GPS User's Guide

Release 2017

AdaCore

May 16, 2017

GPS is a complete integrated development environment. It integrates with a wide range of tools, providing easy access to each. It integrates especially well with AdaCore's tools but can easily be extended to work with other tools by writing small plugins in Python.

Here is a summary of the features of the GNAT Programming Studio:

- *Multiple Document Interface*

  GPS uses a multiple document interface, allowing you to organize windows the way you want and organize your desktop by floating them to other screens or dragging them to any location. (GPS restores the desktop the next time it 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, and alias expansion, among other features.

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

  Any compiler called by a command line can be integrated in GPS, with built in support for GNAT, **`gcc`**, and **`make`**. You can easily navigate through error messages, and automatic code fixing is provided for many common errors. GPS includes support for cross-compilers (running compilers on a different machine than the one on which GPS is running).

- Project management (*Project Handling*)

  You can use project files (editable either graphically or manually) to describe attributes of a project, including the location of sources, their naming schemes, and how they should be built. GPS provides a graphical browser to analyze both dependencies between your projects and between sources within your projects.

- Integration with various *Version Control System*

  CVS, subversion, **`git`**, and ClearCase are supported out of the box. You can add support for others by customizing some XML plugins.

- Intelligent *Source Navigation*

  By leveraging information provided by the compilers and using its own parsers, GPS allows you to find program information such as the declaration of entities and their references, that would otherwise be hard to locate. It also provides advanced capabilities such as call graphs and UML-like entity browsers.

- Full debugger integration (*Debugging*)

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

- Integration with code analysis tools (*Tools*)

  GPS integrates tightly with various command-line tools such as **gcov** and GNATcoverage (for the coverage of your code) and CodePeer and Spark (to analyze your code). In most cases, it provides graphical rendering of their output, often integrated with the editor itself so the information is available where and when you need it.

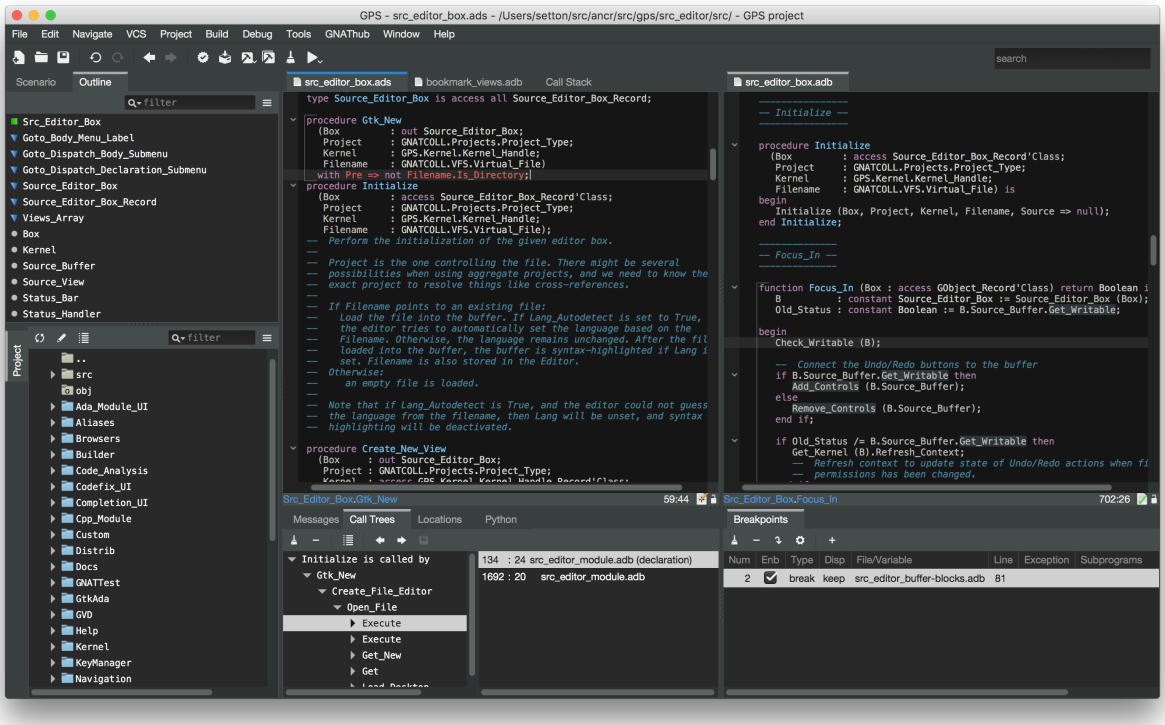
- Fully customizable (*Customizing and Extending GPS*)

  GPS provides an extensive Python API, allowing you to customize existing features or easily develop your own new plugins. Simpler customization can be done through the numerous preferences and local settings.

CHAPTER

ONE

DESCRIPTION OF THE MAIN WINDOW



The GNAT Programming Studio has one main window, which is where you perform most of your work. However, GPS is very flexible and lets you organize your desktop many different ways, as discussed in a later section (*Multiple Document Interface*).

There are also other windows that might pop up at various times, documented in this section.

1.1 The Workspace

The overall workspace is based on a multiple document interface (see *Multiple Document Interface*) and can contain any number of windows, the most important of which are usually the editors. However, GPS also provides a large number of views that you can add to the workspace. The sections below list them.

1.1.1 Common features of the views

Some views are part of the default desktop and are visible by default. Open the other views through one of the submenus of the *Tools* menu, most often *Tools* → *Views*.

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

There is often a button to the right of these local toolbars that opens a local settings menu. This menu either contains more actions you can perform in that view or various configuration settings allowing you to change the behavior or display of the view.

Some views also have a filter in their local toolbar. You can use these filters to reduce the amount of information displayed on the screen by only displaying those lines matching the filter.

If you click on the left icon of the filter, GPS brings up a popup menu to allow you to configure the filter:

- Use the first three entries to chose the search algorithm (full text match, regular expression, or fuzzy matching). These modes are similar to the ones used in the omni-search (see *The omni-search*).
- The next entry is *Invert filter*. When you select this option, lines that do not match the filter are displayed, instead of the default behavior of displaying ones that match the filter. You can also enable this mode temporarily by beginning the filter with the string `not:`. For example, a filter in the *Locations* view saying `not:warning` hides all warning messages.
- Select the last entry, *Whole word*, when you only want to match full words, not substrings.

1.1.2 Common features of browsers

GPS presents a view of information using an interactive display called a "browser", which shows a canvas containing boxes you can manipulate. Browsers provide the following additional capabilities:

- Links

  Boxes can be linked together and remain linked when they are moved. The are different types of links; see the description of the various browsers for more details.

  Hide links using a button on the local toolbar. This keeps the canvas more readable at the cost of losing information. You can also hide only a subset of links. Even when links are hidden, if you select a box, boxes linked to it are still highlighted.

- Scrolling

  When many boxes are displayed, the currently visible area may be too small for all of them. When that happens, GPS adds scrollbars. You can also scroll using the arrow keys, or by dragging the background while pressing the left mouse button.

- Layout

  GPS organizes the boxes in a browser using a simple layout algorithm, which is layer oriented: items with no parents are put in the first layer, 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. If you move boxes, this algorithm tries to preserve their relative positions as much as possible.

  Use the *Refresh layout* button in the local toolbar to recompute the layout at any time, including that of boxes you moved.

- Moving boxes

  Move boxes with the mouse. Drag the box by clicking on its title bar. The box's links are still displayed during the move, so you can see whether it overlaps any other box. If you try to move the box outside the visible part of the browser, it is scrolled.

- Selecting boxes

  Select a box by clicking it.

  The title bar of selected boxes is a different color. All boxes linked to them also use a different title bar color and so do the links. This is the most convenient way to visualize the relationships between boxes when many are present in the browser.

  Use buttons in the local toolbar to either remove the selected boxes or remove the boxes that are not selected.

- Zooming

  GPS provides several different zoom levels. Use the *zoom in*, *zoom out*, and *zoom* buttons in the local toolbar to change the level and use the latter to explicitly select the level you want.

  You can also press the `alt` key and use the mouse wheel to zoom in or out.

  This capability is generally useful when many boxes are displayed in the browser to allow you to get an overview of the layout and the relationships between the boxes.

- export

  Export the entire contents of a browser as a `PNG` or `SVG` image using the *Export to...* button in the local toolbar.

- Hyper-links

  Some boxes contain hyper links, displayed in blue by default, and underlined. Clicking on these generally displays new boxes.

- Contextual menus

  Right-clicking on boxes displays a contextual menu with actions you can perform on that box. These actions are specific to the kind of box you clicked.

- Grid

  By default, GPS doesn't display a grid on the canvas. Use the local settings menu to show the grid (uncheck *Draw grid*) or to force items to align on the grid (*Align on grid*).

Icons for source language entities

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

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

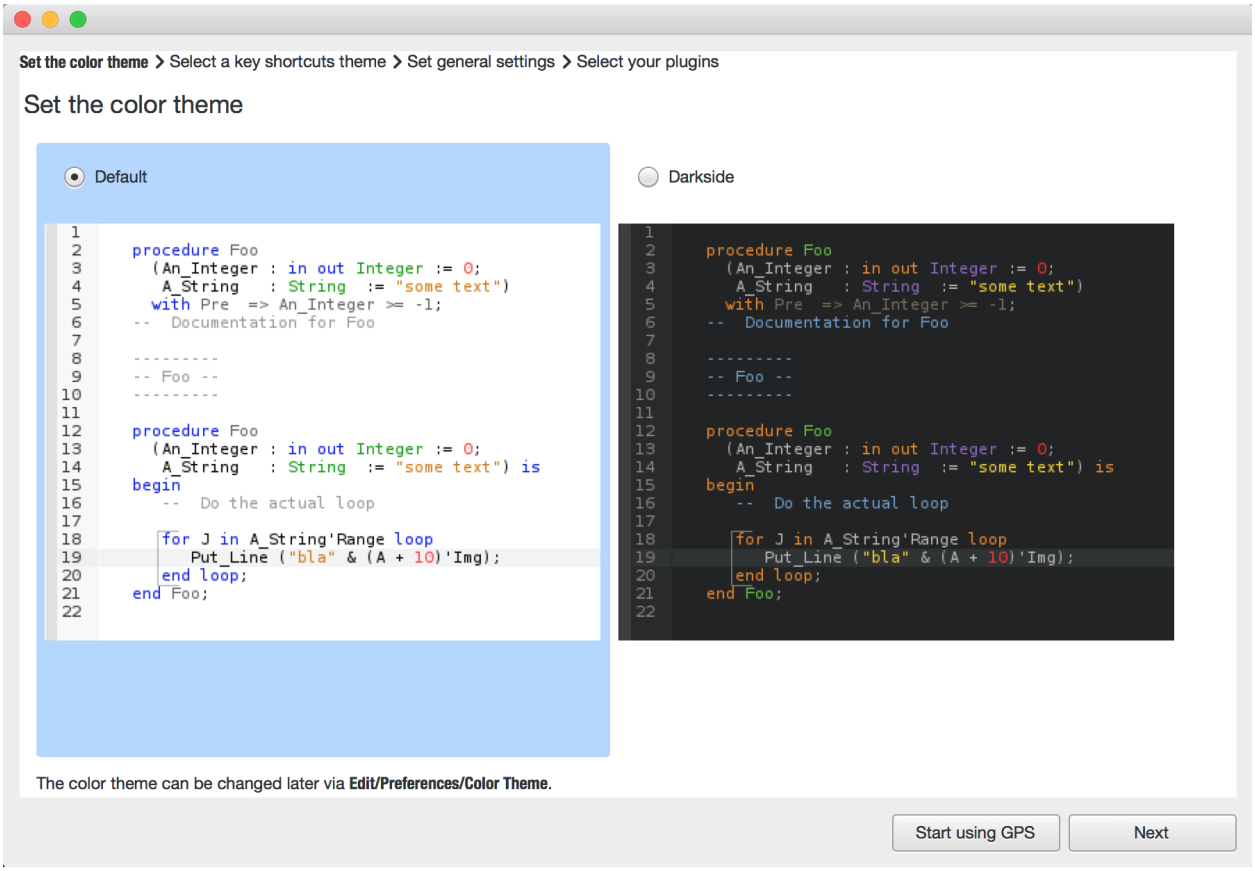
Icons for 'protected' and 'private' entities appear within an enclosing box indicating a compile-time visibility constraint. For entities with 'protected' visibility, the enclosing box is gray. 'Private' entities are enclosed by a red box. Icons for 'public' entities have no enclosing box. For example, a variable with 'private' visibility is represented by an icon consisting of a dot enclosed by a red box. These additional decorations 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 by a red box.

Language constructs are mapped to categories in a language-specific manner. For example, C++ namespaces and Ada packages correspond to the *package* category and C functions and Ada subprograms correspond to the *method* category. The *generic* category is a general category representing other language entities, but not all possible language constructs are mapped to categories and icons. (Specifically, the *generic* category does not correspond to Ada generic units or C++ templates.)

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

Icons for a given entity may appear more than once within a view. For example, an Ada private type has both a partial view in the visible part of the enclosing package and a full view in the private part of the package. A triangle icon will appear for each of the two occurrences of the type name, one with the additional decoration indicating 'private' visibility.

