columns*)

Tells the process about the size of its terminal. Rows and columns should (but need not) be the number of visible rows and columns of the terminal in which the process is running.

Parameters

- **rows** – An integer
- **columns** – An integer

wait ()

Block the execution of the script until the process has finished executing. The exit callback registered when the process was started will be called before returning from this function.

This function returns the exit status of the command.

Returns An integer

15.5.49 GPS.Project

class `GPS.Project` (*name*)

Represents a project file. See also the GPS documentation on how to create new project attributes.

See Also:

`GPS.Project.__init__()`

Related hooks:

- “project_view_changed”: Called whenever the project is recomputed, ie one of its attributes was changed by the user, the environment variables are changed,...

Then is a good time to test the list of languages (`GPS.Project.languages()`) that the project supports, and do language-specific customizations

- “project_changed”: A new project was loaded. The hook above will be called after this one

__init__ (*name*)

Initializes an instance of `GPS.Project`. The project must be currently loaded in GPS

Parameters **name** – The project name

See Also:

`GPS.Project.name()`

add_attribute_values (*attribute, package, index, value*)

Add some values to an attribute. You can add as much as many values you need at the end of the param list. If the package is not specified, the attribute at the toplevel of the project is queried. The index only needs to be specified if it applies to that attribute.

Parameters

- **attribute** – A string, the name of the attribute
- **package** – A string, the name of the attribute’s package
- **index** – A string, the name of the index for the specific value of this attribute
- **value** – A string, the name of the first value to add

See Also:

`GPS.Project.set_attribute_as_string()`

`GPS.Project.remove_attribute_values()`

`GPS.Project.clear_attribute_values()`

```
GPS.Project.root().add_attribute_values(  
    "Default_Switches", "Compiler", "ada", "-gnatwa", "-gnatwe");
```

add_dependency (*path*)

This command adds a new dependency from self to the project file pointed to by *path*. This is the equivalent of putting a `with` clause in self, and means that the source files in self can depend on source files from the imported project

Parameters *path* – The path to another project to depend on

See Also:

```
GPS.Project.remove_dependency()
```

add_main_unit (**args*)

Add some main units to the current project, and for the current scenario. The project is not saved automatically

Parameters *args* – Any number of arguments, at least one

static add_predefined_paths (*sources='', objects=''*)

Add some predefined directories to the source path or the objects path. These will be searched when GPS needs to open a file by its base name, in particular from the File->Open From Project dialog. The new paths are added in front, so that they have priorities over previously defined paths.

Parameters

- **sources** – A list of directories separated by the appropriate separator (':' or ';' depending on the system)
- **objects** – As above

```
GPS.Project.add_predefined_paths(os.pathsep.join(sys.path))
```

add_source_dir (*directory*)

Add a new source directory to the project. The new directory is added in front of the source path. You should call `recompute()` after calling this method, to recompute the list of source files. The directory is added for the current value of the scenario variables only. Note that if the current source directory for the project is not specified explicitly in the `.gpr` file, it will be overridden by the new directory you are adding. If the directory is already part of the source directories for the project, it is not added a second time.

Parameters *directory* – A string

See Also:

```
GPS.Project.source_dirs()
```

```
GPS.Project.remove_source_dir()
```

ancestor_deps ()

Return the list of projects that might contain sources that depend on the project's sources. When doing extensive searches it isn't worth checking other projects. Project itself is included in the list.

This is also the list of projects that import self.

Returns A list of instances of `GPS.Project`

```
for p in GPS.Project("kernel").ancestor_deps():  
    print p.name()
```

```
# will print the name of all the projects that import kernel.gpr
```

clear_attribute_values (*attribute, package, index*)

Clear the values list of an attribute.

If the package is not specified, the attribute at the toplevel of the project is queried.

The index only needs to be specified if it applies to that attribute.

Parameters

- **attribute** – A string, the name of the attribute
- **package** – A string, the name of the attribute’s package
- **index** – A string, the name of the index for the specific value of this attribute

dependencies (*recursive=False*)

Return the list of projects on which self depends (either directly if recursive is False, or including indirect dependencies if recursive is True).

Parameters recursive – A boolean

Returns A list of `GPS.Project` instances

file ()

Return the project file

Returns An instance of `GPS.File`

generate_doc (*recursive=False*)

Generate the documentation of the project and its subprojects if recursive is True, and display it with the default browser

Parameters recursive – A boolean

See Also:

`GPS.File.generate_doc()`

get_attribute_as_list (*attribute, package='', index=''*)

Fetch the value of the attribute in the project.

If the package is not specified, the attribute at the toplevel of the project is queried.

The index only needs to be specified if it applies to that attribute.

If the attribute value is stored as a simple string, a list with a single element is returned. This function always returns the value of the attribute in the currently selected scenario.

Parameters

- **attribute** – A string, the name of the attribute
- **package** – A string, the name of the attribute’s package
- **index** – A string, the name of the index for the specific value of this attribute

Returns A list of strings

See Also:

`GPS.Project.scenario_variables()`

`GPS.Project.get_attribute_as_string()`

`GPS.Project.get_tool_switches_as_list()`

```
# If the project file contains the following text:
#
#   project Default is
#     for Exec_Dir use "exec/";
#     package Compiler is
#       for Switches ("file.adb") use ("-c", "-g");
#     end Compiler;
#   end Default;

# Then the following commands;

a = GPS.Project("default").get_attribute_as_list("exec_dir")
=> a = ("exec/")

b = GPS.Project("default").get_attribute_as_list(
  "switches", package="compiler", index="file.adb")
=> b = ("-c", "-g")
```

get_attribute_as_string (*attribute*, *package=''*, *index=''*)

Fetch the value of the attribute in the project.

If the package is not specified, the attribute at the toplevel of the project is queried.

The index only needs to be specified if it applies to that attribute.

If the attribute value is stored as a list, the result string is a concatenation of all the elements of the list. This function always returns the value of the attribute in the currently selected scenario.

When the attribute is not explicitly overridden in the project, the default value is returned. This default value is the one described in an XML file (see the GPS documentation for more information). This default value is not necessarily valid, and could for instance be a string starting with a parenthesis, as explained in the GPS documentation.

Parameters

- **attribute** – A string, the name of the attribute
- **package** – A string, the name of the attribute's package
- **index** – A string, the name of the index for the specific value of this attribute

Returns A string, the value of this attribute

See Also:

`GPS.Project.scenario_variables()`

`GPS.Project.get_attribute_as_list()`

`GPS.Project.get_tool_switches_as_string()`

```
# If the project file contains the following text:
#   project Default is
#     for Exec_Dir use "exec/";
#     package Compiler is
#       for Switches ("file.adb") use ("-c", "-g");
#     end Compiler;
#   end Default;

a = GPS.Project("default").get_attribute_as_string("exec_dir")
=> a = "exec/"

b = GPS.Project("default").get_attribute_as_string(
```

```
"switches", package="compiler", index="file.adb")
=> b = "-c -g"
```

get_executable_name (*main*)

Return the name of the executable, either read from the project or computed from main

Parameters *main* – `GPS.File`

Returns A string

get_property (*name*)

Return the value of the property associated with the project. This property might have been set in a previous GPS session if it is persistent. An exception is raised if no such property already exists for the project

Parameters *name* – A string

Returns A string

See Also:

```
GPS.Project.set_property()
```

get_tool_switches_as_list (*tool*)

Same as `get_attribute_as_list`, but specialized for the switches of a specific tool. Tools are defined through XML customization files, see the GPS documentation for more information

Parameters *tool* – The name of the tool whose switches you want to get

Returns A list of strings

See Also:

```
GPS.Project.get_attribute_as_list()
```

```
GPS.Project.get_tool_switches_as_string()
```

```
# If GPS has loaded a customization file that contains the following
# tags:
#
#   <?xml version="1.0" ?>
#   <toolexample>
#     <tool name="Find">
#       <switches>
#         <check label="Follow links" switch="-follow" />
#       </switches>
#     </tool>
#   </toolexample>
```

```
# The user will as a result be able to edit the switches for Find in
# the standard Project Properties editor.
```

```
# Then the python command
```

```
GPS.Project("default").get_tool_switches_as_list("Find")
```

```
# will return the list of switches that were set by the user in the
# Project Properties editor.
```

get_tool_switches_as_string (*tool*)

Same as `GPS.Project.get_attribute_as_string`, but specialized for a specific tool.

Parameters *tool* – The name of the tool whose switches you want to get

Returns A string

See Also:`GPS.Project.get_tool_switches_as_list()`**is_modified** (*recursive=False*)

Return True if the project has been modified but not saved yet. If recursive is true, then the return value takes into account all projects imported by self

Parameters **recursive** – A boolean

Returns A boolean

languages (*recursive=False*)

Return the list of languages that are used for the sources of the project (and its subprojects if recursive is True). This can be used to detect whether some specific action in a module should be activated or not. Language names are always lowercase

Parameters **recursive** – A boolean

Returns A list of strings

```
# The following example adds a new menu only if the current project
# supports C. This is refreshed every time the project is changed by
# the user.
```

```
import GPS
c_menu=None

def project_recomputed(hook_name):
    global c_menu
    try:
        ## Check whether python is supported
        GPS.Project.root().languages(recursive=True).index("c")
        if c_menu == None:
            c_menu = GPS.Menu.create("/C support")
    except:
        if c_menu:
            c_menu.destroy()
            c_menu = None

GPS.Hook("project_view_changed").add(project_recomputed)
```

static load (*filename, force=False, keep_desktop=False*)

Load a new project, which replaces the current root project, and return a handle to it. All imported projects are also loaded at the same time. If the project is not found, a default project is loaded.

If *force* is True, then the user will not be asked whether to save the current project, whether it was modified or not.

If *keep_desktop* is False, then load saved desktop configuration, keep current otherwise

Parameters

- **filename** – A string, the full path to a project file
- **force** – A boolean
- **keep_desktop** – A boolean

Returns An instance of `GPS.Project`

name ()

Return the name of the project. This doesn't include directory information, see `self.file().name()` if you wish to access that information

Returns A string, the name of the project

object_dirs (*recursive=False*)

Return the list of object directories for this project. If Recursive is True, the source directories of imported projects is also returned. There might be duplicate directories in the returned list

Parameters recursive – A boolean

Returns A list of strings

properties_editor ()

Launch a graphical properties editor for the project

static recompute ()

Recompute the contents of a project, including the list of source files that are automatically loaded from the source directories. The project file is not reloaded from the disk, and this should only be used if you have created new source files outside of GPS for instance

```
GPS.Project.recompute()
```

remove_attribute_values (*attribute, package, index, value*)

Removes some specific values from an attribute. You can set as much as many values you need at the end of the param list.

If the package is not specified, the attribute at the toplevel of the project is queried.

The index only needs to be specified if it applies to that attribute.

Parameters

- **attribute** – A string, the name of the attribute
- **package** – A string, the name of the attribute's package
- **index** – A string, the name of the index for the specific value of this attribute
- **value** – A string, the name of the first value to remove

See Also:

```
GPS.Project.set_attribute_as_string()
```

```
GPS.Project.add_attribute_values()
```

```
GPS.Project.clear_attribute_values()
```

```
GPS.Project.root().remove_attribute_values(
    "Default_Switches", "Compiler", "ada", "-gnatwa", "-gnatwe");
```

remove_dependency (*imported*)

Remove a dependency between two projects. You must call GPS.Project.recompute() once you are done doing all the modifications on the projects

Parameters imported – An instance of GPS.Project

See Also:

```
GPS.Project.add_dependency()
```

remove_property (*name*)

Removes a property associated with a project

Parameters name – A string

See Also:

```
GPS.Project.set_property()
```

remove_source_dir (*directory*)

Remove a source directory from the project. You should call `recompute()` after calling this method, to recompute the list of source files. The directory is added for the current value of the scenario variables only

Parameters *directory* – A string

See Also:

```
GPS.Project.add_source_dir()
```

rename (*name*, *path*='*<current path>*')

Rename and move a project file (the project will only be put in the new directory when it is saved, but will not be removed from its original directory). You must call `GPS.Project.recompute()` sometime after changing the name.

Parameters

- **name** – A string
- **path** – A string

static root ()

Return the root project currently loaded in GPS

Returns An instance of `GPS.Project`

```
print "Current project is " + GPS.Project.root().name()
```

static scenario_variables ()

Return the list of scenario variables for the current project hierarchy, and their current value. These variables are visible at the top of the Project View in the GPS window. The initial value for these variables is set from the environment variables' value when GPS is started. However, changing the value of the environment variable later on doesn't change the value of the scenario variable.

Returns hash table associating variable names and values

See Also:

```
GPS.Project.set_scenario_variable()
```

```
GPS.Project.scenario_variables()["foo"]  
=> returns the current value for the variable foo
```

static scenario_variables_cmd_line (*prefix*='')

Return a concatenation of `VARIABLE=VALUE`, each preceded by the given prefix. This string will generally be used when calling external tools, for instance `make` or `GNAT`

Parameters *prefix* – String to print before each variable in the output

Returns a string

```
# The following GPS action can be defined in an XML file, and will launch  
# the make command with the appropriate setup for the environment  
# variables:  
# <action name="launch make"> # <shell lang="python">GPS.scenario_variables
```

static scenario_variables_values ()

Return a hash table where keys are the various scenario variables defined in the current project and values the different values that this variable can get.

Returns A hash table of strings

search (*pattern*, *case_sensitive=False*, *regex=False*, *scope='whole'*, *recursive=True*)

Return the list of matches for *pattern* in all the files belonging to the project (and its imported projects if *recursive* is true (default)). *Scope* is a string, and should be any of 'whole', 'comments', 'strings', 'code'. The latter will match only for text outside of comments

Parameters

- **pattern** – A string
- **case_sensitive** – A boolean
- **regex** – A boolean
- **scope** – One of (“whole”, “comments”, “strings”, “code”)
- **recursive** – A boolean

Returns A list of GPS.FileLocation instances

set_attribute_as_string (*attribute*, *package*, *index*, *value*)

Sets the value of an attribute. The attribute has to be stored as a single value. If the package is not specified, the attribute at the toplevel of the project is queried. The index only needs to be specified if it applies to that attribute.

Parameters

- **attribute** – A string, the name of the attribute
- **package** – A string, the name of the attribute’s package
- **index** – A string, the name of the index for the specific value of this attribute
- **value** – A string, the name of the value to set

See Also:

`GPS.Project.add_attribute_values()`

`GPS.Project.remove_attribute_values()`

`GPS.Project.clear_attribute_values()`

set_property (*name*, *value*, *persistent=False*)

Associates a string property with the project. This property is retrievable during the whole GPS session, or across GPS sessions if *persistent* is set to True.

This is different than setting instance properties through Python’s standard mechanism in that there is no guarantee that the same instance of GPS.Project will be created for each physical project on the disk, and therefore you would not be able to associate a property with the physical project itself

Parameters

- **name** – A string
- **value** – A string
- **persistent** – A boolean

See Also:

`GPS.Project.get_property()`

`GPS.Project.remove_property()`

`GPS.File.set_property()`

static set_scenario_variable (*name, value*)

Change the value of a scenario variable. You need to call `GPS.Project.recompute()` to activate this change (so that multiple changes to the project can be grouped)

Parameters

- **name** – A string
- **value** – A string

See Also:

`GPS.Project.scenario_variables()`

source_dirs (*recursive=False*)

Return the list of source directories for this project. If `Recursive` is `True`, the source directories of imported projects is also returned. There might be duplicate directories in the returned list

Parameters recursive – A boolean

Returns A list of strings

See Also:

`GPS.Project.add_source_dir()`

sources (*recursive=False*)

Return the list of source files for this project. If `recursive` is `true`, then all sources from imported projects are also returned. Otherwise, only the direct sources are returned. The basenames of the returned files are always unique: not two files with the same basenames are returned, and the one returned is the first one seen while traversing the project hierarchy

Parameters recursive – A boolean

Returns A list of instances of `GPS.File`

update_xref (*recursive=False*)

Updates the cross-reference information in memory for all files of the project. This doesn't regenerate that information, just read all the `.ali` files found in the object directory of the project (and all imported projects if `recursive` is `True`). This should generally be called before calling `GPS.freeze_xref`, for efficiency.

Parameters recursive – A boolean

15.5.50 `GPS.ProjectTemplate`

class `GPS.ProjectTemplate`

This class is used to manipulate GPS Project Templates.

static add_templates_dir (*noname*)

Add a directory to the path in which GPS looks for templates. GPS will look for project templates in immediate subdirectories of this directory.

Parameters noname – A `GPS.File` pointing to a directory.

15.5.51 `GPS.ReferencesCommand`

class `GPS.ReferencesCommand`

This is the type of the commands returned by the references extractor.

See Also:

`GPS.Command()`

`GPS.Entity.references()`

get_result()

Returns the references that have been found so far by the command.

Returns A list of strings

See Also:

`GPS.Entity.references()`

15.5.52 `GPS.Revision`

class `GPS.Revision`

General interface to the revision browser

static add_link (*file, revision_1, revision_2*)

Create a link between `revision_1` and `revision_2` for the given file

Parameters

- **file** – A string
- **revision_1** – A string
- **revision_2** – A string

static add_log (*file, revision, author, date, log*)

Add a new log entry into the revision browser

Parameters

- **file** – A string
- **revision** – A string
- **author** – A string
- **date** – A string
- **log** – A string

static add_revision (*file, revision, symbolic_name*)

Register a new symbolic name (tag or branches) corresponding to the specified revision of file

Parameters

- **file** – A string
- **revision** – A string
- **symbolic_name** – A string

static clear_view (*file*)

Clear file's revision view

Parameters **file** – A string

15.5.53 `GPS.Search`

class `GPS.Search`

This class provides an interface to the search facilities used for the GPS omni-search. In particular, this allows you to search file names, sources, actions,...

This class provides facilities exported directly by Ada, so that you can for instance look for file names by writing:

```
s = GPS.Search.lookup(GPS.Search.FILE_NAMES)
s.set_pattern("search", flags=GPS.Search.FUZZY)
while True:
    (has_next, result) = s.get()
    if result:
        print result.short
    if not has_next:
        break
```

However, one of the mandatory GPS plugins augments this base class with high-level constructs such as iterators, and now you can write code as:

```
for result in GPS.Search.search(
    GPS.Search.FILE_NAMES, "search", GPS.Search.FUZZY):
    print result.short
```

Iterations are meant to be done in the background, so they are split into small units.

It is possible to create your own search providers (which would be fully included in the omni-search of GPS) by subclassing this class, as in:

```
class MySearchResult(GPS.Search_Result):
    def __init__(self, str):
        self.short = str
        self.long = "Long description: %s" % str

    def show(self):
        print "Showing a search result: '%s'" % self.short

class MySearchProvider(GPS.Search):
    def __init__(self):
        # Override default so that we can build instances of our class
        pass

    def set_pattern(self, pattern, flags):
        self.pattern = pattern
        self.flags = flags
        self.current = 0

    def get(self):
        if self.current == 3:
            return (False, None)    # no more matches
        self.current += 1
        return (True, # might have more matches
              MySearchResult("<b>match</b> %d for '%s' (flags=%d)"
                             % (self.current, self.pattern, self.flags)))

GPS.Search.register("MySearch", MySearchProvider())
```

ACTIONS = 'Actions'

BOOKMARKS = 'Bookmarks'

The various contexts in which a search can occur.

BUILDS = 'Build'

CASE_SENSITIVE = 8

ENTITIES = 'Entities'

FILE_NAMES = 'File names'

FUZZY = 1

OPENED = 'Opened'

REGEXP = 4

The various types of search, similar to what GPS provides in its omni-search.

SOURCES = 'Sources'

SUBSTRINGS = 2

WHOLE_WORD = 16

Flags to configure the search, that can be combined with the above.

__init__ ()

Always raises an exception, use `GPS.Search.lookup` to retrieve an instance.

get ()

Returns the next occurrence of the pattern.

Returns a tuple containing two elements; the first element is a boolean that indicates whether there might be further results; the second element is either `None` or an instance of `GPS.Search.Result`. It might be set even if the first element is `False`. On the other hand, it might be `None` even if there might be further results, since the search itself is split into small units. For instance, when searching in sources, each source file will be parsed independently. If a file does not contain a match, `next()` will return a tuple that contains `True` (there might be matches in other files) and `None` (there were no match found in the current file).

static lookup (*name*)

Lookup one of the existing search factories.

Parameters *name* – a string, one of `GPS.Search.FILE_NAMES`, `GPS.Search.SOURCES`,...

next ()

Returns the next non-null result. This might take longer than `get()`, since it will keep looking until it actually finds a new result. It raises `StopIteration` when there are no more results.

static register (*name, factory*)

Register a new custom search. This will be available to users via the omni-search in GPS, or of course via the `GPS.Search` class.

Parameters

- **name** – a string
- **factory** – an instance of `GPS.Search` that will be reused every time the user starts a new search.

static search (*context, pattern, flags=2*)

A high-level wrapper around `lookup` and `set_pattern` to make python code more readable (see general documentation for `GPS.Search`).

Parameters

- **context** – a string, for instance `GPS.Search.SOURCES`
- **pattern** – a string
- **flags** – an integer, see `GPS.Search.set_pattern`

set_pattern (*pattern*, *flags=0*)

Set the search pattern.

Parameters

- **pattern** – a string
- **flags** – an integer, the combination of values such as `GPS.Search.FUZZY`, `GPS.Search.REGEXP`, `GPS.Search.SUBSTRINGS`, `GPS.Search.CASE_SENSITIVE`, `GPS.Search.WHOLE_WORD`.

15.5.54 `GPS.Search.Result`

class `GPS.Search.Result`

A class that represents the results found by `GPS.Search`.

long = ''

A long version of the description. For instance, when looking in sources it might contain the full line that matches

short = ''

The short description of the result

__init__ ()

`x.__init__(...)` initializes `x`; see `help(type(x))` for signature

show ()

Execute the action associated with the result. This action depends on where you were searching. For instance, search in file names would as a result open the corresponding file; searching in bookmarks would jump to the corresponding location; search in actions would execute the corresponding action.

15.5.55 `GPS.Style`

class `GPS.Style` (*name*, *create*)

This class is used to manipulate GPS Styles, which are used for instance to represent graphical attributes given to Messages.

This class is fairly low-level, and we recommend using the class `gps_utils.highlighter.OverlayStyle()` instead. That class provides similar support for specifying attributes, but makes it easier to highlight sections of an editor with that style, or to remove the highlighting.

__init__ (*name*, *create*)

Create a Style

Parameters

- **name** – A String indicating the name of the Style
- **create** – A File indicating the file

```
# Create a new style
s=GPS.Style("my new style")

# Set the background color to yellow
s.set_background("#ffff00")

# Apply the style to all the messages
[m.set_style(s) for m in GPS.Message.list()]
```

get_background()

Returns a string, background of the style

get_foreground()

Returns a string, foreground of the style

get_in_speedbar()

Return a Boolean indicating whether this style is shown in the speedbar.

Returns a boolean

get_name()

Returns a string, the name of the style.

static list()

Return a list of all styles currently registered in GPS.

Returns a list of `GPS.Style`

set_background(*noname*)

Set the background of style to the given color.

Parameters *noname* – A string representing a color, for instance “blue” or “#0000ff”

set_foreground(*noname*)

Set the foreground of style to the given color.

Parameters *noname* – A string representing a color, for instance “blue” or “#0000ff”

set_in_speedbar(*noname*)

Whether this style should appear in the speedbar.

Parameters *noname* – A Boolean

15.5.56 `GPS.SwitchesChooser`

class `GPS.SwitchesChooser` (*name, xml*)

This class represents a gtk widget that can be used to edit a tool’s command line.

__init__ (*name, xml*)

Creates a new `SwitchesChooser` widget from the tool’s name and switch description in xml format.

Parameters

- **name** – A string
- **xml** – A string

get_cmd_line()

Return the tool’s command line parameter

Returns A string

set_cmd_line (*cmd_line*)

Modify the widget’s aspect to reflect the command line.

Parameters *cmd_line* – A string

15.5.57 `GPS.Task`

class `GPS.Task`

This class provides an interface to the background tasks being handled by GPS, such as the build commands, the query of cross references, etc. These are the same tasks that are visible through the GPS Task Manager.

Note that the classes represented with this class cannot be stored.

visible = False

Whether the task has a visible progress bar in GPS's toolbar or the task manager.

block_exit ()

Return True iff this task should block the exit of GPS

Returns A boolean

interrupt ()

Interrupt the task

static list ()

Returns a list of `GPS.Task`, all running tasks

name ()

Return the name of the task

Returns A string

pause ()

Pause the task

progress ()

Return the current progress of the task

Returns A list containing the current step and the total steps

resume ()

Resume the paused task

status ()

Return the status of the task

Returns A string

15.5.58 `GPS.Timeout`

class `GPS.Timeout` (*timeout, action*)

This class gives access to actions that must be executed regularly at specific intervals

See Also:

```
GPS.Timeout.__init__()  
  
## Execute callback three times and remove it  
import GPS;  
  
def callback(timeout):  
    timeout.occure += 1  
    print "A timeout occur=" + `timeout.occure`  
    if timeout.occure == 3:  
        timeout.remove()
```

```
t = GPS.Timeout(500, callback)
t.occure = 0
```

__init__ (*timeout, action*)

A timeout object executes a specific action repeatedly, at a specified interval, as long as it is registered. The action takes a single argument, which is the instance of GPS.Timeout that called it.

Parameters

- **timeout** – The timeout in milliseconds at which to execute the action
- **action** – A subprogram parameter to execute periodically

remove ()

Unregister a timeout

15.5.59 GPS.ToolButton

class GPS.ToolButton (*stock_id, label, on_click*)

This class represents a button that can be inserted in the toolbar

See Also:

GPS.ToolButton.__init__()

__init__ (*stock_id, label, on_click*)

Initializes a new button. When the button is pressed by the user, on_click is called with the following single parameter:

- \$1 = The instance of GPS.Button

Parameters

- **stock_id** – A string identifying the icon
- **label** – A string, the text that appears on the button
- **on_click** – A subprogram, see the GPS documentation

```
b = GPS.ToolButton("gtk-new", "New File",
                  lambda x : GPS.execute_action("/File/New"))
GPS.Toolbar().insert(b, 0)
```

15.5.60 GPS.Toolbar

class GPS.Toolbar

Interface to commands related to the toolbar. This allows you to add new combo boxes to the GPS toolbars. Note that this can also be done through XML files, see the GPS documentation

See Also:

GPS.Toolbar.__init__()

import GPS

```
def on_changed(entry, choice):
    print "changed " + choice + ' ' + entry.custom
```

```
def on_selected(entry, choice):
```

```
print "on_selected " + choice + ' ' + entry.custom

ent = GPS.Combo("foo", label="Foo", on_changed=on_changed)

GPS.Toolbar().append(ent, tooltip => "What it does")

ent.custom = "Foo" ## Create any field you want
ent.add(choice="Choice1", on_selected=on_selected)
ent.add(choice="Choice2", on_selected=on_selected)
ent.add(choice="Choice3", on_selected=on_selected)
```

It is easier to use this interface through XML customization files, see the GPS documentation. However, this can also be done through standard GPS shell commands:

```
Combo "foo" "Foo" "on_changed_action"
Toolbar
Toolbar.append %1 %2 "What it does"

Toolbar
Toolbar.get %1 "foo"
Combo.add %1 "Choice1" "on_selected"action"
```

`__init__()`

Initializes a new instance of the toolbar, associated with the default toolbar of GPS. This is called implicitly from python

`append(widget, tooltip='')`

Add a new widget in the toolbar. This can be an instance of `GPS.Combo`, or a `GPS.Button`, or a `GPS.ToolButton`.