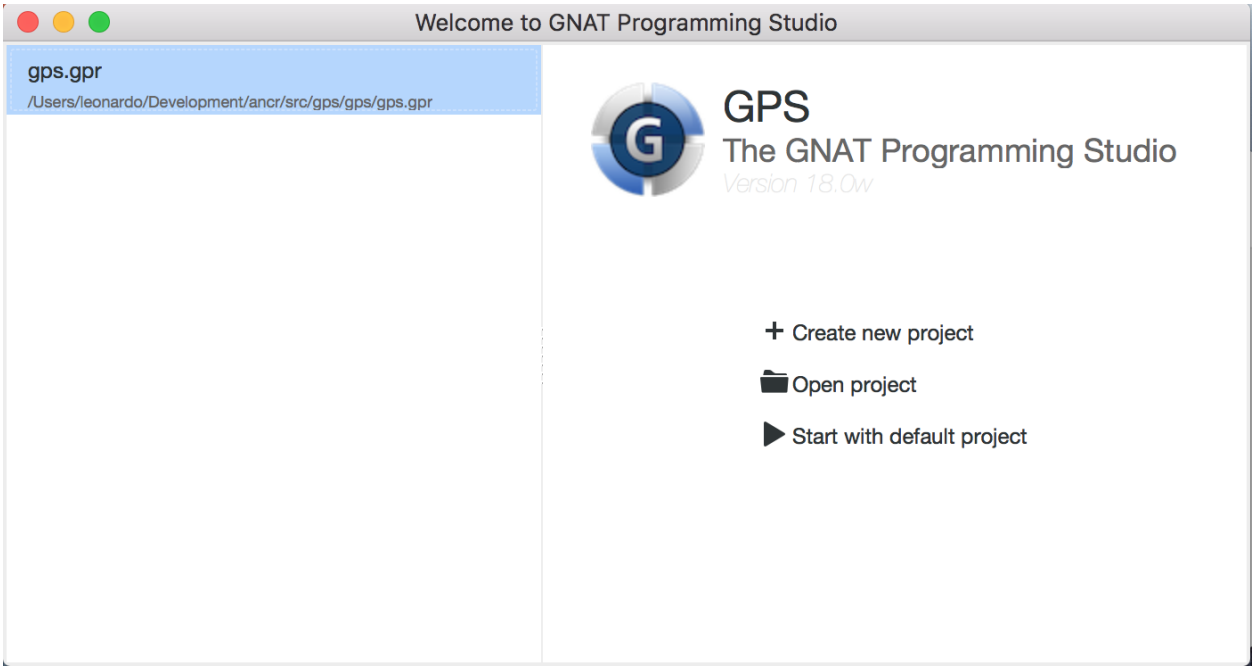
1.2 The Preferences Assistant



When starting GPS for the first time, a preferences assistant window opens, allowing you to configure some general preferences (color theme, key bindings etc.).

You can skip the remaining pages of the preferences assistant by clicking on the *Start using GPS* button or by clicking on red cross.

1.3 The Welcome Dialog



When GPS starts, it looks for a project file to load so it knows where to find the sources of your project. This project is often specified on the command line (via a **-P** switch). If not, and the current directory contains only one project file, GPS selects it automatically. Finally, if you specify the name of a source file to edit, GPS loads a default project. If GPS cannot find a project file, it displays a welcome dialog, giving you the following choices:

- *Create new project*

  Clicking on this button launches 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 example.

- *Open project*

  Clicking on this button opens up a file browser, allowing you to select a a project file to load.

- *Start with default*

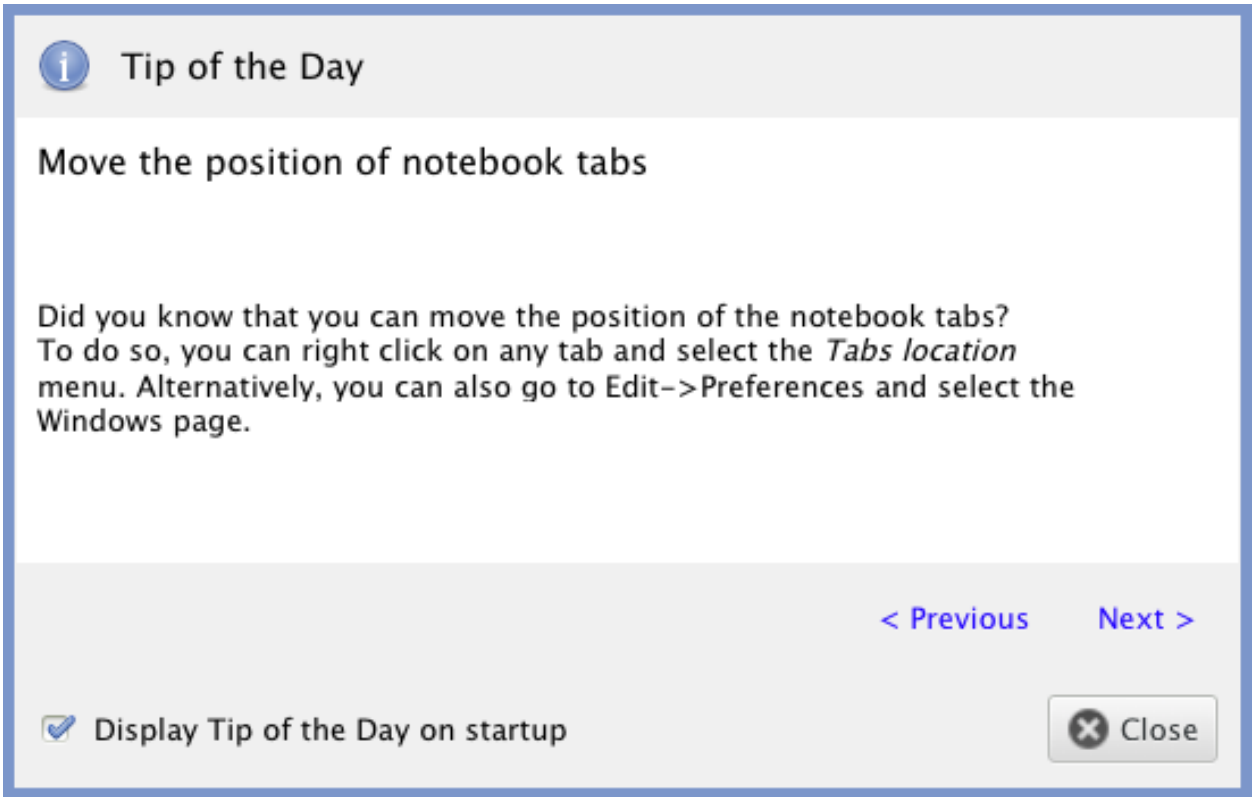
  Clicking on this button causes GPS to look for a project called `default.gpr` in the current directory and load it if found. Otherwise, it copies the default project `<prefix>/share/gps/default.gpr` into the current directory and loads it. GPS removes this temporary copy when exiting or loading another project if you have not modified the copy.

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

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

In addition to these choices, you can also load a recently opened project by clicking the project of interest in the left-hand pane listing the known recent projects.

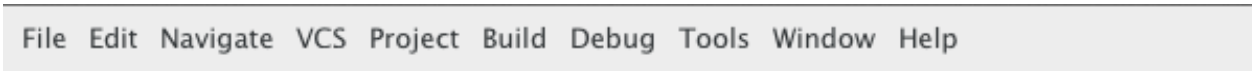
1.4 The Tip of the Day



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

Disable this dialog by unchecking the *Display Tip of the Day on startup* check box. To reenable this dialog, go to the *Edit* → *Preferences* menu.

1.5 The Menu Bar



GPS provides a standard menu bar giving access to all operations. However, it is usually easier to access a feature using the various contextual menus provided throughout GPS: these give direct access to the most relevant actions in the current context (for example, a project, directory, file, or entity). Contextual menus pop up when you click the right mouse button or use the special `open contextual menu` key on most keyboards.

You can access the following entries from the menu bar:

- *File* (see *The File Menu*)
- *Edit* (see *The Edit Menu*)
- *Navigate* (see *The Navigate Menu*)
- *Project* (see *The Project Menu*)
- *Build* (see *The Build Menu*)

- *Debug* (see *The Debug Menu*)
- *Tools* (see *The Tools Menu*)
- *SPARK*

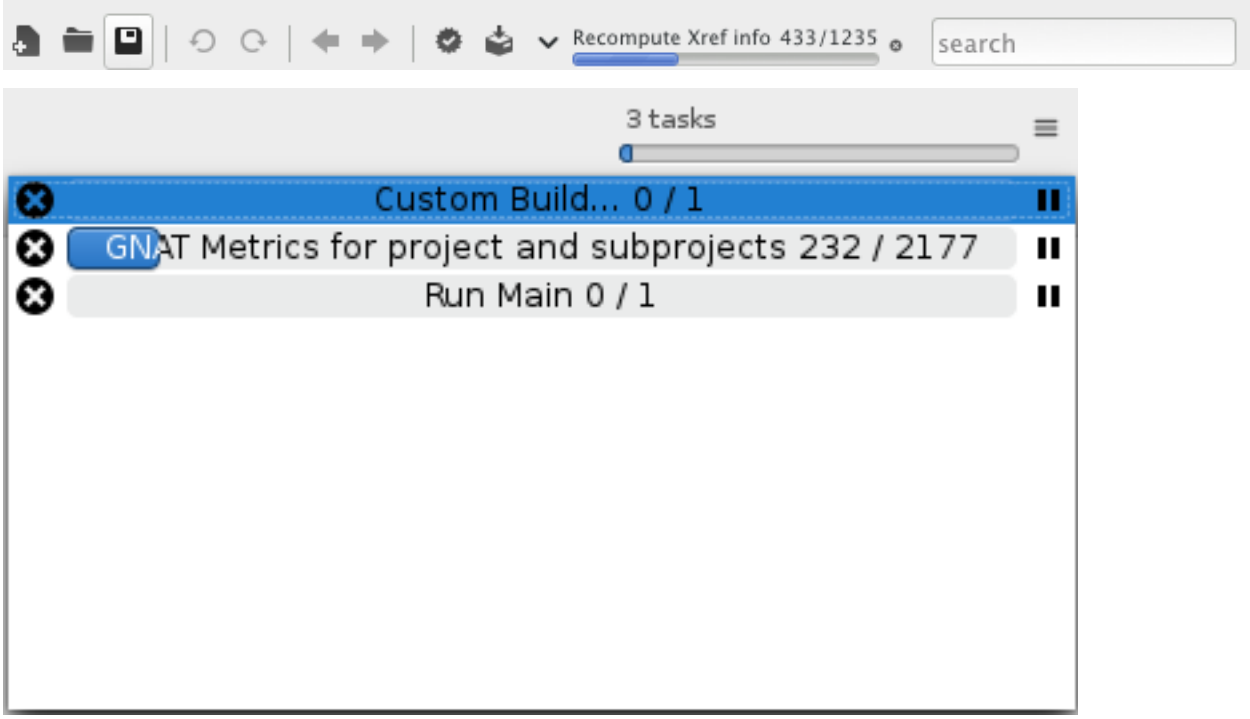
  This menu is available if the SPARK toolset is installed on your system and available on your PATH. See *Help → SPARK → Reference → Using SPARK with GPS* for more details.

- *CodePeer*

  This menu is available if the CodePeer toolset is installed on your system and available on your PATH. See your CodePeer documentation for more details.

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

1.6 The Tool Bar

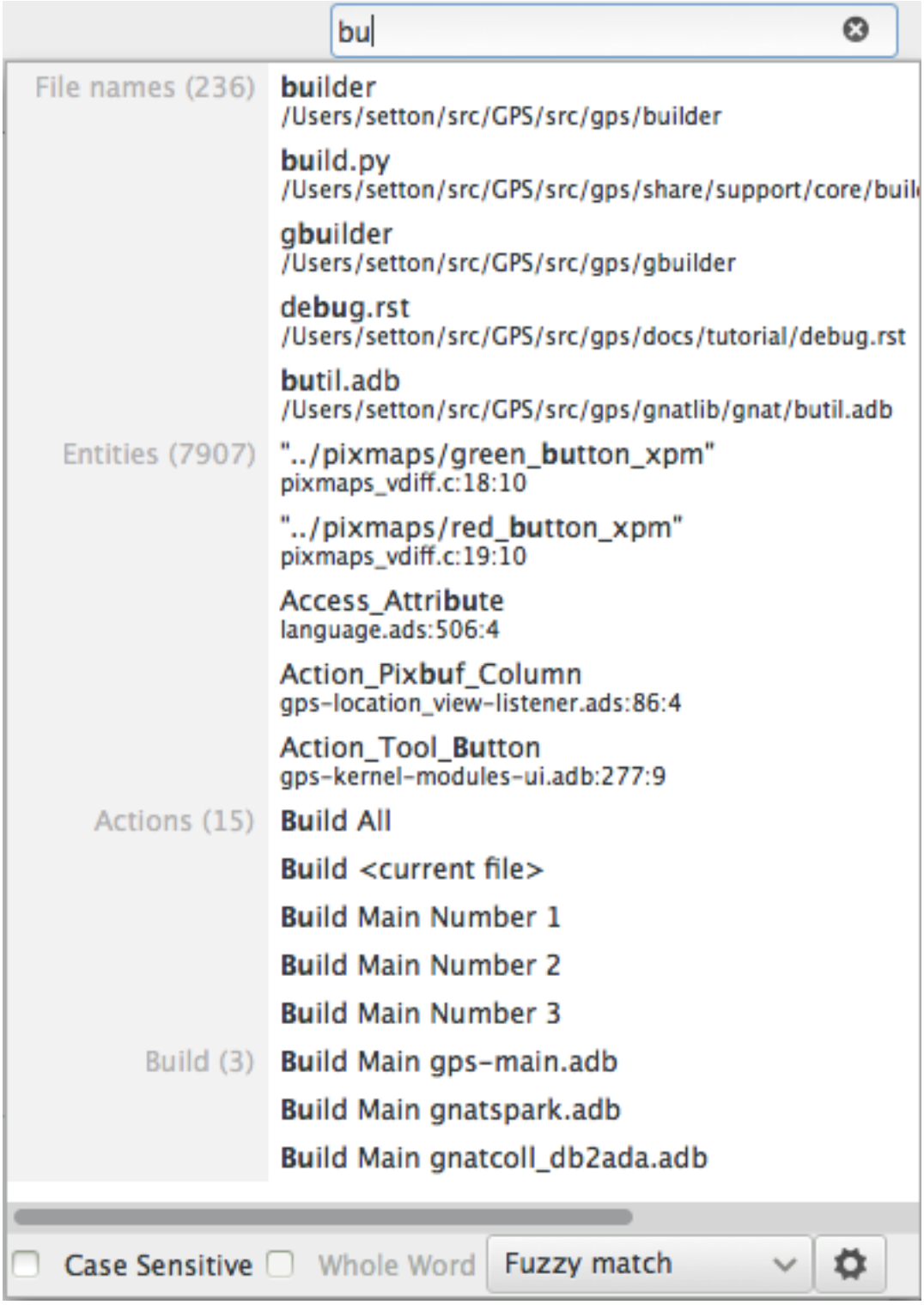


The tool bar provides shortcuts to some common actions:

- Create a new file
- Open an existing file (see also the omni-search on the right of the bar)
- Save the current file
- Undo or redo last editing
- Go to previous or next saved location
- Multiple customizable buttons to build, clean, run or debug your project
- multiple buttons to stop and continue the debugger, step to the next instruction, and other similar actions when a debugger is running.