Parameters

- **widget** – An instance of `GPS.GUI`
- **tooltip** – A string

`get(id)`

Return the toolbar entry matching the given id. An error is raised if no such entry exists. The same instance of `GPS.Combo` is always returned for each specific id, therefore you can store your own fields in this instance and access it later.

Parameters `id` – A string, the name of the entry to get

Returns An instance of `GPS.Combo`

```
ent = GPS.Combo("foo")
GPS.Toolbar().append(ent)
ent.my_custom_field = "Whatever"

print GPS.Toolbar().get("foo").my_custom_field
=> "Whatever"
```

`get_by_pos(position)`

Return the position-th widget in the toolbar. If the widget was created from a scripting language, its instance is returned. Otherwise, a generic instance of `GPS.GUI` is returned. This can be used to remove some items from the toolbar for instance

Parameters `position` – An integer, starting at 0

Returns An instance of a child of `GPS.GUI`

```
GPS.Toolbar().get_by_pos(0).set_sensitive(False)
# can be used to gray out the first item in the toolbar
```

insert (*widget*, *pos=-1*, *tooltip=""*)

Add a new widget in the toolbar. This can be an instance of GPS.Combo, or a GPS.Button, or a GPS.ToolButton.

Parameters

- **widget** – An instance of `GPS.GUI`
- **pos** – The position at which to insert the widget
- **tooltip** – A string

15.5.61 GPS.Unexpected_Exception

class `GPS.Unexpected_Exception`

An exception raised by GPS. It indicates an internal error in GPS, raised by the Ada code itself. This exception is unexpected and indicates a bug in GPS itself, not in the python script, although it might be possible to modify the latter to work around the issue

15.5.62 GPS.VCS

class `GPS.VCS`

General interface to version control systems

static annotate (*file*)

Display the annotations for file

Parameters **file** – A string

static annotations_parse (*vcs_identifier*, *file*, *output*)

Parses the output of the annotations command (cvs annotate for instance), and add the corresponding information to the left of the editor

Parameters

- **vcs_identifier** – A string
- **file** – A string
- **output** – A string

static commit (*file*)

Commit file

Parameters **file** – A string

static diff_head (*file*)

Show differences between local file and the head revision

Parameters **file** – A string

static diff_working (*file*)

Show differences between local file and the working revision

Parameters **file** – A string

static get_current_vcs ()

Return the system supported for the current project

Returns A string

static get_log_file (*file*)

Returns the GPS File corresponding to the log file for given file.

Parameters **file** – A string

static get_status (*file*)

Query the status for file

Parameters **file** – A string

static log (*file, revision*)

Get the revision changelog for file. If revision is specified, query the changelog for this specific revision, otherwise query the entire changelog

Parameters

- **file** – A string
- **revision** – A string

static log_parse (*vcs_identifier, file, string*)

Parses string to find log entries for file. This command uses the parser in the XML description node for the VCS corresponding to vcs_identifier.

Parameters

- **vcs_identifier** – A string
- **file** – A string
- **string** – A string

static remove_annotations (*file*)

Remove the annotations for file

Parameters **file** – A string

static repository_dir (*tag_name=''*)

Returns the repository root directory, or if tag_name is specified the repository directory for the given tag or branch.

Parameters **tag_name** – A string

static repository_path (*file, tag_name=''*)

Returns the trunk repository path for file or if tag_name is specified the repository path on the given tag or branch path.

Parameters

- **file** – A string
- **tag_name** – A string

static revision_parse (*vcs_identifier, file, string*)

Parses string to find revisions tags and branches information for file. This command uses the parser in the XML description node for the VCS corresponding to vcs_identifier.

Parameters

- **vcs_identifier** – A string
- **file** – A string
- **string** – A string

static set_reference (*file, reference*)

Record a reference file (the file on which a diff buffer is based for example) for a given file

Parameters

- **file** – A string
- **reference** – A string

static status_parse (*vcs_identifier, string, clear_logs, local, dir=''*)

Parses a string for VCS status. This command uses the parsers defined in the XML description node for the VCS corresponding to *vcs_identifier*.

- When *local* is FALSE, the parser defined by the node *status_parser* is used.
- When *local* is TRUE, the parser defined by the node *local_status_parser* is used.

If *clear_logs* is TRUE, the revision logs editors are closed for files that have the VCS status “up-to-date”. Parameter *dir* indicates the directory in which the files matched in *string* are located.

Parameters

- **vcs_identifier** – A string
- **string** – A string
- **clear_logs** – A boolean
- **local** – A boolean
- **dir** – A string

static supported_systems ()

Show the list of supported VCS systems

Returns List of strings

static update (*file*)

Update file

Parameters **file** – A string

static update_parse (*vcs_identifier, string, dir=''*)

Parses a string for VCS status. This command uses the parsers defined in the XML description node for the VCS corresponding to *vcs_identifier*.

Parameter *dir* indicates the directory in which the files matched in *string* are located.

Parameters

- **vcs_identifier** – A string
- **string** – A string
- **dir** – A string

15.5.63 GPS.Vdiff

class GPS.Vdiff

This class provides access to the graphical comparison between two or three files or two versions of the same file within GPS. A visual diff is a group of two or three editors with synchronized scrolling. Differences are rendered using blank lines and color highlighting.

static `__init__` ()

This function prevents the creation of a visual diff instance directly. You must use `GPS.Vdiff.create()` or `GPS.Vdiff.get()` instead.

See Also:

`GPS.Vdiff.create()`

`GPS.Vdiff.get()`

close_editors ()

Close all editors implied in a visual diff.

static create (file1, file2, file3='')

If none of the files given as parameter is already used in a visual diff, this function creates a new visual diff and returns it. Otherwise, `None` is returned.

Parameters

- **file1** – An instance of `GPS.File`
- **file2** – An instance of `GPS.File`
- **file3** – An instance of `GPS.File`

Returns An instance of `GPS.Vdiff`

files ()

Return the list of files used in a visual diff.

Returns A list of `GPS.File`

static get (file1, file2='', file3='')

Return an instance of an already existing visual diff. If an instance already exists for this visual diff, it is returned. All files passed as parameters have to be part of the visual diff but not all files of the visual diff have to be passed for the visual diff to be returned. For example if only one file is passed the visual diff that contains it, if any, will be returned no matter it is a two or three files visual diff.

Parameters

- **file1** – An instance of `GPS.File`
- **file2** – An instance of `GPS.File`
- **file3** – An instance of `GPS.File`

static list ()

This function returns the list of visual diff currently opened in GPS.

Returns A list `GPS.Vdiff`

```
# Here is an example that demonstrates how to use GPS.Vdiff.list to  
# close all the visual diff.
```

```
# First two visual diff are created  
vdiff1 = GPS.Vdiff.create(GPS.File("a.adb"), GPS.File("b.adb"))  
vdiff2 = GPS.Vdiff.create(GPS.File("a.adb"), GPS.File("b.adb"))
```

```
# Then we get the list of all current visual diff  
vdiff_list = GPS.Vdiff.list()
```

```
# And we iterate on that list in order to close all editors used in  
# each visual diff from the list.
```

```
for vdiff in vdiff_list:
```

```

files = vdiff.files()

# But before each visual diff is actually closed, we just inform
# the user of the files that will be closed.

for file in files:
    print "Beware! " + file.name () + "will be closed."

# Finally, we close the visual diff

vdiff.close_editors()

```

recompute()

Recompute a visual diff. The content of each editor used in the visual diff is saved. The files are recompared and the display is redone (blank lines and color highlighting).

15.5.64 GPS.XMLViewer

class GPS.**XMLViewer**(*name, columns=3, parser=None, on_click=None, on_select=None, sorted=False*)

This class represents Tree-based views for XML files

__init__(*name, columns=3, parser=None, on_click=None, on_select=None, sorted=False*)

Create a new XMLViewer, with the given name.

columns is the number of columns that the table representation should have. The first column is always the one used for sorting the table.

parser is a subprogram called for each XML node that is parsed. It takes three arguments: the name of the XML node being visited, its attributes (in the form "attr='foo' attr='bar'"), and the text value of that node. This subprogram should return a list of strings, one per visible column create for the table. Each element will be put in the corresponding column.

If *parser* is unspecified, the default is to display in the first column the tag name, in the second column the list of attributes, and in the third column when it exists the textual contents of the node.

on_click is an optional subprogram. It is called every time the user double-click on a line, and is passed the same arguments as Parser. It has no return value.

on_select has the same profile as *on_click*, but is called when the user has selected a new line, not double-clicked on it.

If *sorted* is True, then the resulting graphical list is sorted on the first column.

Parameters

- **name** – A string
- **columns** – An integer
- **parser** – A subprogram
- **on_click** – A subprogram
- **on_select** – A subprogram
- **sorted** – A boolean

```

# Display a very simply tree. If you click on the file name,
# the file will be edited.
import re

```

```
xml = '''<project name='foo'>
    <file>source.adb</file>
</project>'''

view = GPS.XMLViewer("Dummy", 1, parser, on_click)
view.parse_string(xml)

def parser(node_name, attrs, value):
    attr = dict()
    for a in re.findall(''' (\w+)=['"](.*)['"]\B''', attrs):
        attr[a[0]] = a[1]

    if node_name == "project":
        return [attr["name"]]

    elif node_name == "file":
        return [value]

def on_click(node_name, attrs, value):
    if node_name == "file":
        GPS.EditorBuffer.get(GPS.File(value))
```

create_metric (*name*)

Create a new XMLViewer for an XML file generated by gnatmetric. Name is used as the name for the window

Parameters **name** – A string

parse (*filename*)

Replace the contents of self by that of the XML file

Parameters **filename** – An XML file

parse_string (*str*)

Replace the contents of self by that of the XML string

Parameters **str** – A string

USEFUL PLUG-INS

16.1 User plug-ins

GPS comes with a number of plug-ins. Some of these plugins are activated by default, others are not. In both cases, you can control which plug-ins should be activated by using the menu *Tools/Plug-ins*.

This section highlights a few of these plug-ins, but GPS comes with many more plug-ins. The dialog *Tools/Plug-ins* will show their description, so that you can decide whether you want them or not.

16.1.1 The `auto_highlight_occurrences.py` module

This plug-in highlights all occurrences of the entity under the cursor.

Whenever your cursor rests in a new location, GPS will search for all other places where this entity is referenced using either the cross-reference engine in GPS, if this information is up-to-date, or a simple textual search. Each of these occurrences will then be highlighted in a color depending on the kind of the entity.

This plug-in does its work in the background, whenever GPS is not busy responding to your actions, so it should have limited impact on the performances and responsiveness of GPS.

If you are interested in doing something similar in your own plugins, we recommend you look at the `gps_utils.highlighter.Background_Highlighter` class instead, which provides the underlying framework.

A similar plug-in which you might find useful is in the `gps_utils.highlighter.Regexp_Highlighter` class. By creating a simple python file in your `gps` directory, you are able to highlight any regular expression in the editor, which is useful for highlighting text like “TODO”, or special comments for instance.

16.1.2 The `dispatching.py` module

Highlighting all dispatching calls in the current editor

This package will highlight with a special background color all dispatching calls found in the current editor. In particular, at such locations, the cross-references might not lead accurate result (for instance “go to body”), since the exact subprogram that is called is not known until run time.

16.2 Helper plug-ins

A number of plug-ins are in fact useful when you want to create your own plug-ins.

16.2.1 The `gps_utils` module

`gps_utils.execute_for_all_cursors` (*editor*, *mark_fn*, *extend_selection=False*)

Execute the function *mark_fn* for every cursor in the editor, meaning, the main cursor + every existing multi cursor. *mark_fn* has the prototype `def mark_fn(EditorBuffer, EditorMark)`

`gps_utils.freeze_prefs` ()

A context manager that temporarily freezes GPS' `preferences_changed` signal from being emitted, and then reactivated it. This is useful when modifying a large number of preferences as a single batch.

This can be used as:

```
with gps_utils.freeze_prefs():
    GPS.Preference(...).set(...)
    GPS.Preference(...).set(...)
    GPS.Preference(...).set(...)
```

`gps_utils.in_ada_file` (*context*)

Returns True if the focus is currently inside an Ada editor

`gps_utils.in_xml_file` (*context*)

Returns True if the focus is in an XML editor

class `gps_utils.interactive` (*category='General'*, *filter=''*, *menu=''*, *key=''*, *contextual=''*, *name=''*, *before=''*, *after=''*)

A decorator with the same behavior as `make_interactive()`. This can be used to easily associate a function with an interactive action, menu or key, so that a user can conveniently call it:

```
@interactive("Editor", menu="/Edit/Foo")
def my_function():
    pass
```

`gps_utils.is_writable` (*context*)

Returns True if the focus is currently inside a writable editor

`gps_utils.make_interactive` (*callback*, *category='General'*, *filter=''*, *menu=''*, *key=''*, *contextual=''*, *name=''*, *before=''*, *after=''*)

Declare a new GPS action (an interactive function, in Emacs talk), associated with an optional menu and default key. The description of the action is automatically taken from the documentation of the python function. Likewise the name of the action is taken from the name of the python function, unless specified with *name*.

Parameters

- **callback** – is a python function that requires no argument, although it can have optional arguments (none will be set when this is called from the menu or the key shortcut). Alternatively, *callback* can also be a class: when the user executes the action, a new instance of the class is created, so it is expected that the work is done in the `__init__` of the class. This is in particular useful for classes that derive from `CommandWindow`.
- **menu** – The name of a menu to associate with the action. It will be placed within its parent just before the item referenced as *before*, or after the item referenced as *after*.

`gps_utils.remove_interactive` (*menu=''*, *name=''*, *contextual=''*)

Undo the effects of `make_interactive`.

`gps_utils.save_current_window` (*f*, *args=[]*, *kwargs={}*)

Save the window that currently has the focus, executes *f*, and reset the focus to that window.

`gps_utils.save_dir` (*fn*)

Saves the current directory before executing the instrumented function, and restore it on exit. This is a python decorator which should be used as:

```
@save_dir
def my_function():
    pass
```

`gps_utils.save_excursion(f, args=[], kwargs={}, undo_group=True)`

Save current buffer, cursor position and selection and execute `f`. (`args` and `kwargs`) are passed as arguments to `f`. They indicate that any number of parameters (named or unnamed) can be passed in the usual way to `save_excursion`, and they will be transparently passed on to `f`. If `undo_group` is `True`, then all actions performed by `f` will be grouped so that the user needs perform only one single undo to restore previous start.

Then restore the context as it was before, even in the case of abnormal exit.

Example of use:

```
def my_subprogram():
    def do_work():
        pass # do actual work here
    save_excursion(do_work)
```

See also the `with_save_excursion` decorator below for cases when you need to apply `save_excursion` to a whole function.

`gps_utils.with_save_current_window(fn)`

A decorator with the same behavior as `save_current_window`.

`gps_utils.with_save_excursion(fn)`

A decorator with the same behavior as `save_excursion`. To use it, simply add `@with_save_excursion` before the definition of the function. This ensures that the current context will be restored when the function terminates:

```
@with_save_excursion
def my_function():
    pass
```

16.2.2 The `gps_utils.highlighter.py` module

This file provides various classes to help highlight patterns in files.

class `gps_utils.highlighter.Background_Highlighter` (*style*)

An abstract class that provides facilities for highlighting parts of an editor. If possible, this highlighting is done in the background so that it doesn't interfere with the user typing. Example of use:

```
class Example(Background_Highlighter):
    def process(self, start, end):
        ... analyze the given range of lines, and perform highlighting
        ... where necessary.

e = Example()
e.start_highlight(buffer1) # start highlighting a first buffer
e.start_highlight(buffer2) # start highlighting a second buffer
```

Parameters `style` (*OverlayStyle*) – style to use for highlighting.

on_start_buffer (*buffer*)

Called before we start processing a new buffer.

process (*start, end*)

Called to highlight the given range of editor. When this is called, previous highlighting has already been removed in that range.

Parameters

- **start** (*GPS.EditorLocation*) – start of region to process.
- **end** (*GPS.EditorLocation*) – end of region to process.

remove_highlight (*buffer=None*)

Remove all highlighting done by self in the buffer.

Parameters **buffer** (*GPS.EditorBuffer*) – defaults to the current buffer**set_style** (*style*)

Change the current highlight style.

Parameters **style** (*OverlayStyle*) – style to use for highlighting.**start_highlight** (*buffer=None, line=None, context=None*)

Start highlighting the buffer, possibly in the background.

Parameters

- **buffer** (*GPS.EditorBuffer*) – The buffer to highlight (defaults to the current buffer). This buffer is added to the list of buffers, and will be processed when other buffers are finished.
- **line** (*integer*) – The line the highlighting should start from. By default, this is the current line in the editor, so that the user sees changes immediately. But you could chose to start from the top of the file instead.
- **context** (*integer*) – Number of lines before and after ‘line’ that should be highlighted. By default, the whole buffer is highlighted.

stop_highlight (*buffer=None*)

Stop the background highlighting of the buffer, but preserves any highlighting that has been done so far.

Parameters **buffer** (*GPS.EditorBuffer*) – If specified, highlighting is only stopped for a specific buffer.**class** `gps_utils.highlighter.Location_Highlighter` (*style, context=2*)

An abstract class that can be used to implement highlighter related to the cross-reference engine. As usual, such an highlighter does its job in the background. To find the places to highlight in the editor, this class relies on having a list of entities and their references. This list will in general be computed once when we start processing a new buffer:

```
class H(Location_Highlighter):
    def recompute_refs(self, buffer):
        return ...computation of references within file ...
```

Parameters **context** (*integer*) – The number of lines both before and after a given reference where we should find for possible approximate matches. This is used when the reference returned by the xref engine was outdated.**recompute_refs** (*buffer*)

Called before we start processing a new buffer.

Returns a list of tuples, each of which contains an (name, *GPS.FileLocation*). The highlighting is only done if the text at the location is name. Name should be a byte-sequence that encodes a UTF-8 strings, not the unicode string itself (the result of *GPS.EditorBuffer.get_chars* or *GPS.Entity.name* can be used).**class** `gps_utils.highlighter.On_The_Fly_Highlighter` (*style, context_lines=0*)

This abstract class provides a way to easily highlight text in an editor. When possible, the highlighting is done

in the background, in which case it is also done on the fly every time the file is modified. If pygobject is not available, the highlighting is only done when the file is opened or saved

Parameters

- **style** (*OverlayStyle*) – the style to apply.
- **context_lines** (*integer*) – The number of lines (plus or minus) around the current location that get refreshed when a local highlighting is requested.

do_highlight (*buffer, start, end*)

Do the highlighting in the range of text. This needs to be overridden to do anything useful

must_highlight (*buffer*)

Parameters **buffer** (*GPS.EditorBuffer*) – The buffer to test.

Returns whether to highlight this buffer. The default is to highlight all buffers, but some highlightings might apply only to specific languages for instance

Return type boolean

start ()

Start highlighting. This is automatically called from `__init__`, and only needs to be called when you have called `stop()` first. Do not call this function multiple times.

stop ()

Stop highlighting through self

class `gps_utils.highlighter.OverlayStyle` (*name, foreground='', background='', weight=None, slant=None, editable=True, whole_line=False, speedbar=False, **kwargs*)

Description for a style to apply to a section of an editor. In practice, this could be implemented as an editor overlay, or a message, depending on whether highlighting should be done on the whole line or not.

Parameters

- **name** (*string*) – name of the overlay so that we can remove it later.
- **foreground** (*string*) – foreground color
- **background** (*string*) – background color
- **weight** (*string*) – one of “bold”, “normal”, “light”
- **slant** (*string*) – one of “normal”, “oblique”, “italic”
- **whole_line** (*boolean*) – whether to highlight the whole line, up to the right margin
- **speedbar** (*boolean*) – whether to show a mark in the speedbar to the left of editors. This forces `whole_line` to True.
- **kwargs** – other properties supported by EditorOverlay

apply (*start, end*)

Apply the highlighting to part of the buffer.

Parameters

- **start** (*GPS.EditorLocation*) – start of highlighted region.
- **end** (*GPS.EditorLocation*) – end of highlighted region.

remove (*start, end=None*)

Remove the highlighting in whole or part of the buffer. :param `GPS.EditorLocation` start: start of region. :param `GPS.EditorLocation` end: end of region. If unspecified, the highlighting for the whole buffer is removed.

`use_messages ()`

Returns Whether this style will use a *GPS.Message* or a *GPS.EditorOverlay* to highlight.

Return type boolean

class `gps_utils.highlighter.Regexp_Highlighter` (*regexp, style, context_lines=0*)

The `Regexp_Highlighter` is a concrete implementation to highlight editors based on regular expressions. One example is for instance to highlight tabs or trailing spaces on lines, when this is considered improper style:

```
Regexp_Highlighter(  
    regexp=" +|\s+$",  
    style=OverlayStyle(  
        name="tabs style",  
        strikethrough=True,  
        background="#FF7979"))
```

Another example is to highlight TODO lines. Various conventions exist to mark these in the sources, but the following should catch some of these:

```
Regexp_Highlighter(  
    regexp="TODO.*|\?\?\??.*",  
    style=OverlayStyle(  
        name="todo",  
        background="#FF7979"))
```

Another example is a class to highlight Spark comments. This should only be applied when the language is spark:

```
class Spark_Highlighter(Regexp_Highlighter):  
    def must_highlight(self, buffer):  
        return buffer.file().language().lower() == "spark"  
  
Spark_Highlighter(  
    regexp="--#.*$",  
    style=OverlayStyle(  
        name="spark", foreground="red"))
```

Parameters

- **regexp** (*string*) – the regular expression to search for. It should preferably apply to a single line, since highlighting is done on small sections of the editor at a time, and it might not detect cases where the regular expression would match across sections.
- **style** (*OverlayStyle*) – the style to apply.

class `gps_utils.highlighter.Text_Highlighter` (*text, style, whole_word=False, context_lines=0*)

Similar to `Regexp_Highlighter`, but highlights constant text instead of a regular expression. By default, highlighting is done in all buffer, override the function `must_highlight` to reduce the scope.

Parameters

- **text** (*string*) – the text to search for. It should preferably apply to a single line, since highlighting is done on small sections of the editor at a time, and it might not detect cases where the text would match across sections.
- **style** (*OverlayStyle*) – the style to apply.

16.2.3 The `gps_utils.console_process.py` module

class `gps_utils.console_process.ANSI_Console_Process` (*process*, *args*='')

This class has a purpose similar to `Console_Process`. However, this class does not attempt to do any of the high-level processing of prompt and input that `Console_Process` does, and instead forward immediately any of the key strokes within the console directly to the external process. It also provides an ANSI terminal to the external process. The latter can thus send escape sequences to change colors, cursor position,...

class `gps_utils.console_process.Console_Process` (*process*, *args*='', *close_on_exit*=True, *force*=False, *ansi*=False, *manage_prompt*=True)

This class provides a way to spawn an interactive process and do its input/output in a dedicated console in GPS. The process is created so that it does not appear in the task manager, and therefore the user can exit GPS without being asked whether or not to kill the process.

You can of course derive from this class easily. Things are slightly more complicated if you want in fact to derive from a child of `GPS.Console` (for instance a class that would handle ANSI escape sequences). The code would then look like:

```
class ANSI_Console (GPS.Console):
    def write (self, txt): ...

class My_Process (ANSI_Console, Console_Process):
    def __init__ (self, process, args=""):
        Console_Process.__init__ (self, process, args)
```

In the list of base classes for `My_Process`, you must put `ANSI_Console` before `Console_Process`. This is because python resolves overridden methods by looking depth-first search from left to right. This way, it will see `ANSI_Console.write` before `Console_Process.write` and therefore use the former.

However, because of that the `__init__` method that would be called when calling `My_Process (...)` is also that of `ANSI_Console`. Therefore you must define your own `__init__` method locally.

See also the class `ANSI_Console_Process` if you need your process to execute within a terminal that understands ANSI escape sequences.

Parameters

- **force** (*boolean*) – If True, a new console is opened, otherwise an existing one will be reused (although you should take care in this case if you have multiple processes attached to the same console).
- **manage_prompt** (*boolean*) – If True, then GPS will do some higher level handling of prompts: when some output is done by the process, GPS will temporarily hide what the user was typing, insert the output, and append what the user was typing. This is in general suitable but might interfere with external programs that do their own screen management through ANSI commands (like a Unix shell for instance).

on_completion (*input*)

The user has pressed <tab> in the console. The default is just to insert the character, but if you are driving a process that knows about completion, such as an OS shell for instance, you could have a different implementation. *input* is the full input till, but not including, the tab character

on_destroy ()

This method is called when the console is being closed. As a result, we terminate the process (this also results in a call to `on_exit`)

on_exit (*status*, *remaining_output*)

This method is called when the process terminates. As a result, we close the console automatically, although we could decide to keep it open as well

on_input (*input*)

This method is called when the user has pressed <enter> in the console. The corresponding command is then sent to the process

on_interrupt ()

This method is called when the user presses control-c in the console. This interrupts the command we are currently processing

on_key (*keycode, key, modifier*)

The user has pressed a key in the console (any key). This is called before any of the higher level `on_completion` or `on_input` callbacks. If this subprogram returns True, GPS will consider that the key has already been handled and will not do its standard processing with it. By default, we simply let the key through and let GPS handle it.

Parameters `key` – the unicode character (numeric value) that was entered by the user. `_modifier_` is a mask of the control and shift keys that were pressed at the same time. See the Mask constants above. `keycode` is the code of the key, which is useful for non-printable characters. It is set to 0 in some cases if the input is simulated after the user has copied some text into the console

This function is also called for each character pasted by the user in the console. If it returns True, then the selection will not be inserted in the console.

on_output (*matched, unmatched*)

This method is called when the process has emitted some output. The output is then printed to the console

on_resize (*console, rows, columns=None*)

This method is called when the console is being resized. We then let the process know about the size of its terminal, so that it can adapt its output accordingly. This is especially useful with processes like `gdb` or unix shells

GNU FREE DOCUMENTATION LICENSE

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

17.1 PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document ‘free’ in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of ‘copyleft’, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

17.2 APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The ‘Document’, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as ‘you’.

A ‘Modified Version’ of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A ‘Secondary Section’ is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The ‘Invariant Sections’ are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The ‘Cover Texts’ are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A ‘Transparent’ copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not ‘Transparent’ is called ‘Opaque’.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The ‘Title Page’ means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, ‘Title Page’ means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

17.3 VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

17.4 COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of

Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

17.5 MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled 'History', and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled 'History' in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the 'History' section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled 'Acknowledgements' or 'Dedications', preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled 'Endorsements'. Such a section may not be included in the Modified Version.
- Do not retitle any existing section as 'Endorsements' or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled ‘Endorsements’, provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

17.6 COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled ‘History’ in the various original documents, forming one section entitled ‘History’; likewise combine any sections entitled ‘Acknowledgements’, and any sections entitled ‘Dedications’. You must delete all sections entitled ‘Endorsements.’