When GPS is performing background actions, such as loading cross-reference information or all actions involving external processes (including compiling), it displays a progress bar in the toolbar showing when the current task(s) will be completed. Click on the button to pop up a window showing the details of the tasks. This window is a Tasks view, and can be used to pause or interrupt running tasks (see *The Tasks view*). This window can be discarded by pressing *ESC* or by clicking anywhere else in the GPS. This window also disappears when there are no more running tasks.

1.7 The omni-search



The final item in the toolbar is "omni-search". Use this to search for text in various contexts in GPS, such as filenames (for convenient access to source files), the entities referenced in your application, and your code.

There are various ways to use the omni-search:

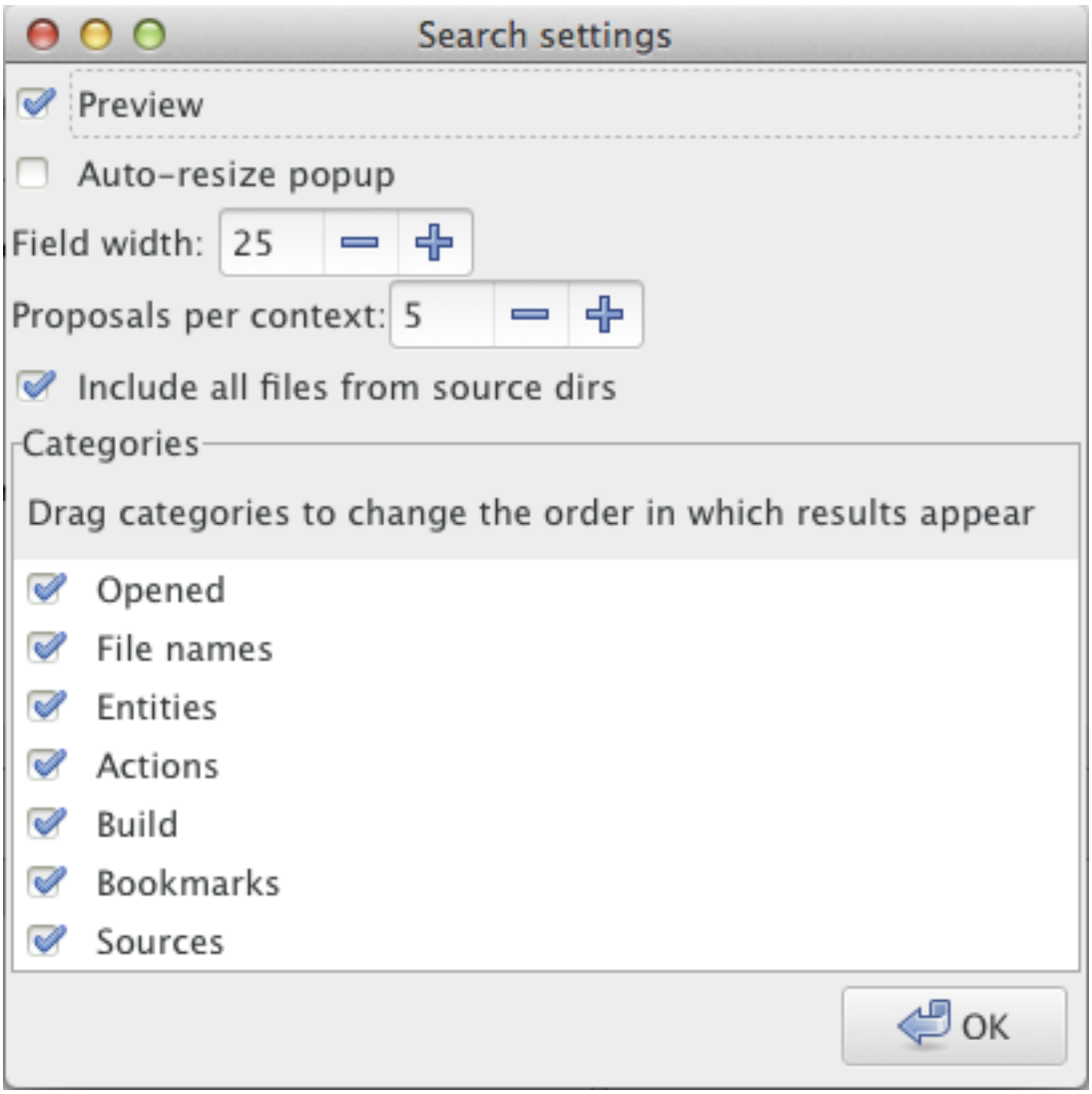
- The simplest way is to click on it and type the pattern you want to find. GPS immediately starts searching in the background for possible matching open windows, file names, entities, GPS actions, bookmarks, and source files. For each context, GPS display how many matches there are in that context but only displays the five matches with the highest score.

  Click on the name of context to search only in that context. For example, if GPS shows 20 file names matching your search (while only displaying the five first), click on *file names* to view all 20 names and exclude the results from all the other contexts. If you click on the context again, GPS again displays the results from all contexts.

- If you are searching in a single context, GPS defines a number of actions to which you can bind key shortcuts via the *Edit* → *Key Shortcuts* dialog instead of using the above procedure. These actions are found in the *Search* category and are called *Global Search in context:*. GPS includes a menu for two of them by default: *File* → *Open From Project...* searches filenames, while *Navigate* → *Goto Entity...* searches all entities defined in your project.

Each context displays its results slightly differently and clicking on a result has different effects in each context. For example, clicking on a file name opens the corresponding file, while clicking on an entity jumps to its declaration and clicking on a bookmark displays the source file containing the bookmark.

Press `enter` at any point to select the top item in the list of search results.



You may have no interest in some search contexts. Disable them by clicking the *Settings* icon at the bottom-right corner of the completion popup. The resulting dialog displays a list of all contexts to be searched; clicking on any of the checkboxes next to the names disables searching that context. This list is only displayed when you started the omni-search by clicking on it in the toolbar. If you started it via `shift-F3` or the equivalent *File* → *Open From Project...* menu, only a subset of the settings are displayed.

You can also reorder the contexts from this settings dialog, which affects the order in which they are searched and displayed. We recommend keeping the *Sources* context last, because it is the slowest and while GPS is searching it, cannot search the other, faster, contexts.

In the settings dialog, you can chose 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. It displays the corresponding source file or the details for the matching GPS action or bookmark. You can also select the number of results to 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 search context looks for file names and is convenient for quickly opening files. By default, it looks 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 example 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 be annoying if the source directories include too many irrelevant files. Use the *Include all files from source dirs* setting to control this behavior.

GPS allows you to chose among various search algorithms:

- *Full Text* checks whether the text you typed appears exactly as you specified it within the context (for example, a file name, the contents of a file, or the name of an entity).
- *Regular Expression* assumes the text you typed is a valid regular expression and searches for it. If it is not a valid regexp, it tries to search for the exact text (like *Full Text*).
- *Fuzzy Match* tries to find each of the characters you typed, in that order, but possibly with extra characters in between. This is often the fastest way to search, but might requires a bit of getting used to. For example, the text `mypks` matches the file name `MY_PacKage.adS` because the letters shown in upper cases are contained in the filename.

  When searching within source files, the algorithm is changed slightly, to avoid having too many matches. In that context, GPS only allows a close approximations between the text you typed and the text it tries to match (for example, one or two extra or missing characters).

Select the algorithm to use at the bottom of the popup window containing the search results.

Once it finds a match, GPS assigns it a score, used to order the results in the most meaningful way for you. Scoring is based on a number of criteria:

- length of the match

  For example, when searching file names, it is more likely that typing 'foo' was intended to match 'foo.ads' than 'the_long_foo.ads'.

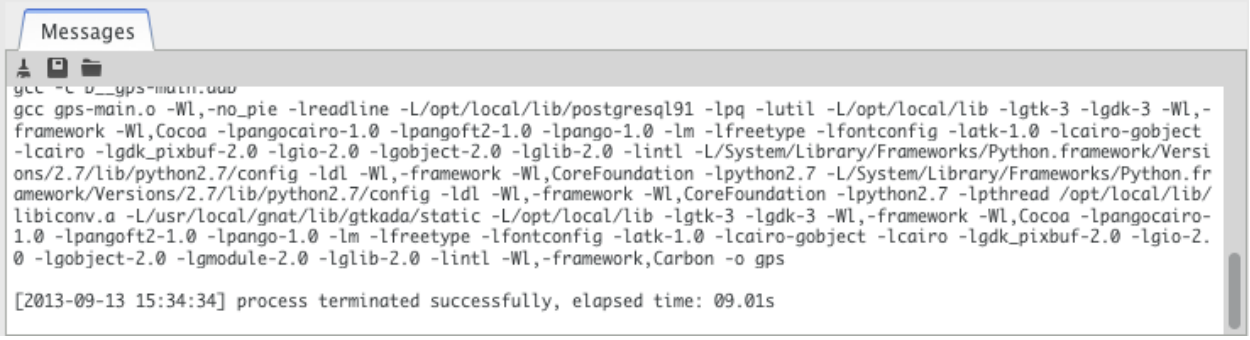
- the grouping of characters in the match

  As we have seen, when doing a fuzzy match GPS allows extra characters between the ones you typed. But the closer the ones you typed are in the match result, the more likely it is that 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 raises its score.

1.8 The *Messages* view



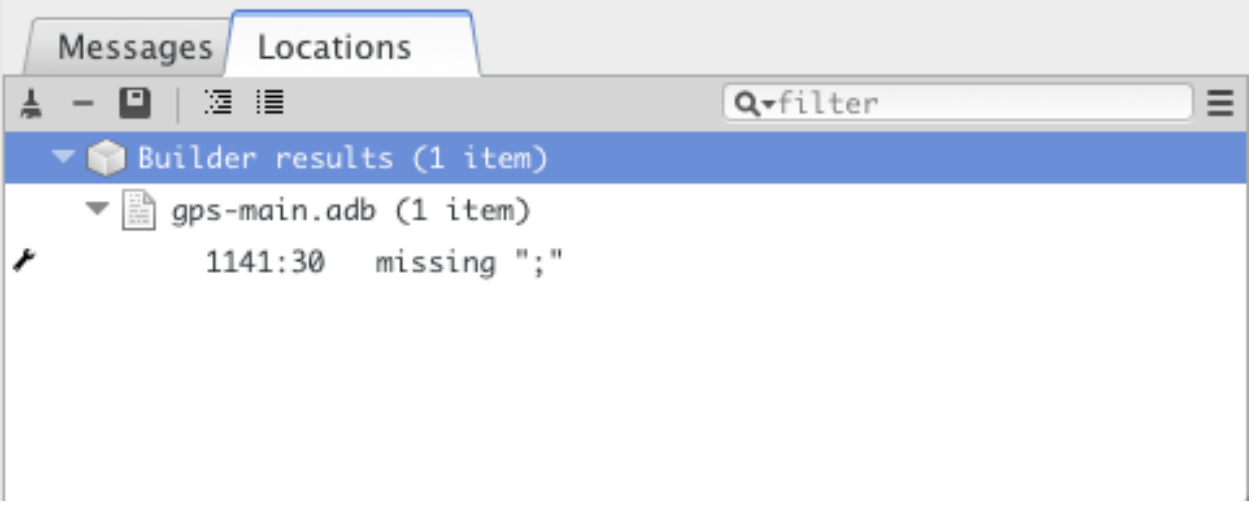
The *Messages* view, which is readonly, displays information and feedback about operations, including build output, information about processes launched, and error messages.

Its local toolbar contains buttons to *Clear* the contents of the window, as well as *Save* and *Load* from files. The latter operation also parses those messages into the *Locations* window.