17.7 COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

17.8 AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an ‘aggregate’, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

17.9 TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

17.10 TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

17.11 FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License ‘or any later version’ applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

17.12 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c) YEAR YOUR NAME.
```

```
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 the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
A copy of the license is included in the section entitled ‘GNU
Free Documentation License’.
```

If you have no Invariant Sections, write ‘with no Invariant Sections’ instead of saying which ones are invariant. If you have no Front-Cover Texts, write ‘no Front-Cover Texts’ instead of ‘Front-Cover Texts being LIST’; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

INDICES AND TABLES

- *genindex*

This document may be copied, in whole or in part, in any form or by any means, as is or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy.

PYTHON MODULE INDEX

a

`auto_highlight_occurrences`, 377

d

`dispatching`, 377

g

GPS, 249

`gps_utils`, 378

`gps_utils.console_process`, 383

`gps_utils.highlighter`, 379

INDEX

Symbols

- eval, 218
- load, 218
- c, 163
- u, 163
- __init__() (GPS.Action method), 259
- __init__() (GPS.Activities method), 260
- __init__() (GPS.AreaContext method), 262
- __init__() (GPS.Bookmark method), 262
- __init__() (GPS.BuildTarget method), 263
- __init__() (GPS.Button method), 264
- __init__() (GPS.CodeAnalysis method), 266
- __init__() (GPS.Codefix method), 267
- __init__() (GPS.CodefixError method), 269
- __init__() (GPS.Combo method), 270
- __init__() (GPS.CommandWindow method), 271
- __init__() (GPS.Console method), 274
- __init__() (GPS.Contextual method), 278
- __init__() (GPS.Debugger method), 281
- __init__() (GPS.DocgenTagHandler method), 285
- __init__() (GPS.EditorBuffer method), 295
- __init__() (GPS.EditorHighlighter method), 301
- __init__() (GPS.EditorLocation method), 302
- __init__() (GPS.EditorMark method), 306
- __init__() (GPS.EditorOverlay method), 307
- __init__() (GPS.EditorView method), 309
- __init__() (GPS.Entity method), 310
- __init__() (GPS.EntityContext method), 315
- __init__() (GPS.File method), 315
- __init__() (GPS.FileContext method), 320
- __init__() (GPS.FileLocation method), 320
- __init__() (GPS.GUI method), 321
- __init__() (GPS.Help method), 323
- __init__() (GPS.Hook method), 334
- __init__() (GPS.Logger method), 338
- __init__() (GPS.MDIWindow method), 342
- __init__() (GPS.Menu method), 343
- __init__() (GPS.Message method), 345
- __init__() (GPS.OutputParserWrapper method), 347
- __init__() (GPS.Preference method), 348
- __init__() (GPS.Process method), 350
- __init__() (GPS.Project method), 353
- __init__() (GPS.Search method), 365
- __init__() (GPS.Search_Result method), 366
- __init__() (GPS.Style method), 366
- __init__() (GPS.SwitchesChooser method), 367
- __init__() (GPS.Timeout method), 369
- __init__() (GPS.ToolButton method), 369
- __init__() (GPS.Toolbar method), 370
- __init__() (GPS.Vdiff static method), 373
- __init__() (GPS.XMLViewer method), 375
- <Language>, 187
- <action>, 170
- <alias>, 190
- <button>, 182
- <case_exceptions>, 199
- <check>, 210
- <choice>, 196
- <combo-entry>, 211
- <combo>, 211
- <contextual>, 181
- <default-value-dependency>, 212
- <dependency>, 211
- <doc_path>, 200
- <documentation_file>, 199
- <entry>, 182, 212
- <expansion>, 212
- <external>, 170
- <field>, 211
- <filter_and>, 177
- <filter_or>, 177
- <filter>, 170, 177
- <index>, 196
- <initial-cmd-line>, 209
- <key>, 183
- <language>, 208
- <menu>, 179
- <popup>, 211
- <pref>, 185
- <preference>, 184
- <project_attribute>, 193
- <radio>, 211
- <shell>, 170, 196

<specialized_index>, 196
 <spin>, 211
 <stock_icons>, 201
 <string>, 195
 <submenu>, 179
 <switches>, 209
 <theme>, 186
 <title>, 179, 210
 <tool>, 207
 <vsearch-pattern>, 186

A

a2ps, 162
 accept_input() (GPS.Console method), 276
 action, 170
 repeat next, 58
 Action (class in GPS), 259
 action_hooks, 231
 ACTIONS (GPS.Search attribute), 364
 active (GPS.Logger attribute), 338
 Activities (class in GPS), 260
 activity, 245
 activity log template, 245
 Ada, 96, 111, 157
 cross-references, 67
 ADA_PROJECT_PATH, 73
 Add To Extending Project, 79
 add() (GPS.Combo method), 270
 add() (GPS.Hook method), 334
 add() (GPS.Locations static method), 336
 add() (GPS.MDI static method), 340
 add_all_gcov_project_info() (GPS.CodeAnalysis
 method), 266
 add_attribute_values() (GPS.Project method), 353
 add_blank_lines() (GPS.Editor static method), 286
 add_case_exception() (GPS.Editor static method), 286
 add_dependency() (GPS.Project method), 354
 add_doc_directory() (GPS.HTML static method), 322
 add_file() (GPS.Activities method), 260
 add_gcov_file_info() (GPS.CodeAnalysis method), 266
 add_gcov_project_info() (GPS.CodeAnalysis
 method), 266
 add_input() (GPS.Console method), 276
 add_link() (GPS.Revision static method), 363
 add_location_command() (in module GPS), 250
 add_log() (GPS.Revision static method), 363
 add_main_unit() (GPS.Project method), 354
 add_multi_cursor() (GPS.EditorBuffer method), 295
 add_predefined_paths() (GPS.Project static method), 354
 add_revision() (GPS.Revision static method), 363
 add_source_dir() (GPS.Project method), 354
 add_special_line() (GPS.EditorBuffer method), 295
 add_templates_dir() (GPS.ProjectTemplate
 static
 method), 362

alias, 56
 Alias (class in GPS), 262
 aliases, 161, 190, 244
 ancestor_deps() (GPS.Project method), 354
 annotate() (GPS.VCS static method), 371
 annotations_parse() (GPS.VCS static method), 371
 ANSI_Console_Process (class
 gps_utils.console_process), 383
 ant, 100
 append() (GPS.Toolbar method), 370
 apply() (gps_utils.highlighter.OverlayStyle method), 381
 apply_overlay() (GPS.EditorBuffer method), 295
 AreaContext (class in GPS), 262
 argument, 174
 as-directory, 211
 as-file, 211
 ASCII, 50, 115
 assembly, 107
 attributes() (GPS.Entity method), 310
 auto, 121
 auto save, 154
 auto_highlight_occurrences (module), 377
 automatic casing
 exceptions, 58
 autosave delay, 156

B

background color, 154
 Background_Highlighter (class in gps_utils.highlighter),
 379
 backward_overlay() (GPS.EditorLocation method), 302
 base_name() (in module GPS), 250
 bash, 33
 beginning_of_buffer() (GPS.EditorBuffer method), 295
 beginning_of_line() (GPS.EditorLocation method), 302
 block folding, 156
 block highlighting, 156
 block_end() (GPS.EditorLocation method), 302
 block_end_line() (GPS.EditorLocation method), 303
 block_exit() (GPS.Task method), 368
 block_fold() (GPS.Editor static method), 286
 block_fold() (GPS.EditorLocation method), 303
 block_get_end() (GPS.Editor static method), 286
 block_get_level() (GPS.Editor static method), 287
 block_get_name() (GPS.Editor static method), 287
 block_get_start() (GPS.Editor static method), 287
 block_get_type() (GPS.Editor static method), 287
 block_level() (GPS.EditorLocation method), 303
 block_name() (GPS.EditorLocation method), 303
 block_start() (GPS.EditorLocation method), 303
 block_start_line() (GPS.EditorLocation method), 303
 block_type() (GPS.EditorLocation method), 303
 block_unfold() (GPS.Editor static method), 287
 block_unfold() (GPS.EditorLocation method), 303

- blocks_fold() (GPS.EditorBuffer method), 295
 - blocks_unfold() (GPS.EditorBuffer method), 296
 - board, 106
 - body() (GPS.Entity method), 310
 - bookmark, 32, 131
 - Bookmark (class in GPS), 262
 - BOOKMARKS (GPS.Search attribute), 364
 - breakpoint, 112, 116, 160, 161
 - breakpoint editor, 112
 - breakpoints, 115
 - browse() (GPS.HTML static method), 322
 - browsers, 78
 - buffer() (GPS.EditorLocation method), 303
 - buffer() (GPS.EditorView method), 309
 - build, 98
 - auto fix errors, 15
 - build modes, 22
 - elaboration circularities, 37
 - executing application, 34
 - hiding warning messages, 15
 - multiple compilers, 103
 - toolbar buttons, 9
 - toolchains, 103
 - build modes, 103
 - build targets, 100
 - BUILDS (GPS.Search attribute), 364
 - BuildTarget (class in GPS), 263
 - Button (class in GPS), 264
- ## C
- C, 111, 115, 160
 - cross-references, 67
 - C++, 96, 160
 - cross-references, 67
 - call graph, 70, 78
 - called_by() (GPS.Entity method), 310
 - called_by_browser() (GPS.Entity method), 311
 - callgraph, 29
 - export, 4
 - calls() (GPS.Entity method), 311
 - case indentation, 158
 - case preserving, 98
 - case sensitive, 98
 - case_exceptions, 199
 - CASE_SENSITIVE (GPS.Search attribute), 364
 - casing
 - automatic, 59
 - category() (GPS.Entity method), 311
 - cd() (in module GPS), 250
 - center() (GPS.EditorView method), 309
 - ChangeLog file, 128
 - character set, 59, 154
 - characters_count() (GPS.EditorBuffer method), 296
 - check() (GPS.Logger method), 339
 - children() (GPS.MDI static method), 341
 - clear() (GPS.CodeAnalysis method), 266
 - clear() (GPS.Combo method), 270
 - clear() (GPS.Console method), 276
 - clear_attribute_values() (GPS.Project method), 354
 - clear_cache command, 219
 - clear_input() (GPS.Console method), 276
 - clear_view() (GPS.Revision static method), 363
 - ClearCase, 121
 - ClearCase Native, 121
 - client/server, 147
 - clipboard, 65, 154
 - Clipboard (class in GPS), 265
 - clone() (GPS.BuildTarget method), 264
 - cloning editors, 46
 - close dialog on match, 98
 - close() (GPS.Debugger method), 282
 - close() (GPS.Editor static method), 287
 - close() (GPS.EditorBuffer method), 296
 - close_editors() (GPS.Vdiff method), 374
 - Code Coverage, 138
 - code coverage, 132
 - code fix, 155
 - code fixing, 134
 - code folding, 55
 - CodeAnalysis (class in GPS), 266
 - codefix, 15
 - Codefix (class in GPS), 267
 - Codefix.errors, 216
 - CodefixError (class in GPS), 269
 - CodefixError.fix, 216
 - CodefixError.possible_fixes, 216
 - Coding Standard, 132
 - coding standard, 132
 - color, 155, 157, 161, 162, 164
 - column index, 164
 - column() (GPS.EditorLocation method), 303
 - column() (GPS.FileLocation method), 320
 - Combo (class in GPS), 269
 - Command (class in GPS), 271
 - command line, **243**
 - P, 6
 - command() (GPS.Debugger method), 282
 - CommandWindow (class in GPS), 271
 - commit() (GPS.Activities method), 260
 - commit() (GPS.VCS static method), 371
 - compare, 126
 - compilation, 98
 - compile() (GPS.File method), 316
 - complete block, 55
 - complete identifier, 55
 - completion, 49, 53
 - compute_xref() (in module GPS), 250
 - conditional line, 158

- console, 13
 - Console (class in GPS), 273
 - Console_Process (class in `gps_utils.console_process`), 383
 - contents() (GPS.Clipboard static method), 265
 - Context (class in GPS), 277
 - context length, 164
 - Contextual (class in GPS), 278
 - contextual menu
 - browsers → called by, 32
 - browsers → calls, 32
 - browsers → calls (recursively), 32
 - called by, 30
 - calls, 30
 - contextual menus, 230
 - contextual() (GPS.Action method), 259
 - contextual_context() (in module GPS), 250
 - continuation line, 158
 - copy, 65
 - copy() (GPS.Clipboard static method), 265
 - copy() (GPS.Editor static method), 288
 - copy() (GPS.EditorBuffer method), 296
 - copy_clipboard() (GPS.Console method), 276
 - core file, 106
 - count (GPS.Logger attribute), 338
 - create() (GPS.Action method), 259
 - create() (GPS.Bookmark static method), 262
 - create() (GPS.Contextual method), 278
 - create() (GPS.Menu static method), 343
 - create() (GPS.Preference method), 348
 - create() (GPS.Vdiff static method), 374
 - create_dynamic() (GPS.Contextual method), 279
 - create_link() (GPS.Console method), 276
 - create_mark() (GPS.Editor static method), 288
 - create_mark() (GPS.EditorLocation method), 303
 - create_metric() (GPS.XMLViewer method), 376
 - create_overlay() (GPS.EditorBuffer method), 296
 - creating, 233
 - cross debugger, 106
 - cross environment, 145
 - cross-references, 67
 - CSS, 245
 - current line, 49
 - current() (GPS.Clipboard static method), 265
 - current() (GPS.MDI static method), 341
 - current_context() (in module GPS), 252
 - current_view() (GPS.EditorBuffer method), 296
 - cursor() (GPS.EditorView method), 309
 - cursor_center() (GPS.Editor static method), 288
 - cursor_get_column() (GPS.Editor static method), 288
 - cursor_get_line() (GPS.Editor static method), 288
 - cursor_set_position() (GPS.Editor static method), 288
 - customization, 153, 168
 - cut, 65
 - cut() (GPS.Editor static method), 289
 - cut() (GPS.EditorBuffer method), 297
 - CVS, 121
- ## D
- debugger, 146, 247
 - call stack, 108
 - data window, 109
 - toolbar buttons, 9
 - Debugger (class in GPS), 281
 - debugger console, 118
 - debugger windows, 161
 - debugging, 104
 - declaration line, 158
 - declaration() (GPS.Entity method), 311
 - default, 246
 - default desktop, 246
 - Default VCS, 163
 - delete() (GPS.Bookmark method), 263
 - delete() (GPS.EditorBuffer method), 297
 - delete() (GPS.EditorMark method), 306
 - delete() (in module GPS), 252
 - delete_mark() (GPS.Editor static method), 289
 - delimiter, 49
 - dependencies() (GPS.Project method), 355
 - derived_types() (GPS.Entity method), 311
 - describe_functions() (GPS.Hook method), 335
 - description, 173
 - desktop, *see* Multiple Document Interface, 154, 246
 - destroy() (GPS.GUI method), 321
 - dialog() (GPS.MDI static method), 341
 - diff, 163
 - diff3, 163
 - diff_head() (GPS.VCS static method), 371
 - diff_working() (GPS.VCS static method), 371
 - dir() (in module GPS), 252
 - dir_name() (in module GPS), 252
 - directory() (GPS.File method), 316
 - directory() (GPS.FileContext method), 320
 - discriminants() (GPS.Entity method), 311
 - dispatching, 71
 - dispatching (module), 377
 - display line numbers, 156
 - Display subprogram names, 156
 - do_highlight() (`gps_utils.highlighter.On_The_Fly_Highlighter` method), 381
 - Docgen (class in GPS), 283
 - DocgenTagHandler (class in GPS), 284
 - documentation, 132
 - documentation generation, 135
 - documentation() (GPS.Entity method), 311
 - drag-and-drop, 17, 45
 - dump() (GPS.Locations static method), 337
 - dump() (in module GPS), 253

dump_file() (in module GPS), 253
 dump_to_file() (GPS.CodeAnalysis method), 266

E

edit() (GPS.Editor static method), 289
 editing, 47, 50
 editor, 59
 Editor (class in GPS), 286
 EditorBuffer (class in GPS), 295
 EditorHighlighter (class in GPS), 301
 EditorLocation (class in GPS), 302
 EditorMark (class in GPS), 306
 EditorOverlay (class in GPS), 307
 EditorView (class in GPS), 308
 emacs, 50, 64
 emacsclient, 64
 enable_input() (GPS.Console method), 276
 end_line() (GPS.AreaContext method), 262
 end_of_buffer() (GPS.EditorBuffer method), 297
 end_of_line() (GPS.EditorLocation method), 304
 end_of_scope() (GPS.Entity method), 312
 ends_word() (GPS.EditorLocation method), 304
 ENTITIES (GPS.Search attribute), 364
 entities() (GPS.File method), 316
 Entity (class in GPS), 310
 entity() (GPS.EntityContext method), 315
 EntityContext (class in GPS), 315
 environment, 243
 environment variables, 243
 error_at() (GPS.Codefix method), 268
 errors, 164
 errors() (GPS.Codefix method), 268
 example, 176
 exception, 161
 Exception (class in GPS), 315
 exec_in_console() (in module GPS), 253
 execute() (GPS.BuildTarget method), 264
 execute_action, 218
 execute_action() (GPS.Message method), 345
 execute_action() (in module GPS), 253
 execute_asynchronous_action() (in module GPS), 253
 execute_for_all_cursors() (in module gps_utils), 378
 execute_if_possible() (GPS.Action method), 260
 execution, 161, 162
 exit() (in module GPS), 254
 expand_alias() (GPS.EditorBuffer method), 297
 expect() (GPS.Process method), 352
 external, 171
 external editor, 64, 156
 external tool, 207

F

fast project loading, 164
 fields() (GPS.Entity method), 312

File (class in GPS), 315
 file index, 164
 file pattern, 164
 file selector, 40
 file() (GPS.EditorBuffer method), 297
 file() (GPS.FileContext method), 320
 file() (GPS.FileLocation method), 321
 file() (GPS.Help method), 323
 file() (GPS.Project method), 355
 FILE_NAMES (GPS.Search attribute), 365
 file_selector() (GPS.MDI static method), 341
 FileContext (class in GPS), 320
 FileLocation (class in GPS), 320
 files() (GPS.Activities method), 260
 files() (GPS.FileContext method), 320
 files() (GPS.Vdiff method), 374
 filter, 173
 find_all_refs() (GPS.Entity method), 312
 finish_undo_group() (GPS.EditorBuffer method), 297
 fix() (GPS.CodefixError method), 269
 float, 155
 float() (GPS.MDIWindow method), 342
 flush() (GPS.Console method), 276
 font, 154, 157
 forward_char() (GPS.EditorLocation method), 304
 forward_line() (GPS.EditorLocation method), 304
 forward_overlay() (GPS.EditorLocation method), 304
 forward_word() (GPS.EditorLocation method), 304
 freeze_prefs() (in module GPS), 254
 freeze_prefs() (in module gps_utils), 378
 freeze_xref() (in module GPS), 254
 from_file() (GPS.Activities static method), 260
 full_name() (GPS.Entity method), 312
 FUZZY (GPS.Search attribute), 365

G

gcc
 -fdump-xref, 53, 67
 generate body, 55
 generate_doc() (GPS.File method), 316
 generate_doc() (GPS.Project method), 355
 generate_index_file() (GPS.Docgen method), 283
 generic_vcs, 234
 get() (GPS.Activities static method), 260
 get() (GPS.Alias static method), 262
 get() (GPS.Bookmark static method), 263
 get() (GPS.CodeAnalysis static method), 266
 get() (GPS.Command static method), 271
 get() (GPS.Debugger static method), 282
 get() (GPS.EditorBuffer static method), 297
 get() (GPS.MDI static method), 341
 get() (GPS.Menu static method), 344
 get() (GPS.Preference method), 349
 get() (GPS.Search method), 365

- get() (GPS.Toolbar method), 370
- get() (GPS.Vdiff static method), 374
- get_active() (GPS.Menu method), 344
- get_attribute_as_list, 213
- get_attribute_as_list() (GPS.Project method), 355
- get_attribute_as_string, 213
- get_attribute_as_string() (GPS.Project method), 356
- get_background() (GPS.Style method), 366
- get_buffer() (GPS.Editor static method), 289
- get_build_mode() (in module GPS), 254
- get_build_output() (in module GPS), 254
- get_busy() (in module GPS), 255
- get_by_child() (GPS.MDI static method), 341
- get_by_pos() (GPS.Toolbar method), 370
- get_category() (GPS.Message method), 345
- get_char() (GPS.EditorLocation method), 305
- get_chars() (GPS.Editor static method), 289
- get_chars() (GPS.EditorBuffer method), 298
- get_child() (GPS.MDIWindow method), 342
- get_cmd_line() (GPS.SwitchesChooser method), 367
- get_column() (GPS.Editor static method), 290
- get_column() (GPS.Message method), 345
- get_current_file() (GPS.Docgen method), 284
- get_current_vcs() (GPS.VCS static method), 371
- get_doc_dir() (GPS.Docgen method), 284
- get_executable() (GPS.Debugger method), 282
- get_executable_name() (GPS.Project method), 357
- get_extend_selection() (GPS.EditorView method), 309
- get_file() (GPS.Editor static method), 290
- get_file() (GPS.Message method), 345
- get_flags() (GPS.Message method), 345
- get_foreground() (GPS.Style method), 367
- get_home_dir() (in module GPS), 255
- get_in_speedbar() (GPS.Style method), 367
- get_last_line() (GPS.Editor static method), 290
- get_line() (GPS.Editor static method), 290
- get_line() (GPS.Message method), 346
- get_log_file() (GPS.VCS static method), 372
- get_mark() (GPS.EditorBuffer method), 298
- get_mark() (GPS.Message method), 346
- get_multi_cursors_marks() (GPS.EditorBuffer method), 298
- get_name() (GPS.Style method), 367
- get_new() (GPS.EditorBuffer static method), 298
- get_num() (GPS.Debugger method), 282
- get_overlays() (GPS.EditorLocation method), 305
- get_property() (GPS.EditorOverlay method), 307
- get_property() (GPS.File method), 316
- get_property() (GPS.Project method), 357
- get_result() (GPS.Command method), 271
- get_result() (GPS.Process method), 352
- get_result() (GPS.ReferencesCommand method), 363
- get_status() (GPS.VCS static method), 372
- get_system_dir() (in module GPS), 255
- get_text() (GPS.Combo method), 270
- get_text() (GPS.Console method), 276
- get_text() (GPS.Message method), 346
- get_tmp_dir() (in module GPS), 255
- get_tool_switches_as_list, 213
- get_tool_switches_as_list() (GPS.Project method), 357
- get_tool_switches_as_string, 213
- get_tool_switches_as_string() (GPS.Project method), 357
- getdoc() (GPS.Help method), 324
- gif, 182
- Git, 121
- global ChangeLog, 128
- GNAT
 - g, 247
 - gnatQ, 67
 - k, 67
 - ALI files, 67
- GNAT_CODE_PAGE, 244
- gnatkr, 67
- gnatmake, 247
- gnatname, 85
- gnatpp, 55
- gnatstub, 55
- gnattest, 132
- gnuclient, 64
- goto body, 68
- goto declaration, 68
- goto line, 69
- goto() (GPS.Bookmark method), 263
- goto() (GPS.EditorView method), 309
- goto_mark() (GPS.Editor static method), 290
- GPR_PROJECT_PATH, 73
- GPS (module), 249
- gps shell, 219
- GPS_CHANGELOG_USER, 244
- GPS_CUSTOM_PATH, 244
- GPS_DOC_PATH, 244
- GPS_HOME, 244
- GPS_MEMORY_MONITOR, 244
- GPS_PYTHONHOME, 244
- GPS_ROOT, 244
- GPS_STARTUP_LD_LIBRARY_PATH, 244
- GPS_STARTUP_PATH, 244
- gps_utils (module), 378
- gps_utils.console_process (module), 383
- gps_utils.highlighter (module), 379
- graph disable, 119
- graph display, 118
- graph enable, 119
- graph print, 118
- graph undisplay, 119
- group_commit() (GPS.Activities method), 261
- GROUP_CONSOLES (GPS.MDI attribute), 340
- GROUP_DEBUGGER_DATA (GPS.MDI attribute), 340

- GROUP_DEBUGGER_STACK (GPS.MDI attribute), 340
- GROUP_DEFAULT (GPS.MDI attribute), 340
- GROUP_GRAPHS (GPS.MDI attribute), 340
- GROUP_VCS_ACTIVITIES (GPS.MDI attribute), 340
- GROUP_VCS_EXPLORER (GPS.MDI attribute), 340
- GROUP_VIEW (GPS.MDI attribute), 340
- GUI (class in GPS), 321
- ## H
- has_log() (GPS.Activities method), 261
- has_overlay() (GPS.EditorLocation method), 305
- Help (class in GPS), 323
- hexadecimal, 50, 115
- hidden directories pattern, 165
- hide() (GPS.BuildTarget method), 264
- hide() (GPS.Contextual method), 280
- hide() (GPS.GUI method), 321
- hide() (GPS.MDI static method), 341
- hide_coverage_information() (GPS.CodeAnalysis method), 267
- highlight delimiter, 156
- highlight() (GPS.Editor static method), 290
- highlight_range() (GPS.Editor static method), 291
- history, 245
- HOME, 244
- Hook (class in GPS), 324
- Hook.describe, 230
- Hook.list, 230
- Hook.list_types, 230
- Hook.register, 233
- Hook.run, 232
- hooks, 230–233
- html, 162
- HTML (class in GPS), 322
- hyper mode, 154
- hyperlinks, 71
- ## I
- id() (GPS.Activities method), 261
- Implicit status, 163
- imported entities, 70
- imported_by() (GPS.File method), 316
- imports() (GPS.File method), 317
- in_ada_file() (in module gps_utils), 378
- in_xml_file() (in module gps_utils), 378
- indent() (GPS.Editor static method), 291
- indent() (GPS.EditorBuffer method), 298
- indent_buffer() (GPS.Editor static method), 291
- indentation, 48, 157, 160
- indentation level, 158
- indexed, 196
- indexed project attributes, 196
- input_dialog, 214
- input_dialog() (GPS.MDI static method), 341
- insert() (GPS.EditorBuffer method), 299
- insert() (GPS.Toolbar method), 371
- insert_text() (GPS.Editor static method), 291
- inside_word() (GPS.EditorLocation method), 305
- insmod() (in module GPS), 255
- interactive (class in gps_utils), 378
- interactive command, 172
- interactive search, 123
- interrupt, 132
- interrupt() (GPS.Command method), 271
- interrupt() (GPS.Process method), 352
- interrupt() (GPS.Task method), 368
- Invalid_Argument (class in GPS), 336
- is_access() (GPS.Entity method), 312
- is_array() (GPS.Entity method), 312
- is_break_command() (GPS.Debugger method), 282
- is_busy() (GPS.Debugger method), 282
- is_closed() (GPS.Activities method), 261
- is_container() (GPS.Entity method), 312
- is_context_command() (GPS.Debugger method), 282
- is_exec_command() (GPS.Debugger method), 282
- is_floating() (GPS.MDIWindow method), 342
- is_generic() (GPS.Entity method), 312
- is_global() (GPS.Entity method), 312
- is_modified() (GPS.EditorBuffer method), 299
- is_modified() (GPS.Project method), 358
- is_predefined() (GPS.Entity method), 312
- is_present() (GPS.EditorMark method), 306
- is_read_only() (GPS.EditorBuffer method), 299
- is_read_only() (GPS.EditorView method), 309
- is_sensitive() (GPS.GUI method), 321
- is_server_local() (in module GPS), 256
- is_subprogram() (GPS.Entity method), 312
- is_type() (GPS.Entity method), 312
- is_writable() (in module gps_utils), 378
- isatty() (GPS.Console method), 276
- ISO-8859-1, 154
- ## J
- jpeg, 182
- ## K
- key, 50, 183
- key shortcuts, 56
- key() (GPS.Action method), 260
- kill() (GPS.Process method), 352
- ## L
- language, 59
- language() (GPS.File method), 317
- languages() (GPS.Project method), 358
- last_command() (in module GPS), 256
- line index, 164

- line terminator, 156