The actual output of the compilation is displayed in the *Messages* view but is also parsed and many of its messages are displayed more conveniently in the *Locations* view (se *The Locations View*). When a compilation finishes, GPS displays the total elapsed time.

You cannot close the *Messages* view because it might contain important messages. If GPS closed it, you can reopen it with the *Tools* → *Views* → *Messages* menu.

1.9 The *Locations* View



GPS uses the *Location* view, which is also readonly, to display a list of locations in source files (for example, when performing a global search or displaying compilation results).

It displays a hierarchy of categories, each of which contain files, each, in turn, containing messages at specific locations. The category describes the type of messages (for example, search or build results). If the full text of a message is too large to be completely shown in the window, placing the mouse over it pops up a tooltip window with the full text.

Each message in this window corresponds to a line in a source editor. This line has been highlighted and has a mark on its left side. Clicking on a message brings up an editor pointing to that line.

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

- *Clear* removes all entries from the view and, depending on your settings, may also close the view.
- *Remove* removes the currently selected category, file or message as well as the corresponding highlighting in the source editor.
- *Save* saves the contents of the view to a text file for later reference. You cannot load this file back into the *Locations* view, but you can load it into the *Messages* view. However, if you plan to reload it later, it is better to save and reload the contents of the *Messages* view instead.
- *Expand All* and *Collapse All* shows or hides all messages in the view.
- a filter to selectively show or hide some messages. Filtering is done on the text of the message itself (the filter is either text or a regular expression). You can also reverse the filter. For example, typing *warning* in the filter field and reversing the filter hides warning messages

The local settings menu contains the following entries:

- *Sort by subcategory*

  Toggle the sorting of messages by sub-categories. This is useful for separating warnings from errors in build results. The error messages appear first. The default is to sort the message by their location.

- *Sort files alphabetically*

  Sort messages by filenames (sorted alphabetically). The default does not sort by filenames to make it easier to manipulate *Locations* view while the compilation is proceeding. (If sorted, the messages might be reordered while you are trying to click on them).

- *Jump to first location*

  Every time a new category is created, for example, as a result of a compilation or search operation, the first message in that category is automatically selected and the corresponding editor opened, and the focus is given to the Locations view.

- *Warp around on next/previous*

  Controls the behavior of the *Previous tag* and *Next tag* menus (see below).

- *Auto close locations*

  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. Be careful, because the loaded contents might not apply the next time. For example, the source files have changed, or build errors have been fixed. So you should not select this option if those conditions might apply.

- *Preserve messages*

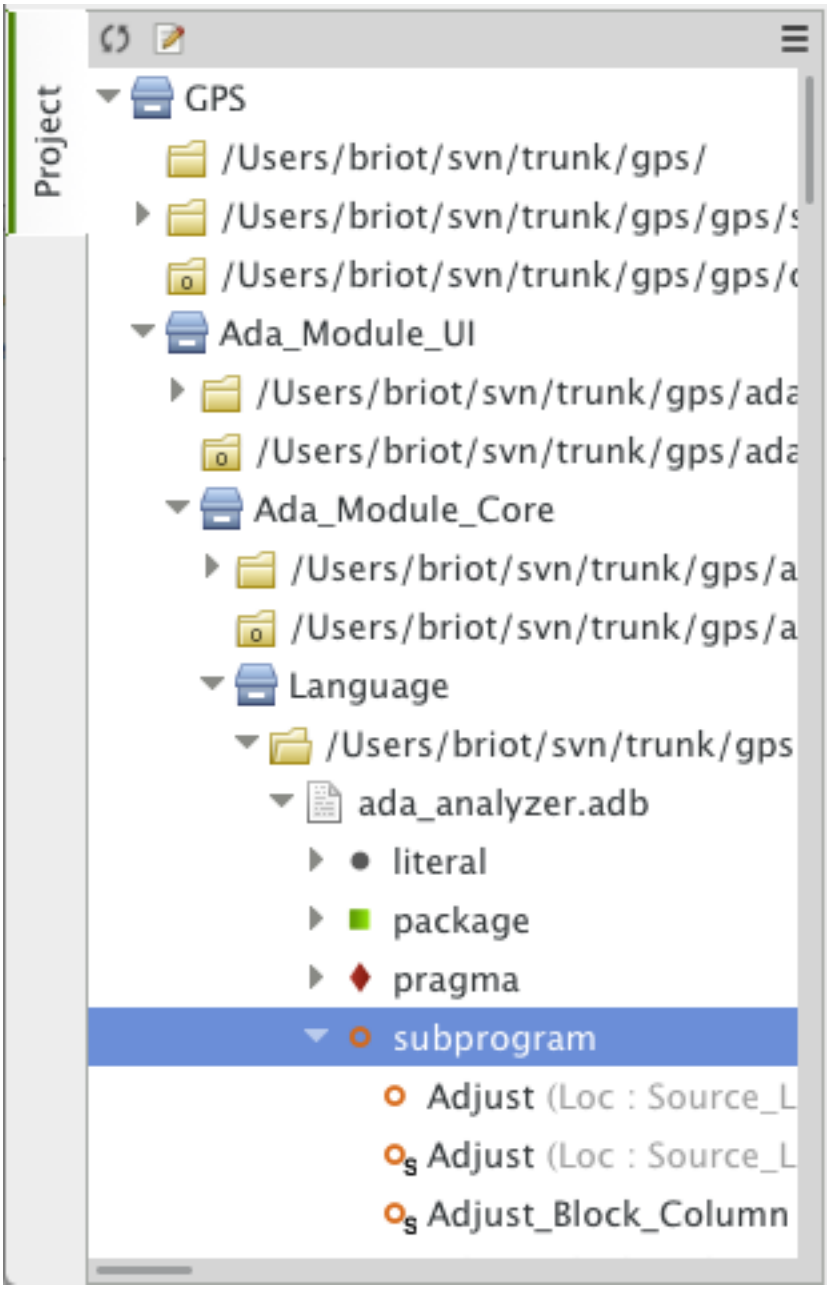
  Preserve more build errors after recompiling. When the Locations view contains build errors, and one of the files is being recompiled, the Locations view will now only update the entries for that file, rather than removing all build errors.

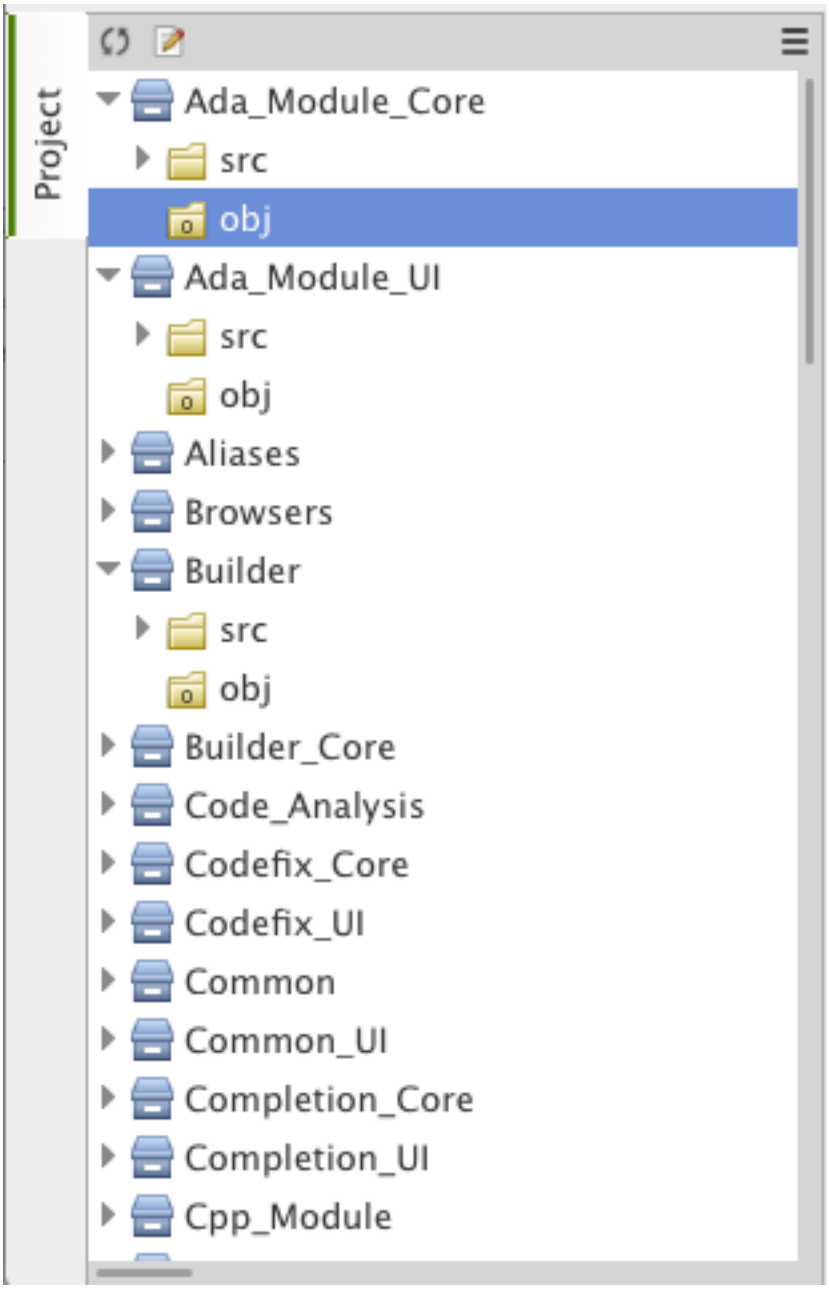
GPS provides two menus to navigate through the locations using the keyboard: *Navigate* → *Previous Tag* and *Navigate* → *Next Tag*. Depending on your settings, they might wrap around after reaching the first or last message.

You can also 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.

1.10 The *Project* view





The project view displays a representation of the various components of your project. By default, it is displayed on the left side of the workspace. Select it using the *Project* → *Project View* or *Tools* → *Views* → *Project* menus.

On Windows, you can drop files (for example, from Windows Explorer) into the project view. If you drop a project file, GPS loads it and it replaces the current project; if you drop a source file, GPS opens it in a new editor.

The project view, combined with the file and outline view, provide an interactive search capability allowing you to quickly search information currently displayed. Start typing the text to search when the view has the focus. Note that the contents of the *Project* view are computed lazily, so not all files are known to this search capability before they have been opened.

This search opens a small window at the bottom of the view where you can interactively type names. The first matching name in the tree is selected when you type it. Use the `up` and `down` keys to navigate through all the items matching the current text.

The various components displayed in the project view are:

projects

Each source file you are working with is part of a project. Projects are a way to record the switches to use for the various tools as well as a number of other properties such as 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* for details on how projects are loaded in GPS).

The *Project* view displays this project hierarchy: the top node is the root project of your application (usually where the source file that contains the main subprogram will be located). A node is displayed for each imported project and recursively for other imported projects. If a project is imported by several projects, it may appear multiple times in the view,

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

An icon with a pen mark is displayed if the project was modified but not saved yet. You can save it at any time by right-clicking the icon. GPS either reminds you to save it before any compilation or saves it automatically, depending on your preference settings.

GPS provides 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. You may find this display useful for deep project hierarchies, where it can make it easier to find projects. Activate this display using the local settings menu to the right of the *Project* view toolbar.

directories

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

You chose whether to see the absolute path names for the directories or paths relative to the location of the project by using the local settings menu *Show absolute paths* of the *Project* view. In all cases, the tooltip displayed when the mouse hovers over a file or directory shows the full path.

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

Use the local setting *Show hidden directories* to select the directories to be considered hidden. Use this to hide version control directories such as `CVS` or `.svn`.

files

Source files are displayed under the node corresponding to the directory containing the file. Only the source files actually belonging to the project (i.e. are written in a language supported by that project and follow its naming scheme) are visible. For more information on supported languages, see *Supported Languages*. A file might appear multiple times in the *Project* view if the project it belongs to is imported by several other projects.

You can drag a file into GPS. This opens a new editor if the file is not already being edited or moves to the existing editor otherwise. If you press `shift` while dragging the file and it is already being edited, GPS creates a new view of the existing editor.

entities

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

Double-clicking on a file or clicking on any entity opens an editor or display showing, respectively, the first line in the file or the line on which the entity is defined.

If you open the search dialog via the *Navigate → Find or Replace...* menu, you can search for anything in the *Project* view, either a file or an entity. Searching for an entity can be slow if you have many files and/or large files.

GPS also provides a contextual menu, called *Locate in Project View*, in source editors. This automatically searches for the first entry in this file in the *Project* view. This contextual menu is also available in other modules, for example when selecting a file in the *Dependency* browser.

The local toolbar of the *Project* view contains a button to reload the project. Use this when you have created or removed source files from other applications and want to let GPS know there might have been changes 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, such as the tool switches or 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 a project node, a contextual menu appears which contains, among others, the following entries that you can use to understand or modify your project:

- *Show projects imported by...*
- *Show projects depending on...*

  Open a new window, the *Project* browser, which displays graphically the relationships between each project in the hierarchy (see *The Project Browser*).

- *Project → Properties*

  Opens a new dialog to interactively edit the attributes of the project (such as tool switches and naming schemes) and is similar to the local toolbar button.

- *Project → Save project...*

  Saves a single project in the hierarchy after you modified it. Modified but unsaved projects in the hierarchy have a special icon (a pen mark on top of the standard icon). If you would rather save all modified projects in a single step, use the menu bar item *Project → Save All*.

  Any time you modify one or more projects, the contents of the project view is automatically refreshed, but 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, a pen mark on top of the standard icon:

- *Project → Edit source file*

  Loads the project file into an editor so you can edit it. Use this if you need to access some features of the project files that are not accessible graphically (such as rename statements and variables).

- *Project → Dependencies*

  Opens the dependencies editor for the selected project (se *The_Project_Dependencies_Editor*).

- *Project → Add scenario variable*

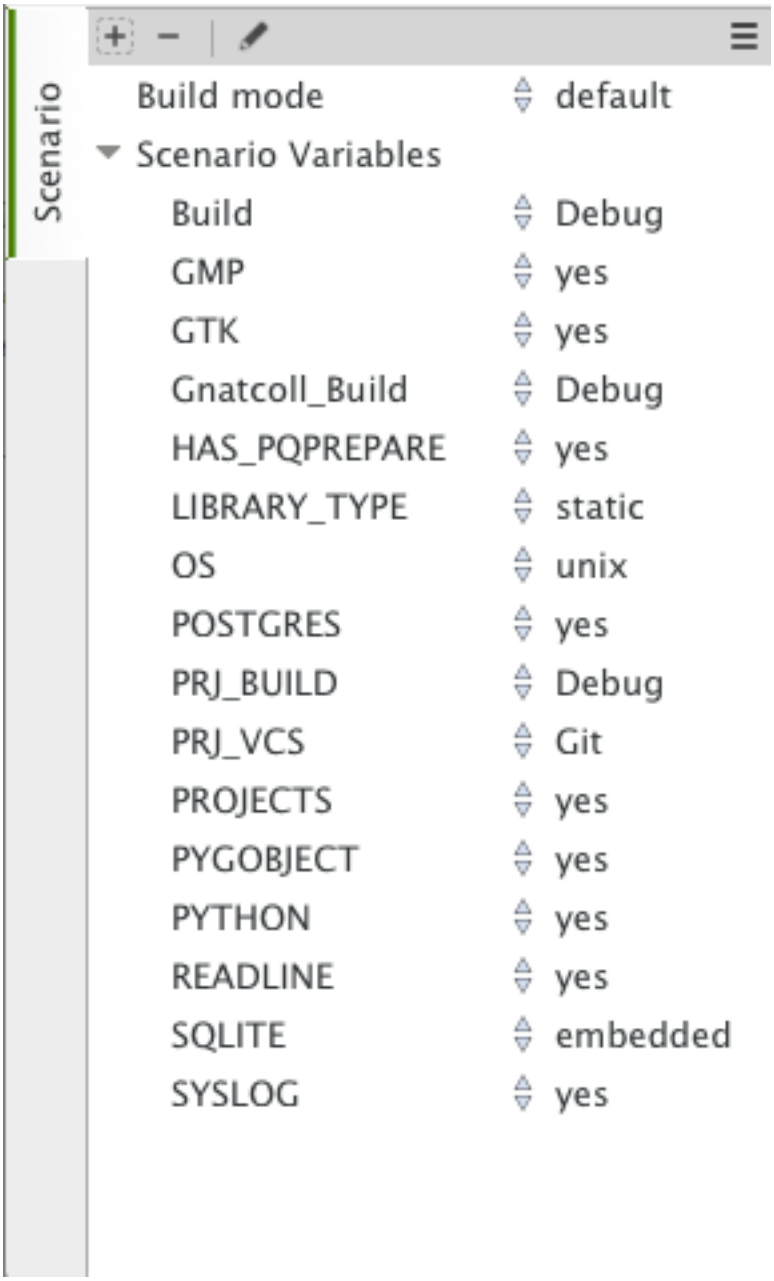
  Adds new scenario variables to the project (see *Scenarios and Configuration Variables*). Howver, you may find it more convenient to use the *Scenario* view for this purpose.

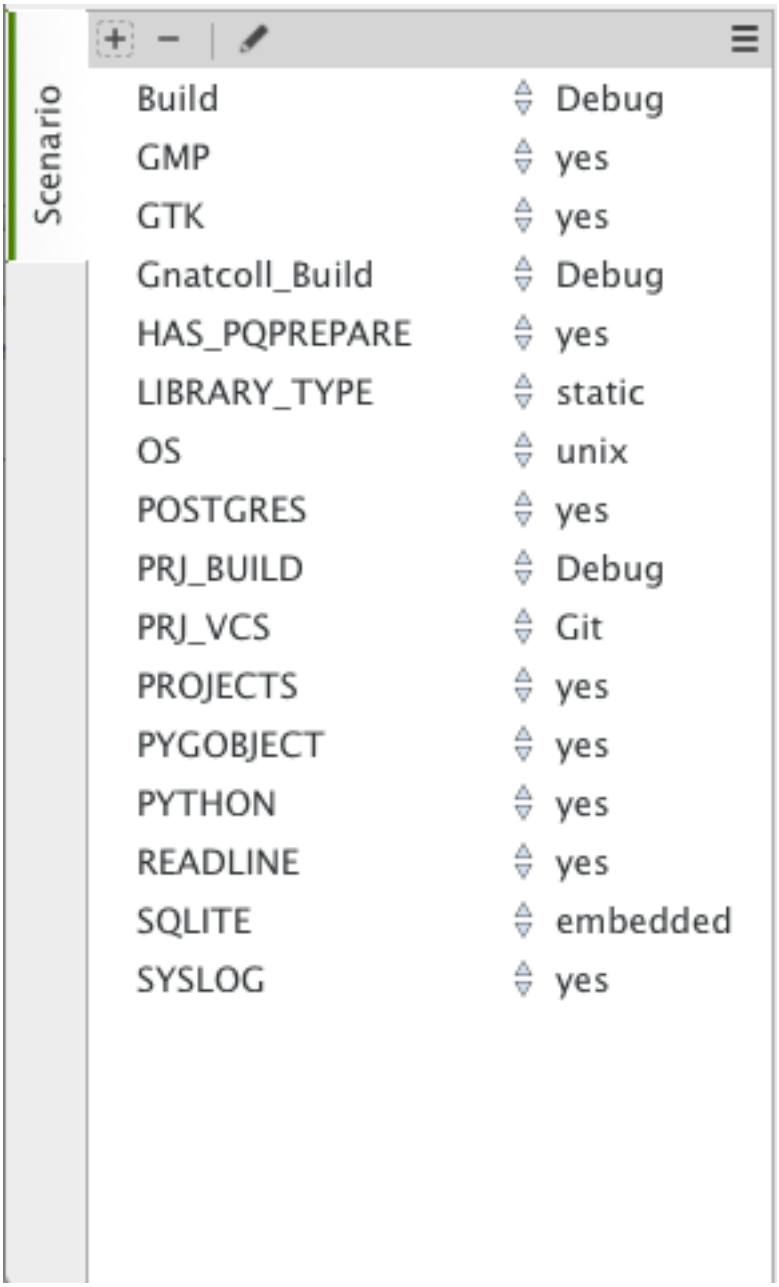
All the entries in the local settings menu can be manipulated via python extensions, which might be useful when writing your own plugins. Here are examples on how to do that:

```
# The 'Show flat view" local setting
GPS.Preference('explorer-show-flat-view').set(True)

# The 'Show absolute paths" local setting
```

```
GPS.Preference('explorer-show-absolute-paths').set(True)

# The 'Show hidden directories' local setting
GPS.Preference('explorer-show-hidden-directories').set(True)

# The 'Show empty directories' local setting
GPS.Preference('explorer-show-empty-directories').set(True)

# The 'Projects before directories' local setting
GPS.Preference('explorer-show-projects-first').set(True)

# The 'Show object directories' local setting
GPS.Preference('explorer-show-object-dirs').set(True)

# The 'Show runtime files' local setting
GPS.Preference('explorer-show-runtime').set(True)

# The 'Group by directories' local setting
GPS.Preference('explorer-show-directories').set(True)
```

1.11 The *Scenario* view





As described in the GNAT User's Guide, project files can be configured through external variables (typically environment variables). This means 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 view 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. Change the current value by clicking on it and selecting the new value among the ones that pop up.

Across sessions, GPS will remember the values you set for scenario variables. On startup, the initial values of the scenario variables come, in decreasing order of priority:

- from the `-X` command line arguments;
- from existing environment variables;

- from the value you set in a previous GPS session;
- from the default set in the project file;
- or else defaults to the first valid value for this variable

Whenever you change the value of any variable, GPS automatically recomputes the project and dynamically changes the list of source files and directories to reflect the new status of the project. Starting a new compilation at that point uses the new switches, and all aspects of GPS are immediately changed to reflect the new setup.

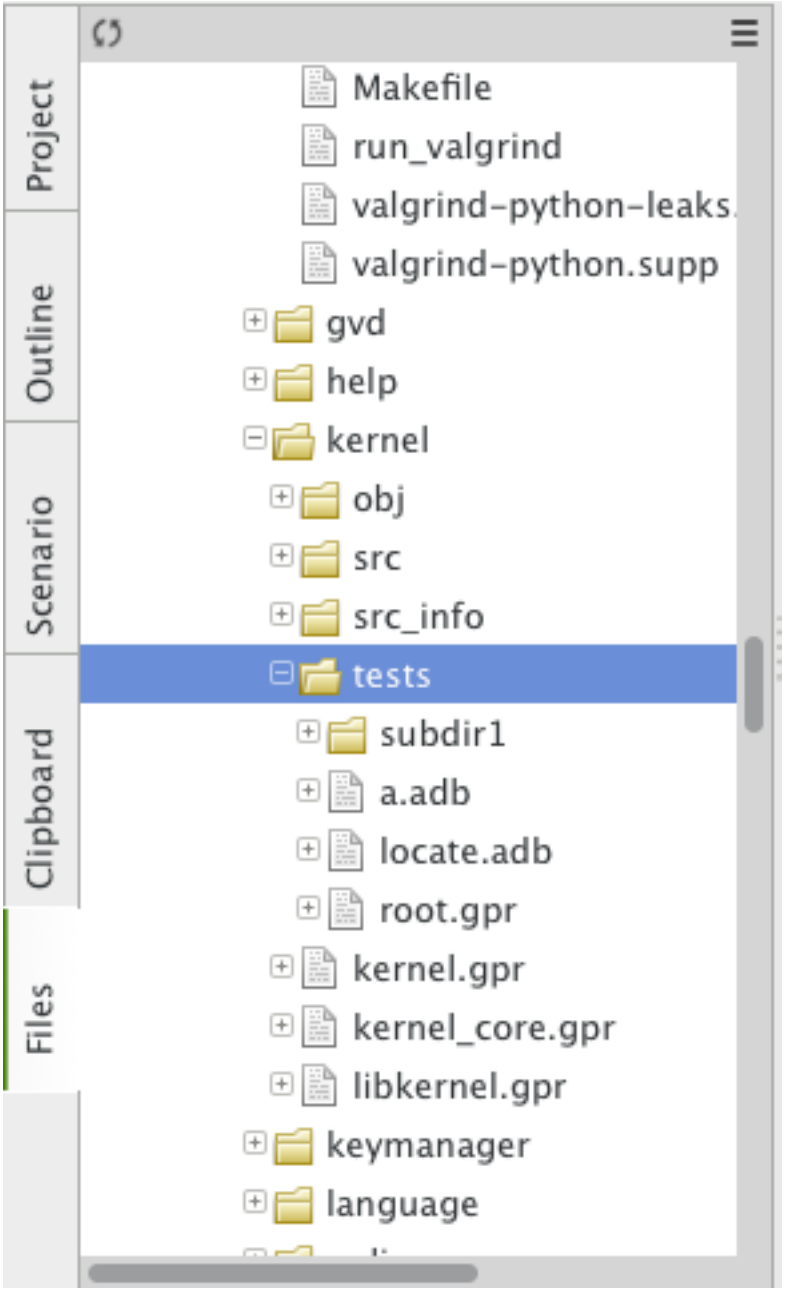
Create new scenario variables by selecting the + icon in the local toolbar of the *Scenario* view. Edit the list of possible values for a variable by clicking on the *edit* button in that toolbar. Delete a variable by clicking on the - button.

Each of these changes impacts the actual project file (`.gpr`), so you might not want to make them if you wrote the project file manually since the impact 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*). Like scenario variables, change the mode 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, use the local settings menu *Show build modes* to disable the display.

1.12 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 on the disk (including Windows drives). You can also explore each source file explored as described in *The Project view*. You can also drop files into the *Files* view to conveniently open a file.

By default, the *Files* view displays all files on disk. You can set filters through the local settings menu to restrict the display to the files and directories belonging to the project (use the *Show files from project only* menu).

1.13 The *Windows* view

The *Windows* view displays the currently opened windows. Open it via the *Tools* → *Views* → *Windows* menu.