 - line() (GPS.EditorLocation method), 305
 - line() (GPS.FileLocation method), 321
 - lines_count() (GPS.EditorBuffer method), 299
 - list() (GPS.Activities static method), 261
 - list() (GPS.Bookmark static method), 263
 - list() (GPS.Command static method), 271
 - list() (GPS.Contextual static method), 281
 - list() (GPS.Debugger static method), 283
 - list() (GPS.EditorBuffer static method), 299
 - list() (GPS.Hook static method), 335
 - list() (GPS.Message static method), 346
 - list() (GPS.Style static method), 367
 - list() (GPS.Task static method), 368
 - list() (GPS.Vdiff static method), 374
 - list_categories() (GPS.Locations static method), 337
 - list_locations() (GPS.Locations static method), 337
 - list_types() (GPS.Hook static method), 335
 - literals() (GPS.Entity method), 312
 - load() (GPS.Project static method), 358
 - load_from_file() (GPS.CodeAnalysis method), 267
 - locate in project view, 19
 - location, 164
 - location() (GPS.CodefixError method), 269
 - location() (GPS.EditorMark method), 306
 - location() (GPS.FileContext method), 320
 - Location_Highlighter (class in gps_utils.highlighter), 380
 - Locations (class in GPS), 336
 - log file, 244
 - log template, 245
 - log() (GPS.Activities method), 261
 - log() (GPS.Logger method), 339
 - log() (GPS.VCS static method), 372
 - log_file() (GPS.Activities method), 261
 - log_parse() (GPS.VCS static method), 372
 - Logger (class in GPS), 338
 - long (GPS.Search_Result attribute), 366
 - look in, 96
 - lookup() (GPS.Search static method), 365
 - lookup_actions() (in module GPS), 256
 - lookup_actions_from_key() (in module GPS), 256
 - ls() (in module GPS), 257
 - lsmod() (in module GPS), 257
- ## M
- macros, 58, 132
 - make() (GPS.File method), 317
 - make_interactive() (in module gps_utils), 378
 - makefile, 100
 - mark_current_location() (GPS.Editor static method), 291
 - MDI, *see* Multiple Document Interface, 41, 155
 - closing windows, 44
 - floating windows, 44
 - perspectives, 46
 - selecting windows, 43
 - MDI (class in GPS), 339
 - MDI.save_all, 213
 - MDIWindow (class in GPS), 342
 - memory view, 115
 - menu, 125, 174
 - build -> ant, 100
 - build -> check semantic, 99
 - build -> check syntax, 99
 - build -> clean -> clean all, 100
 - build -> clean -> clean root, 100
 - build -> compile file, 99
 - build -> makefile, 100
 - build -> project -> <main>, 99
 - build -> project -> build <current file>, 99
 - build -> project -> build all, 99
 - build -> project -> compile all sources, 99
 - build -> project -> custom build, 99
 - build -> run, 34
 - build -> run -> <main>, 100
 - build -> run -> custom, 100
 - build -> settings -> targets, 100
 - build -> settings -> toolchains, 100, 103
 - debug -> continue, 106
 - debug -> data -> assembly, 107, 117
 - debug -> data -> call stack, 107, 109
 - debug -> data -> command history, 108
 - debug -> data -> data window, 107
 - debug -> Data -> display any expression, 108, 118
 - debug -> data -> display arguments, 108
 - debug -> data -> display local variables, 108
 - debug -> data -> display registers, 108
 - debug -> data -> display registers, 118
 - debug -> data -> edit breackpoints, 113
 - debug -> data -> edit breakpoints, 107
 - debug -> data -> examine memory, 108
 - debug -> data -> protection domains, 107
 - debug -> data -> recompute, 108
 - debug -> data -> tasks, 107
 - debug -> data -> threads, 107
 - debug -> debug -> add symbols, 106
 - debug -> debug -> attach, 106
 - debug -> debug -> connect to board, 106
 - debug -> debug -> debug core file, 106
 - debug -> debug -> detach, 106
 - debug -> debug -> kill, 107
 - debug -> debug -> load file, 105, 106
 - debug -> finish, 106
 - debug -> initialize, 105
 - debug -> initialize -> no main file, 106
 - debug -> interrupt, 106
 - debug -> next, 106
 - debug -> next instruction, 106
 - debug -> run, 105

- debug → step, 105
- debug → step instruction, 105
- debug → terminate, 106
- debug → terminate, 105
- debug → terminate current, 105, 106
- edit → aliases, 56
- edit → copy, 29, 52
- edit → create bookmark, 32, 55
- edit → cut, 29, 52
- edit → edit with external editor, 56
- edit → fold all blocks, 55
- edit → format selection, 53
- edit → generate body, 55
- edit → insert file, 53
- edit → insert shell output, 53
- edit → key shortcuts, 56
- edit → more completion, 54
- edit → more completion → complete block, 55
- edit → more completion → complete identifier, 55
- edit → more completion → expand alias, 55
- edit → paste, 29, 52
- edit → paste previous, 29, 52
- edit → preferences, 56
- edit → pretty print, 55
- edit → rectangles, 52, 56
- edit → rectangles → serialize, 52
- edit → redo, 52
- edit → select all, 53
- edit → selection, 55
- edit → selection → comment lines, 55
- edit → selection → move left, 55
- edit → selection → move right, 55
- edit → selection → pipe in external program, 55
- edit → selection → refill, 55
- edit → selection → sort, 55
- edit → selection → sort reverse, 55
- edit → selection → uncomment lines, 55
- edit → selection → untabify, 55
- edit → smart completion, 53
- edit → undo, 52
- edit → unfold all blocks, 55
- file → change directory, 51
- file → close, 51
- file → exit, 52
- file → locations, 51
- file → new, 50
- file → new view, 51
- file → open, 51
- file → open from host, 51
- file → open from project, 11, 12, 51, 75
- file → print, 51
- file → recent, 51
- file → save, 51, 65
- file → save as, 51, 65
- file → save more, 51
- file → save more → all, 65
- navigate → back, 69
- navigate → end of statement, 69
- navigate → find all references, 68
- navigate → find next, 68
- navigate → find or replace, 19, 68, 95
- navigate → find previous, 68
- navigate → forward, 69
- navigate → goto body, 68
- navigate → goto declaration, 68
- navigate → goto entity, 11, 69
- navigate → goto file spec<->body, 69
- navigate → goto line, 69
- navigate → goto matching delimiter, 68
- navigate → next subprogram, 69
- navigate → next tag, 15, 69
- navigate → previous subprogram, 69
- navigate → previous tag, 15, 69
- navigate → start of statement, 69
- project → edit file switches, 81
- project → edit project properties, 19, 81, 90
- project → new, 80
- project → new from template, 80
- project → open, 80
- project → project view, 17, 81
- project → recent, 81
- project → reload project, 81
- project → save all, 81
- project → save_all, 19
- tools, 3
- tools → consoles → auxiliary builds, 104
- tools → consoles → background builds, 102
- tools → consoles → GPS Shell, 33
- tools → consoles → OS Shell, 34
- tools → consoles → Python, 33
- tools → interrupt, 100
- tools → macros, 58
- tools → views, 3
- tools → views → bookmarks, 33
- tools → views → clipboard, 29
- tools → views → files, 23
- tools → views → messages, 14
- tools → views → outline, 28
- tools → views → project, 17
- tools → views → tasks, 34, 100
- tools → views → windows, 25
- window → close, 44
- window → floating, 45
- window → perspectives, 46
- window → perspectives → create new, 46
- windows → split horizontally, 44
- windows → split vertically, 44
- Menu (class in GPS), 343

- menu bar, 8
 - menu separator, 181
 - menu() (GPS.Action method), 260
 - menus, 179
 - Mercurial, 121
 - merge() (GPS.Clipboard static method), 265
 - Message (class in GPS), 345
 - message() (GPS.CodefixError method), 269
 - MESSAGE_IN_LOCATIONS (GPS.Message attribute), 345
 - MESSAGE_IN_SIDEBAR (GPS.Message attribute), 345
 - MESSAGE_IN_SIDEBAR_AND_LOCATIONS (GPS.Message attribute), 345
 - MESSAGE_INVISIBLE (GPS.Message attribute), 345
 - messages, 13
 - methods, 70
 - methods() (GPS.Entity method), 313
 - Metrics, 138
 - metrics, 132
 - Missing_Arguments (class in GPS), 347
 - Mode, 206
 - Model, 205
 - module_name (GPS.Context attribute), 277
 - move() (GPS.EditorMark method), 306
 - multi-unit source files, 86
 - Multiple Document Interface, 3, *see* MDI, 98, 155
 - must_highlight() (gps_utils.highlighter.On_The_Fly_Highlighter method), 381
- ## N
- name() (GPS.Activities method), 261
 - name() (GPS.Bookmark method), 263
 - name() (GPS.Command method), 271
 - name() (GPS.EditorOverlay method), 307
 - name() (GPS.Entity method), 313
 - name() (GPS.File method), 317
 - name() (GPS.MDIWindow method), 343
 - name() (GPS.Project method), 358
 - name() (GPS.Task method), 368
 - name_parameters() (GPS.Entity method), 313
 - navigation, 67
 - network, 147
 - next() (GPS.MDIWindow method), 343
 - next() (GPS.Search method), 365
 - non_blocking_send() (GPS.Debuggger method), 283
- ## O
- object_dirs() (GPS.Project method), 359
 - offset() (GPS.EditorLocation method), 305
 - old diff, 163
 - omni-search, 9
 - on-failure, 172
 - on_completion() (gps_utils.console_process.Console_Process method), 383
 - on_destroy() (gps_utils.console_process.Console_Process method), 383
 - on_exit() (GPS.OutputParserWrapper method), 347
 - on_exit() (gps_utils.console_process.Console_Process method), 383
 - on_input() (gps_utils.console_process.Console_Process method), 383
 - on_interrupt() (gps_utils.console_process.Console_Process method), 384
 - on_key() (gps_utils.console_process.Console_Process method), 384
 - on_output() (gps_utils.console_process.Console_Process method), 384
 - on_resize() (gps_utils.console_process.Console_Process method), 384
 - on_start_buffer() (gps_utils.highlighter.Background_Highlighter method), 379
 - on_stderr() (GPS.OutputParserWrapper method), 347
 - on_stdout() (GPS.OutputParserWrapper method), 348
 - On_The_Fly_Highlighter (class in gps_utils.highlighter), 380
 - opaque, 155
 - open_file_action_hook, 231
 - OPENED (GPS.Search attribute), 365
 - options, *see* command line
 - other_file() (GPS.File method), 317
 - outline, 25
 - outline view, 25
 - output, 215
 - OutputParserWrapper (class in GPS), 347
 - OverlayStyle (class in gps_utils.highlighter), 381
 - overriding operations, 70
- ## P
- parameters() (GPS.Entity method), 313
 - parent_types() (GPS.Entity method), 313
 - parse() (GPS.Codefix static method), 268
 - parse() (GPS.Locations static method), 337
 - parse() (GPS.XMLViewer method), 376
 - parse_string() (GPS.XMLViewer method), 376
 - parse_xml() (in module GPS), 257
 - password, 122, 147, 171, 203
 - paste, 65
 - paste() (GPS.Editor static method), 291
 - paste() (GPS.EditorBuffer method), 299
 - patch, 163
 - path, 204
 - pause() (GPS.Task method), 368
 - perspectives, 46
 - plug-ins, 132, 167
 - auto_highlight_occurrences.py, 49
 - dispatching.py, 71
 - emacs.py, 56
 - makefile.py, 100

- methods.py, 69, 70
 - shell.py, 33
 - png, 182
 - pointed_type() (GPS.Entity method), 313
 - POSITION_AUTOMATIC (GPS.MDI attribute), 340
 - POSITION_BOTTOM (GPS.MDI attribute), 340
 - POSITION_LEFT (GPS.MDI attribute), 340
 - POSITION_RIGHT (GPS.MDI attribute), 340
 - POSITION_TOP (GPS.MDI attribute), 340
 - possible_fixes() (GPS.CodefixError method), 269
 - predefined patterns, 186
 - Preference (class in GPS), 348
 - preferences, 153, 245
 - browsers → show elaboration cycles, 38
 - clipboard size, 29
 - debugger → debugger windows, 106
 - debugger → preserve state on exit, 109, 115
 - display welcome window, 7
 - documentation → leading documentation, 49
 - editor → ada → casing policy, 59
 - editor → ada → identifier casing, 59
 - editor → ada → reserved word casing, 59
 - editor → always use external editor, 65
 - editor → autosave delay, 65
 - editor → block folding, 49
 - editor → block highlighting, 49
 - editor → custom editor command, 64
 - editor → display line numbers, 47
 - editor → display subprogram names, 48
 - editor → external editor, 64
 - editor → fonts & colors → current line color, 49
 - editor → highlight delimiters, 49
 - editor → speed column policy, 48
 - editor → tooltips, 49
 - general → clipboard size, 52
 - general → hyper links, 71
 - general → save desktop on exit, 46, 51
 - search → preserve search context, 96
 - tip of the day, 8
 - windows → all floating, 45
 - windows → destroy floats, 45
 - windows → show title bars, 44
 - pretty print, 55
 - primitive operations, 70
 - primitive_of() (GPS.Entity method), 313
 - print, 51, 162
 - print_line_info() (GPS.Editor static method), 291
 - PrintFile, 162
 - problems, 246
 - Process (class in GPS), 349
 - process() (gps_utils.highlighter.Background_Highlighter method), 379
 - progress bar, 9
 - progress() (GPS.Command method), 271
 - progress() (GPS.Task method), 368
 - project, 145
 - attribute, 76
 - comments, 74
 - creating scenario variables, 76