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

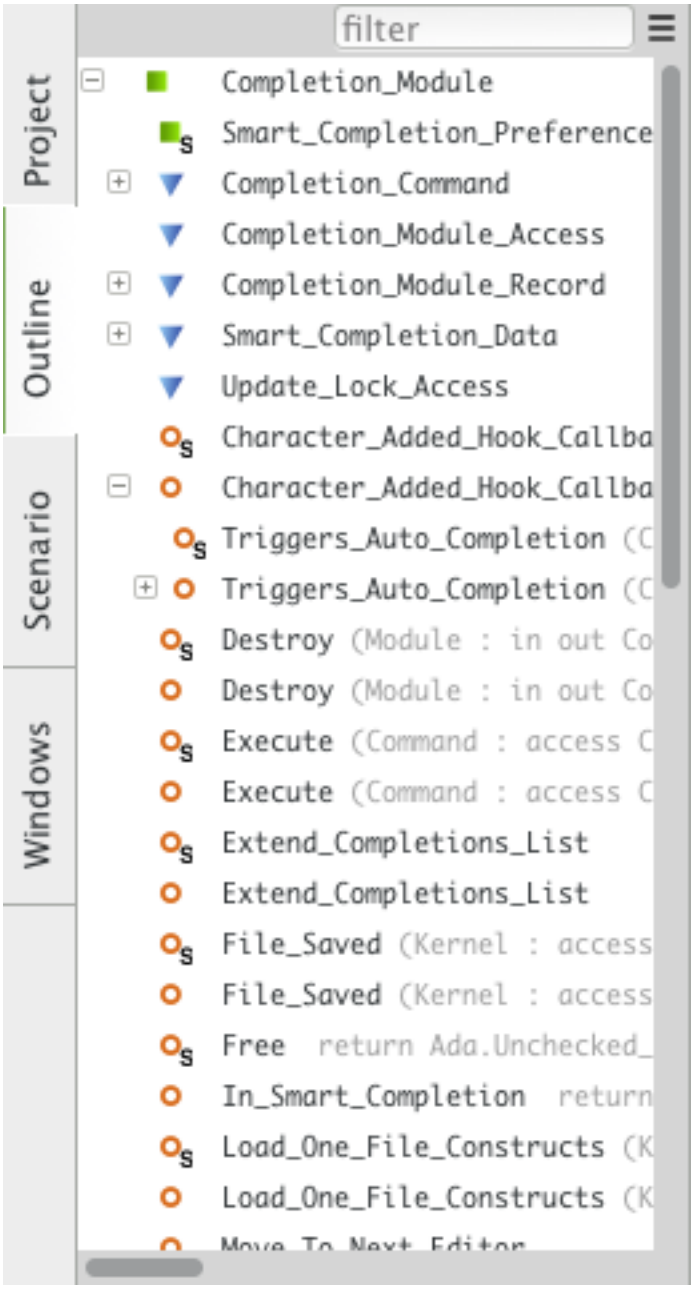
- Sorted alphabetically
- Organized by notebooks, as in the GPS window itself. This view is particularly useful if you have many windows open.

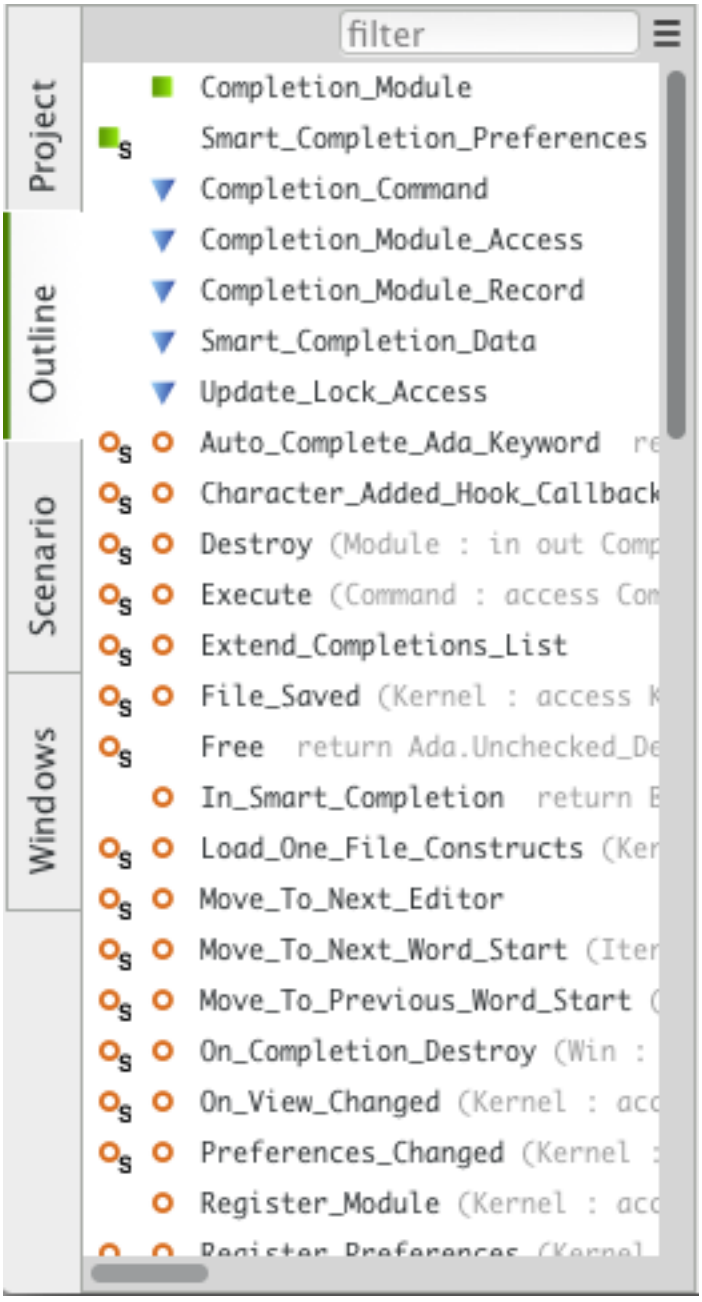
You can also choose, through the local configuration menu, whether only source editors should be visible or whether all windows should be displayed.

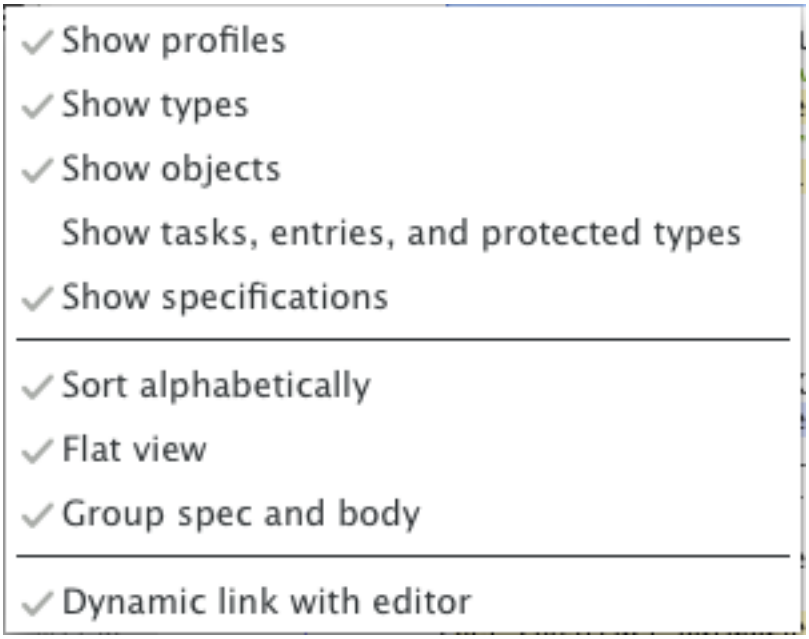
This view allows you to quickly select and focus on a particular window by clicking the corresponding line. If you leave the button pressed, you can drag the window to another place on the desktop (see the description of the *Multiple Document Interface*)

Select multiple windows by clicking while pressing the control or shift keys. You can then click in on the first button in the local toolbar to close all selected windows at once, which is a fast way to clean up your desktop after you have finished working on a task.

1.14 The *Outline* view




Project
Outline
Scenario
Windows
filter
Completion_Module
Smart_Completion_Preferences
Completion_Command
Completion_Module_Access
Completion_Module_Record
Smart_Completion_Data
Update_Lock_Access
Auto_Complete_Ada_Keyword
Character_Added_Hook_Callback
Destroy (Module : in out Comp
Execute (Command : access Com
Extend_Completions_List
File_Saved (Kernel : access K
Free return Ada.Unchecked_De
In_Smart_Completion return B
Load_One_File_Constructs (Ker
Move_To_Next_Editor
Move_To_Next_Word_Start (Iter
Move_To_Previous_Word_Start (
On_Completion_Destroy (Win :
On_View_Changed (Kernel : acc
Preferences_Changed (Kernel :
Register_Module (Kernel : acc



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

Exactly what is displayed depends on the language of the file. For Ada, C and C++ files, this view displays the list of entities declared at the global level in your current file (such as Ada packages, C++ classes, subprograms, and Ada types). This view is refreshed whenever the current editor is modified.

Clicking on any entity in this view automatically jumps to the corresponding line in the file (the spec or the body).

The local settings menu contains multiple check boxes you can use to alter how the outline view is displayed:

- *Show profiles*

  Indicates whether the list of parameters of the subprograms should be displayed. This is particularly useful for languages allowing overriding of entities.

- *Show types*, *Show objects*, *Show tasks, entries, and protected types*

  Controls the display of the specified categories of entities.

- *Show specifications*

  Indicates whether GPS displays 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 disabled, nested subprograms are displayed below the subprogram in which they are declared.

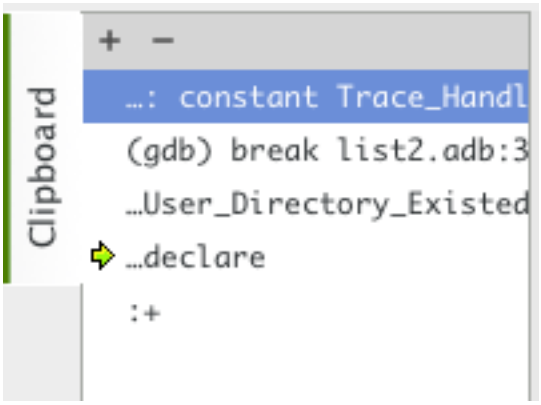
- *Group spec and body*

  Displays up to two icons on each line (one for the spec and one for the body if both occur in the file). Click on one of the icons to go directly to that location. If you click on the name of the entity, you are taken to its declaration unless it is already the current location in the editor, in which case you are taken to its body.

- *Dynamic link with editor*

  Causes the current subprogram to be selected in the outline view each time the cursor position changes in the current editor. This option will slow down GPS.

1.15 The *Clipboard* view



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

When you click the *Edit* → *Copy* or *Edit* → *Cut* menu, GPS adds the current selection to the clipboard. However, unlike many applications, GPS does not discard the previous contents of the clipboard, but instead saves it for future use. By default, up to 10 entries are saved, but you can change that number using the *Clipboard Size* preference.

When you select the *Edit* → *Paste* menu, GPS pastes the last entry added to the clipboard at the current location in the editor. If you then immediately select *Edit* → *Paste Previous*, this newly inserted text is removed and GPS instead inserts the second to last entry. You can keep selecting the same menu to insert progressively older entries.

This mechanism allows you to copy several noncontiguous lines from one place in an editor, switch to another editor, and paste all those lines without having to go back and forth between the two editors.

The *Clipboard* view graphically displays what is currently stored in the clipboard. Open it via the *Tools* → *Views* → *Clipboard* menu.

That view displays a list of entries, each of which is associated with one level of the clipboard. The text displayed for each entry is its first line containing non blank characters with leading characters omitted. GPS prepends or appends `[...]` if the entry is truncated. If you hover over an entry, a tooltip pops up displaying all lines in the entry.

In addition, one entry has an arrow on its left. This indicates the entry to be pasted if you select the *Edit* → *Paste* menu. If you instead select the *Edit* → *Paste Previous* menu, the entry below that is inserted instead.

If you double-click any of these entries, GPS inserts the corresponding text in the current editor and makes the entry you click current, so selecting *Edit* → *Paste* or the equivalent shortcut will insert that same entry again.

The local toolbar in the clipboard view provides two buttons:

- *Append To Previous*.

  The selected entry is appended to the one below and removed from the clipboard so that selecting *Edit* → *Paste* pastes the two entries simultaneously. Use this when you want to copy lines from separate places in a file, merge them, and paste them together one or more times later, using a single operation.

- *Remove*.

  The selected entry is removed from the clipboard.

The *Clipboard* view content is preserved between GPS sessions. However very large entries are removed and replaced with an entry saying "[Big entry has been removed]".

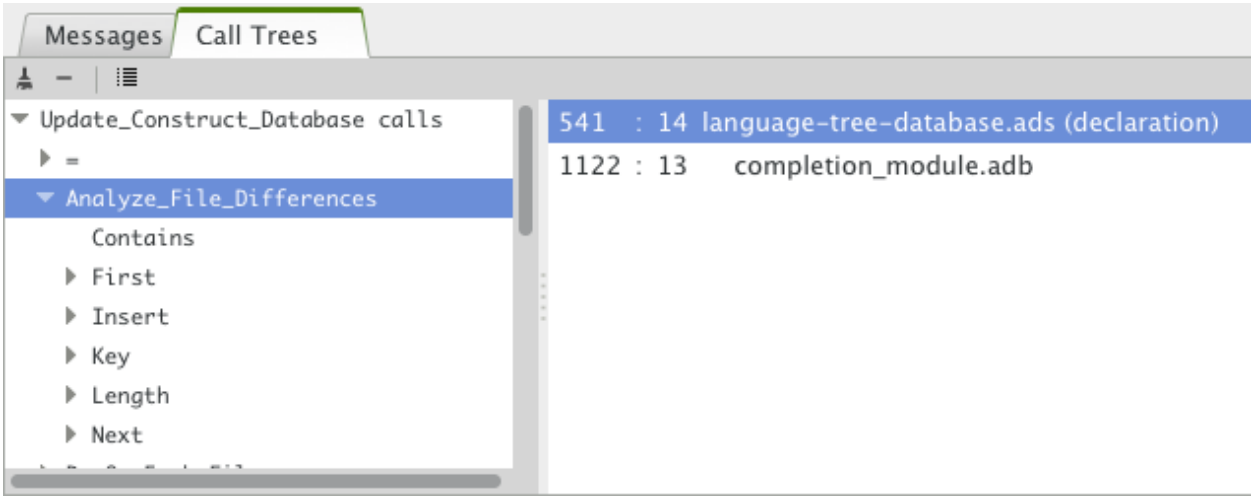
1.16 The *Call trees* view and *Callgraph* browser

These two views play similar roles in that they display the same information about entities, but in two different ways: the *Call tree* view displays the information in a tree, easily navigable and perhaps easier to manipulate when lots of entities are involved, and the *Callgraph* browser displays the information as graphical boxes that you can manipulate on the screen. The latter is best suited to generate a diagram that you can later export to your own documents.

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

Some references are displayed with an additional "(dispatching)" text, which indicates the call to the entity is not explicit in the sources but could potentially occur through dynamic dispatching. (This depends on what arguments are passed to the caller at run time; it is possible the subprogram is in fact never called.)

1.16.1 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.

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, making it very easy to get the information contained in a full callgraph tree. Closing and expanding a node again recomputes the callgraph for the entity.

The right side of the main tree contains a list displays the locations of calls for the selected entity. Click on an entry in this list to open an editor showing the corresponding location.

The *Call tree* 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 when GPS is restarted because its contents might be misleading if the sources have changed.

The local toolbar provides the following buttons:

- *Clear*

  Remove all entries from the Callgraph View.

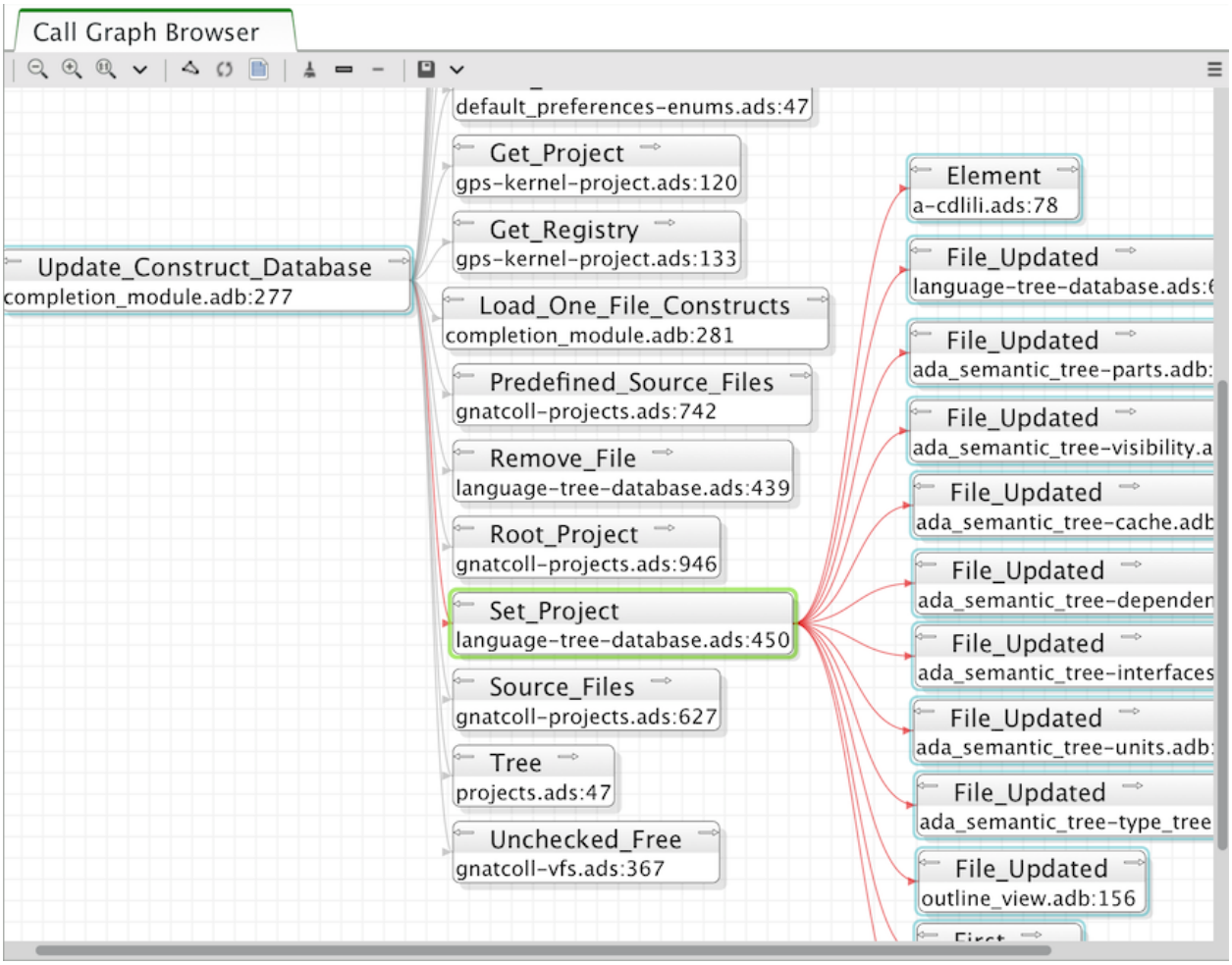
- *Remove entity*

  Remove the selected entity from the Callgraph View.

- *Collapse all*

  Collapse all the entities in the Callgraph View.

1.16.2 Callgraph browser



The *Callgraph* browser graphically displays the relationship between subprogram callers and callees. A link between two items indicates one of them is calling the other.

GPS provides special handling for renamed entities (in Ada): if a subprogram is a renaming of another, both items are displayed in the browser with a special hashed link between the two. Since the renamed subprogram does not have a proper body, you need to ask for the subprograms called by the renamed entity to get the list.

In this browser, clicking on the right arrow in the title bar displays all the entities called by the selected item. Clicking on the left arrow displays all the entities that call the selected item (i.e. its callers).

Open this browser by right-clicking on the name of an entity in a source editor or *Project* view and selecting one of the *Browsers* → *<entity> calls*, *Browsers* → *<entity> calls (recursive)*, or *Browsers* → *<entity> is called by* menus.

All boxes in this browser display the location of their declaration and the list of all 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.

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

- *Go To Spec*

  Open a source editor displaying the declaration of the entity.

- *Go To Body*

  Open a source editor displaying the body of the entity.

- *Locate in Project View*

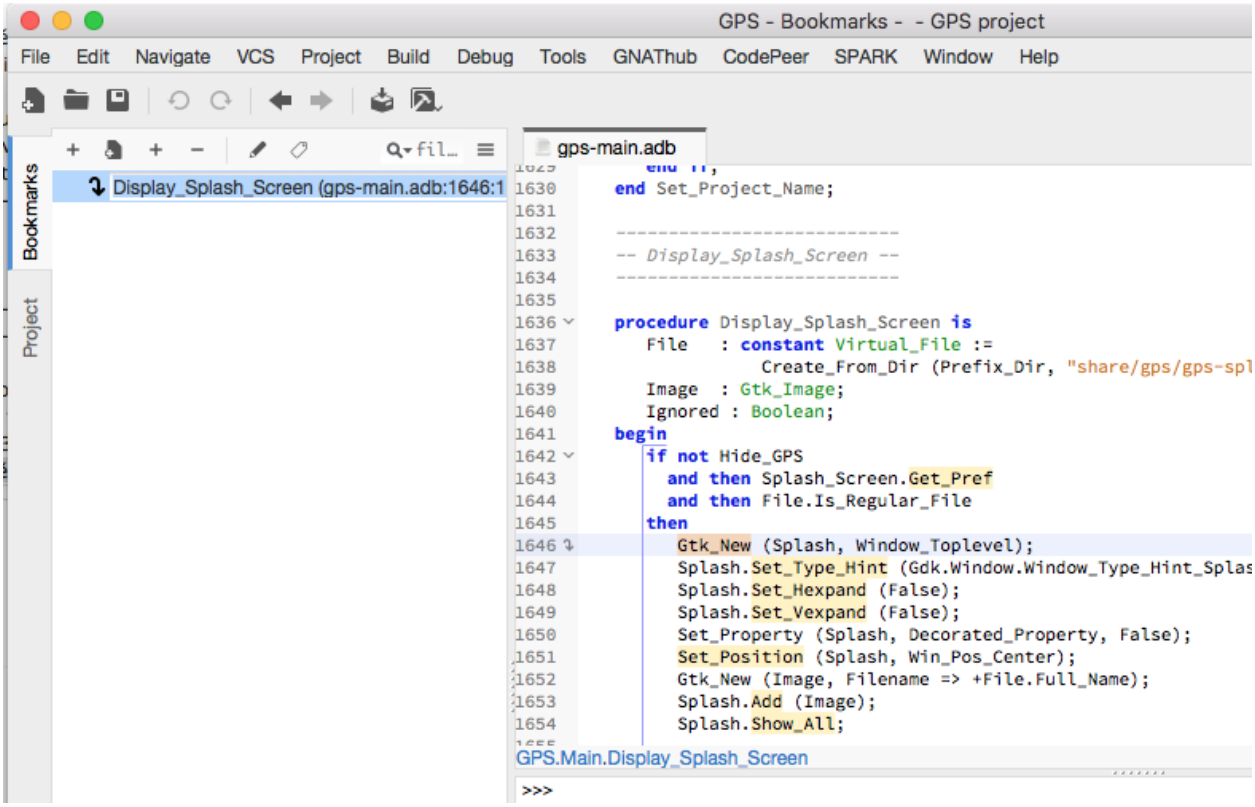
  Move the focus to the project view, and select the first node representing the file in which the entity is declared. This makes it easier to see which other entities are declared in the same file.

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

1.17 The *Bookmarks* view

1.17.1 Basic usage: Creating a new bookmark

The basic usage of bookmarks is as follows: you open a source editor and navigate to the line of interest. You can then create a new bookmark by either using the menu *Edit* → *Create Bookmark* or by opening the *Bookmarks* view (*Tools* → *Views* → *Bookmarks*) and then clicking on the [+] button in the local toolbar. In both cases, the *Bookmarks* view is opened, a new bookmark is created and selected so that you can immediately change its name.



The default name of bookmark is the name of the enclosing subprogram and the initial location of the bookmark (*file:line*). But you can start typing a new name, and press Enter to finally create the bookmark.

In practice, this is really just a few clicks (one of the menu and press `Enter` to use the new name), or even just two key strokes if you have set a keyboard shortcut for the menu, via the Preferences dialog.

At any point in time, you can rename an existing bookmark by either clicking on the button in the local toolbar, or simply with a long press on the bookmark itself.

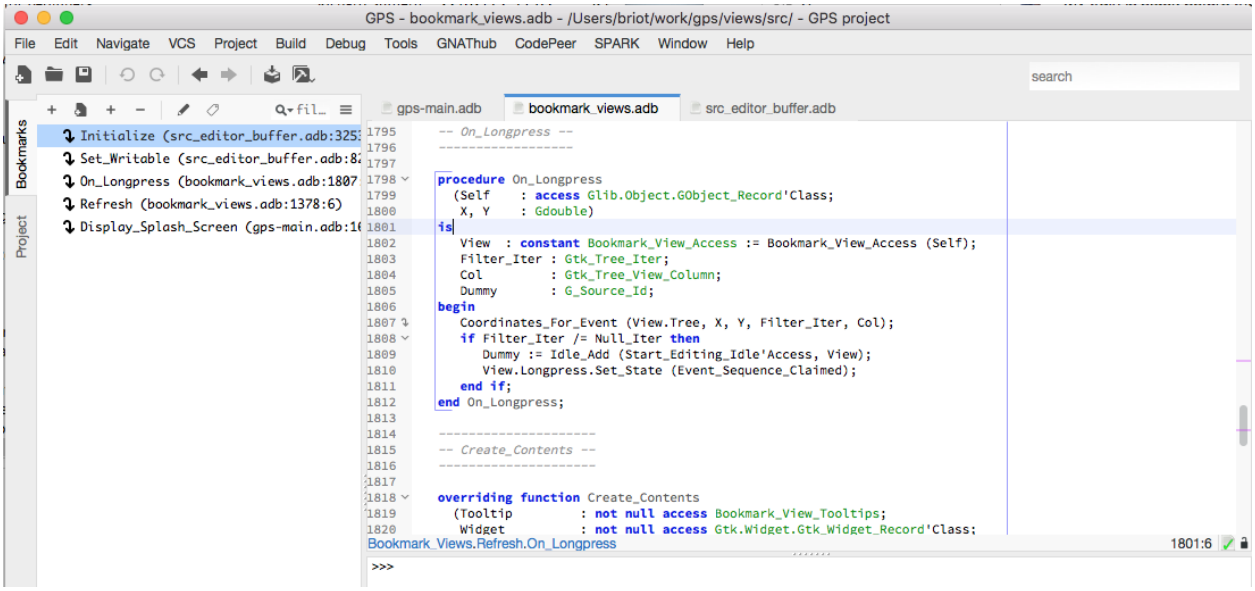
Note the goto icon on the left of the editor line 1646, which indicates there is a bookmark there, as well as the colored mark in the editor scrollbar that helps navigate in the file.

Even though the default name of the bookmark includes a file location, the major benefit of the bookmarks is that they will remain at the same location as the text is edited. In our example, if we add a new subprogram before *Display_Splash_Screen*, the bookmark will still point at the line containing the call to *Gtk_New*, even though that line might now be 1700 for instance.