 - default, 6
 - default project, 246
 - dependencies, 36, 89
 - description, 73
 - editing, 80, 90
 - editing scenario variable, 77
 - exec directory, 84
 - extending, 79
 - languages, 75
 - library, 85
 - load existing project, 7
 - main units, 84
 - naming scheme, 85
 - normalization, 73
 - object directory, 84
 - reload, 19
 - saving, 19
 - scenario variable, 19, 76
 - scenario variables, 19
 - startup, 6
 - subprojects, 74
 - switches, 87
 - viewing dependencies, 35
 - wizard, 7, **81**
 - Project (class in GPS), 353
 - project attributes, 193, 196
 - project templates, 241
 - project view, 15, 95
 - absolute paths, 18
 - flat view, 18
 - project() (GPS.File method), 318
 - project() (GPS.FileContext method), 320
 - projects
 - limited with, 18
 - ProjectTemplate (class in GPS), 362
 - properties_editor() (GPS.Project method), 359
 - protection domain, 107
 - pwd() (in module GPS), 257
 - pyobject, 230
 - python, 132, 220, 225
 - console, 33
 - pywidget() (GPS.GUI method), 322
- ## R
- raise_window() (GPS.MDIWindow method), 343
 - range size, 161
 - read() (GPS.CommandWindow method), 272
 - read() (GPS.Console method), 277
 - readline() (GPS.Console method), 277

- recompute() (GPS.Project static method), 359
 - recompute() (GPS.Vdiff method), 375
 - recompute_refs() (gps_utils.highlighter.Location_Highlighter method), 380
 - record indentation, 158
 - rectangle, 56
 - redo() (GPS.Editor static method), 291
 - redo() (GPS.EditorBuffer method), 299
 - refactoring, 59
 - references() (GPS.Entity method), 313
 - references() (GPS.File method), 318
 - ReferencesCommand (class in GPS), 362
 - refill() (GPS.Editor static method), 292
 - refill() (GPS.EditorBuffer method), 299
 - REGEXP (GPS.Search attribute), 365
 - Regexp_Highlighter (class in gps_utils.highlighter), 382
 - register() (GPS.Hook static method), 335
 - register() (GPS.Search static method), 365
 - register_css() (GPS.Docgen static method), 284
 - register_highlighting() (GPS.Editor static method), 292
 - register_main_index() (GPS.Docgen static method), 284
 - register_tag_handler() (GPS.Docgen static method), 284
 - regular expression, 97
 - relative project path, 164
 - remote, 147, 202–204
 - remote copy, 162
 - remote project, 150
 - remote shell, 162
 - remove() (GPS.BuildTarget method), 264
 - remove() (GPS.Combo method), 270
 - remove() (GPS.EditorHighlighter method), 302
 - remove() (GPS.Hook method), 335
 - remove() (GPS.Message method), 346
 - remove() (GPS.Timeout method), 369
 - remove() (gps_utils.highlighter.OverlayStyle method), 381
 - remove_all_multi_cursors() (GPS.EditorBuffer method), 299
 - remove_annotations() (GPS.VCS static method), 372
 - remove_attribute_values() (GPS.Project method), 359
 - remove_blank_lines() (GPS.Editor static method), 292
 - remove_case_exception() (GPS.Editor static method), 292
 - remove_category() (GPS.Locations static method), 338
 - remove_dependency() (GPS.Project method), 359
 - remove_file() (GPS.Activities method), 261
 - remove_highlight() (gps_utils.highlighter.Background_Highlighter method), 380
 - remove_interactive() (in module gps_utils), 378
 - remove_overlay() (GPS.EditorBuffer method), 299
 - remove_property() (GPS.File method), 318
 - remove_property() (GPS.Project method), 359
 - remove_source_dir() (GPS.Project method), 360
 - remove_special_lines() (GPS.EditorBuffer method), 300
 - removing variable, 78
 - rename() (GPS.Bookmark method), 263
 - rename() (GPS.Entity method), 314
 - rename() (GPS.MDIWindow method), 343
 - rename() (GPS.Menu method), 344
 - rename() (GPS.Project method), 360
 - renaming entities
 - in callgraph, 32
 - repeat_next() (in module GPS), 257
 - replace, 162
 - replace with, 96
 - replace_text() (GPS.Editor static method), 292
 - repository_dir() (GPS.VCS static method), 372
 - repository_path() (GPS.VCS static method), 372
 - reset() (GPS.Help method), 324
 - reset_xref_db() (in module GPS), 257
 - resume() (GPS.Task method), 368
 - return_type() (GPS.Entity method), 314
 - Revision (class in GPS), 363
 - revision_parse() (GPS.VCS static method), 372
 - right margin, 156
 - root() (GPS.Project static method), 360
 - rsync, 203
 - run, 34
 - run() (GPS.Hook method), 335
 - run_until_failure() (GPS.Hook method), 336
 - run_until_success() (GPS.Hook method), 336
- ## S
- save() (GPS.Editor static method), 292
 - save() (GPS.EditorBuffer method), 300
 - save_all() (GPS.MDI static method), 342
 - save_buffer() (GPS.Editor static method), 293
 - save_current_window() (in module gps_utils), 378
 - save_dir() (in module gps_utils), 378
 - save_excursion() (in module gps_utils), 379
 - save_persistent_properties() (in module GPS), 257
 - saving, 65, 115
 - automatic, 65
 - saving breakpoints, 115
 - scenario_variables() (GPS.Project static method), 360
 - scenario_variables_cmd_line() (GPS.Project static method), 360
 - scenario_variables_values() (GPS.Project static method), 360
 - screen shot, 122, 124, 129, 134, 136, 140, 142, 148, 151, 153, 166, 190
 - scripts, 218
 - search, *see* omni-search, 162
 - interactive search in trees, 17
 - project view, 19
 - Search (class in GPS), 363
 - search context, 95, 96
 - search for, 95

- search() (GPS.EditorLocation method), 305
 search() (GPS.File method), 318
 search() (GPS.Project method), 361
 search() (GPS.Search static method), 365
 search_next() (GPS.File method), 319
 Search_Result (class in GPS), 366
 select window on match, 98
 select() (GPS.EditorBuffer method), 300
 select_all() (GPS.Console method), 277
 select_all() (GPS.Editor static method), 293
 select_text() (GPS.Editor static method), 293
 selection_end() (GPS.EditorBuffer method), 300
 selection_start() (GPS.EditorBuffer method), 300
 send() (GPS.Debuggger method), 283
 send() (GPS.Process method), 352
 separate unit, 67
 server, 204, 240
 sessions() (GPS.Codefix static method), 268
 set() (GPS.Preference method), 349
 set_action() (GPS.Message method), 346
 set_active() (GPS.Logger method), 339
 set_active() (GPS.Menu method), 345
 set_attribute_as_string() (GPS.Project method), 361
 set_background() (GPS.CommandWindow method), 272
 set_background() (GPS.Style method), 367
 set_background_color() (GPS.Editor static method), 293
 set_build_mode() (in module GPS), 257
 set_busy() (in module GPS), 257
 set_closed() (GPS.Activities method), 261
 set_cmd_line() (GPS.SwitchesChooser method), 367
 set_extend_selection() (GPS.EditorView method), 309
 set_foreground() (GPS.Style method), 367
 set_in_speedbar() (GPS.Style method), 367
 set_last_command() (in module GPS), 258
 set_multi_cursors_auto_sync() (GPS.EditorBuffer method), 300
 set_multi_cursors_manual_sync() (GPS.EditorBuffer method), 301
 set_pattern() (GPS.Search method), 365
 set_prompt() (GPS.CommandWindow method), 272
 set_property() (GPS.EditorOverlay method), 307
 set_property() (GPS.File method), 319
 set_property() (GPS.Project method), 361
 set_read_only() (GPS.EditorBuffer method), 301
 set_read_only() (GPS.EditorView method), 309
 set_reference() (GPS.VCS static method), 372
 set_scenario_variable() (GPS.Project static method), 361
 set_sensitive() (GPS.Contextual method), 281
 set_sensitive() (GPS.GUI method), 322
 set_size() (GPS.Process method), 352
 set_sort_order_hint() (GPS.Locations static method), 338
 set_sort_order_hint() (GPS.Message static method), 346
 set_style() (GPS.Message method), 346
 set_style() (gps_utils.highlighter.Background_Highlighter method), 380
 set_subprogram() (GPS.Message method), 347
 set_synchronized_scrolling() (GPS.Editor static method), 293
 set_text() (GPS.Button method), 265
 set_text() (GPS.Combo method), 270
 set_title() (GPS.Editor static method), 293
 set_writable() (GPS.Editor static method), 294
 shell, 132, 172
 short (GPS.Search_Result attribute), 366
 show elaboration cycles, 163
 Show hidden directories, 18
 show() (GPS.BuildTarget method), 264
 show() (GPS.Contextual method), 281
 show() (GPS.Entity method), 314
 show() (GPS.GUI method), 322
 show() (GPS.MDI static method), 342
 show() (GPS.Search_Result method), 366
 show_analysis_report() (GPS.CodeAnalysis method), 267
 show_coverage_information() (GPS.CodeAnalysis method), 267
 smart completion, 157
 solving problems, 246
 source file, 50
 source navigation, 67
 source_dirs() (GPS.Project method), 362
 SOURCES (GPS.Search attribute), 365
 sources() (GPS.Project method), 362
 spawn() (GPS.Debuggger static method), 283
 speed column policy, 156
 splash screen, 154
 split() (GPS.MDIWindow method), 343
 Stack Analysis, 141
 stack analysis, 132
 start() (gps_utils.highlighter.On_The_Fly_Highlighter method), 381
 start_highlight() (gps_utils.highlighter.Background_Highlighter method), 380
 start_line() (GPS.AreaContext method), 262
 start_undo_group() (GPS.EditorBuffer method), 301
 starts_word() (GPS.EditorLocation method), 306
 status bar, 154
 status() (GPS.Task method), 368
 status_parse() (GPS.VCS static method), 373
 stock_icons, 201
 stop() (gps_utils.highlighter.On_The_Fly_Highlighter method), 381
 stop_highlight() (gps_utils.highlighter.Background_Highlighter method), 380
 strip blanks, 156
 style, 164
 Style (class in GPS), 366

- submitting bugs, 246
- subprogram parameters, 222
- subprogram_name() (GPS.Editor static method), 294
- subprogram_name() (GPS.EditorLocation method), 306
- substitution, 174
- SUBSTRINGS (GPS.Search attribute), 365
- Subversion, 121
- Subversion Windows, 121
- suggestions, 246
- supported_languages() (in module GPS), 258
- supported_systems() (GPS.VCS static method), 373
- SwitchesChooser (class in GPS), 367
- syntax highlighting, 116

T

- tabs, 55
- tabulation, 157, 160
- Target, 205
- target, 106
- targets, 245
- Task (class in GPS), 368
- tasks, 34
- Text_Highlighter (class in gps_utils.highlighter), 382
- thaw_prefs() (in module GPS), 258
- thaw_xref() (in module GPS), 258
- themes, 165
- themes creation, 186
- Timeout (class in GPS), 368
- tip of the day, 7, 155
- title() (GPS.EditorView method), 310
- toggle_group_commit() (GPS.Activities method), 261
- tool bar, 9, 154, 182
 - progress bar, 9
- Toolbar (class in GPS), 369
- ToolButton (class in GPS), 369
- tools, 131
- tooltip, 48, 116, 156
- tooltip timeout, 156
- tty, 161
- type, 230
- type hierarchy, 70
- type() (GPS.Entity method), 315

U

- undo() (GPS.Editor static method), 294
- undo() (GPS.EditorBuffer method), 301
- Unexpected_Exception (class in GPS), 371
- unhighlight() (GPS.Editor static method), 294
- unhighlight_range() (GPS.Editor static method), 294
- Unix, 247
- unselect() (GPS.EditorBuffer method), 301
- unset_busy() (in module GPS), 258
- update() (GPS.VCS static method), 373
- update_parse() (GPS.VCS static method), 373

- update_xref() (GPS.Project method), 362
- use_messages() (gps_utils.highlighter.OverlayStyle method), 381
- used_by() (GPS.File method), 319
- uses() (GPS.File method), 319

V

- VCS, 84, 121
- VCS (class in GPS), 371
- VCS activities, 124
- VCS explorer, 122
- vcs() (GPS.Activities method), 261
- Vdiff (class in GPS), 373
- version control, 121, 122, 124, 125
- Version Control System, 84
- version() (in module GPS), 258
- vertical layout, 163
- vi, 33, 64
- views() (GPS.EditorBuffer method), 301
- visible (GPS.Task attribute), 368
- visual diff, 132, 133
- VxWorks, 75
- VxWorks AE, 114

W

- wait() (GPS.Process method), 353
- warning index, 164
- welcome dialog, 6, 154
- whole word, 97
- WHOLE_WORD (GPS.Search attribute), 365
- Windows, 40, 244, 247
- windows, 23, 25
 - bookmarks, 32
 - call trees, 29
 - callgraph browser, 29
 - dependency browser, 36
 - elaboration circularities, 37
 - entity browser, 38
 - execution window, 34
 - files view, 22
 - filter, 3
 - local settings menu, 3
 - local toolbar, 3
 - locations, 14
 - main, 2
 - menu bar, 8
 - messages, 13
 - os shell, 33
 - project browser, 35
 - project view, 15
 - python console, 33
 - scenario view, 19
 - shell console, 33
 - task manager, 34

- tip of the day, 7
- welcome dialog, 6
- workspace, 3
- windows view, 23
- with_save_current_window() (in module gps_utils), 379
- with_save_excursion() (in module gps_utils), 379
- wrench icon, 134
- write() (GPS.CommandWindow method), 272
- write() (GPS.Console method), 277
- write_with_links() (GPS.Console method), 277

X

- XMLViewer (class in GPS), 375
- xpm, 182
- xref_db() (in module GPS), 258
- xref_frozen() (in module GPS), 259

Y

- yank, 52, 65
- yes_no_dialog, 214
- yes_no_dialog() (GPS.MDI static method), 342