Of course, GPS is not able to monitor changes that you might do through other editors, so in this case the marks might be altered and stop pointing to the expected location.

1.17.2 Adding more bookmarks

We can create any number of bookmarks, and these have limited impact on performance. So let's do that and create a few more bookmarks, in various files. As you can see in the scrollbar of the editor, we have two bookmarks set in the file bookmark_views.adb, and we can easily jump to them by clicking on the color mark.

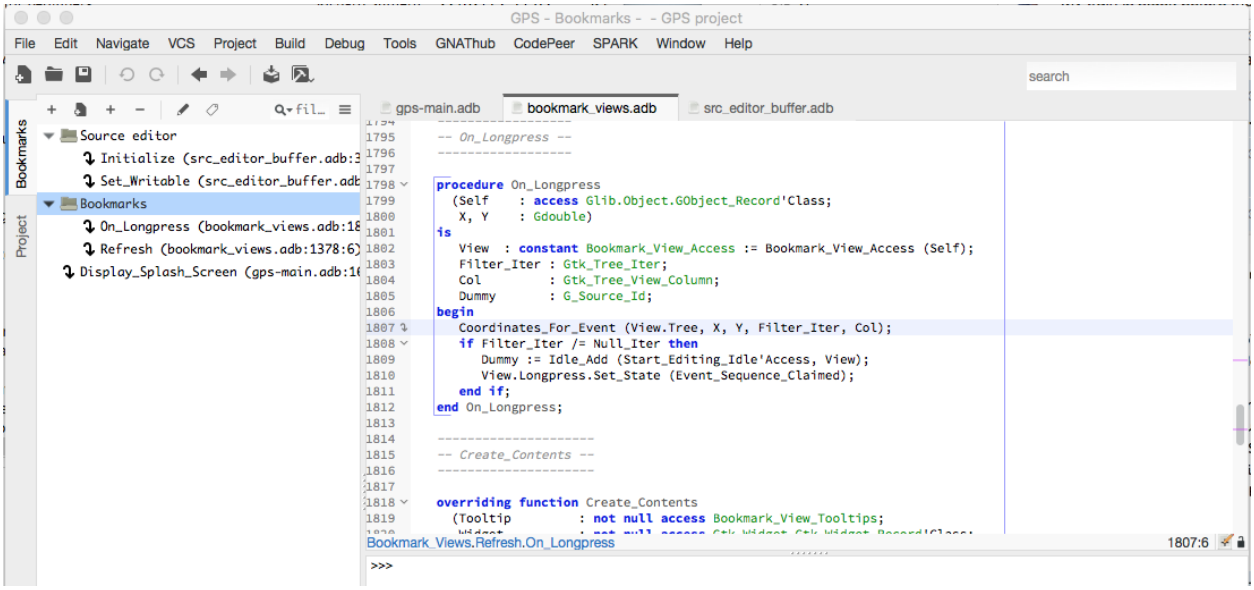


But of course, it is much simpler to double-click inside the *Bookmarks* view itself, on the bookmark of interest to us.

At this point, we have a rather long unorganized list of bookmarks, let's improve.

1.17.3 Organizing bookmarks into groups

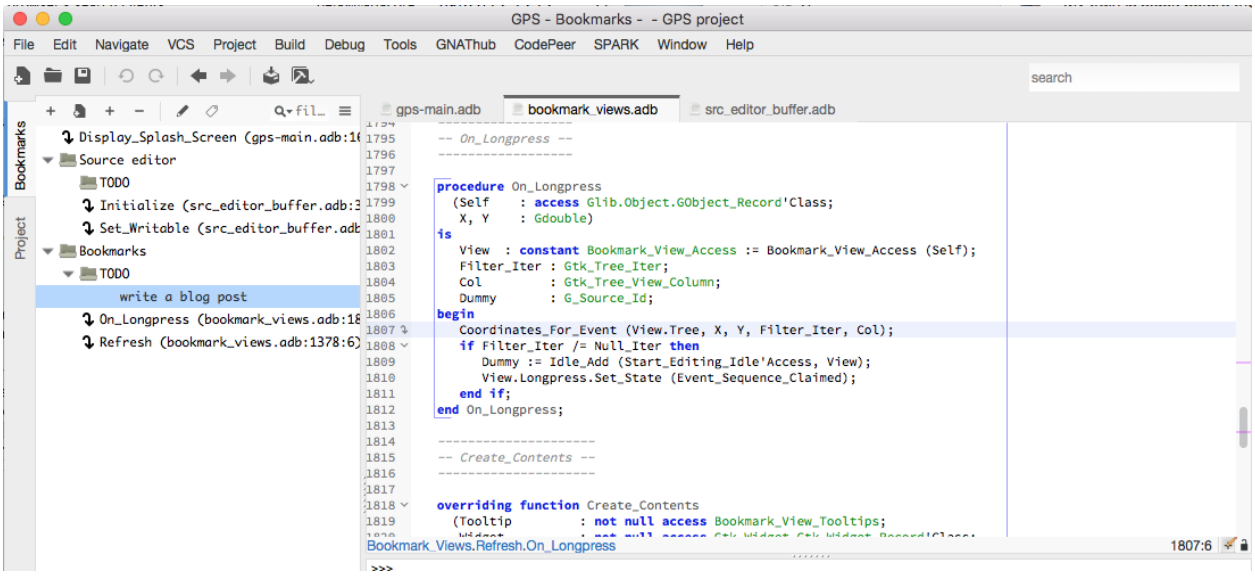
When we create new bookmarks, GPS adds them at the top of the list. We might want to organize them differently, which we can do simply with a drag and drop operation: select the bookmark, keep the mouse pressed, and move it to a better place in the list.



Things become more interesting when you drop a bookmark on top of another one. In this case, GPS creates a group that contains the two bookmarks (and that basically behaves like a folder for files). The group is immediately selected so that you can rename it as you see fit.

In our example, we created two groups, corresponding to two features we are working on.

Groups can be nested to any depth, providing great flexibility. So let's create two nested groups, which we'll name TODO, beneath the two we have created. This is a great way to create a short todo list: one top-level group for the name of the feature, then below one group for the todo list, and a few additional bookmarks to relevant places in the code.



To create these additional groups, we will select the Source editor group, then click on the *Create New Group* button in the local toolbar, and type "TODO<enter>". This will automatically add the new group beneath Source editor. Let's do the same for the bookmarks groups. These two groups are empty for now.

Let's add new entries to them. if we already know where code should be added to implement the new todo item, we can do as before: open the editor, select the line, then click on the *[+]* button. Most often, though, we don't yet know where the implementation will go.

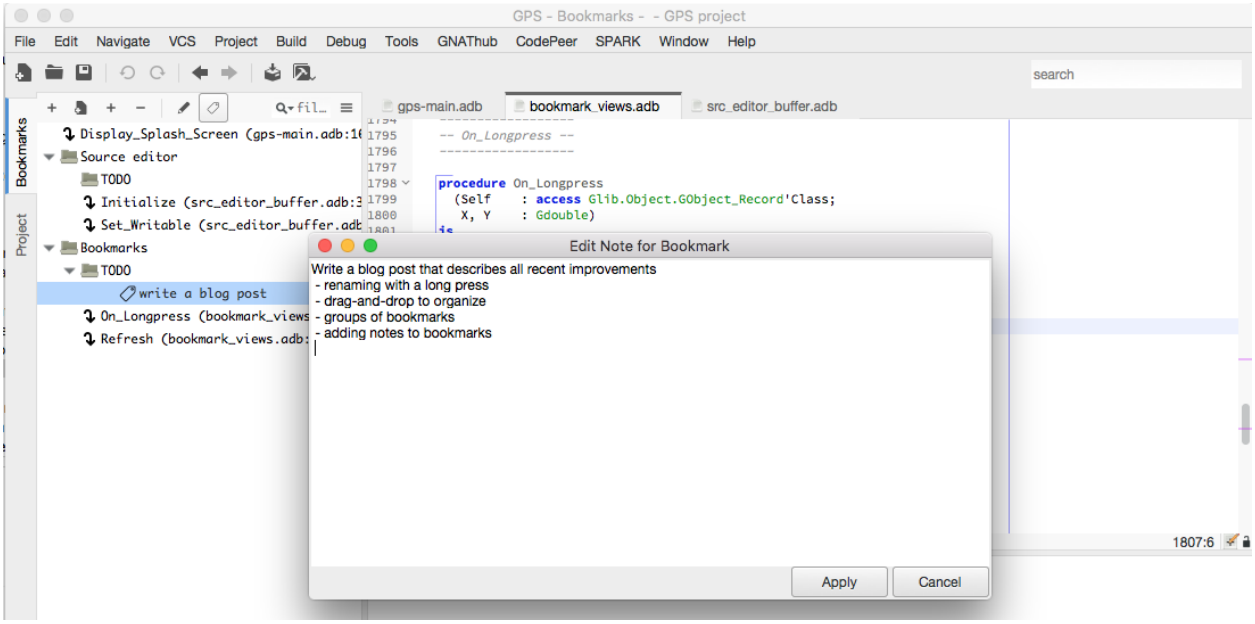
So we want to create an unattached bookmark. Using the name bookmark here is really an abuse of language, since these have no associated source location. But since they are visible in the *Bookmarks* view, it is convenient to name them bookmarks.

To create them, let's select one of the TODO groups, then select the *Create Unattached Bookmark* in the local toolbar, and immediately start typing a brief description of the todo. As you can see in the screenshot, these bookmarks do not have a goto icon, since you cannot double click on them to jump to a source location.

When you *delete* a group, all bookmarks within are also deleted. So once you are done implementing a feature, simply delete the corresponding group to clean up the bookmarks view.

1.17.4 Adding notes

The short name we gave the bookmark is not enough to list all the great ideas we might have for it. Fortunately, we can now add notes to bookmarks, as a way to store more information.



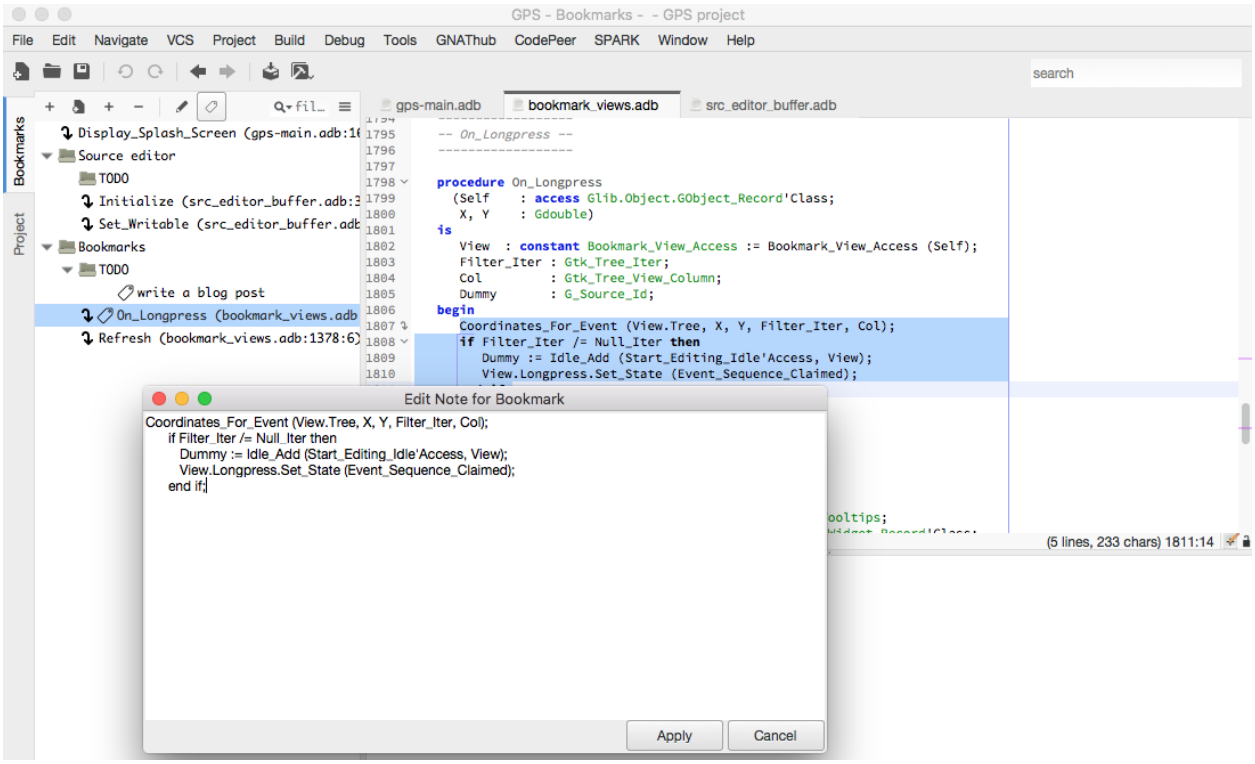
Let's select the "write a blog post" item, then click on the *Edit Note* button in the local toolbar. This opens a small dialog with a large text area where we can type anything we want. Press *Apply* to save the text.

Note how a new tag icon was added next to the bookmark, to indicate it has more information. You can view this information in one of three ways:

- select the bookmark, and click again on the *Edit Note* button as before
- *double-click* on the tag icon.
- leave the mouse hover the bookmark line. This will display a tooltip with extra information on the bookmark: its name, its current location and any note it might have. This is useful if you only want to quickly glance at the notes for one or more bookmarks

1.17.5 Add note with drag and drop

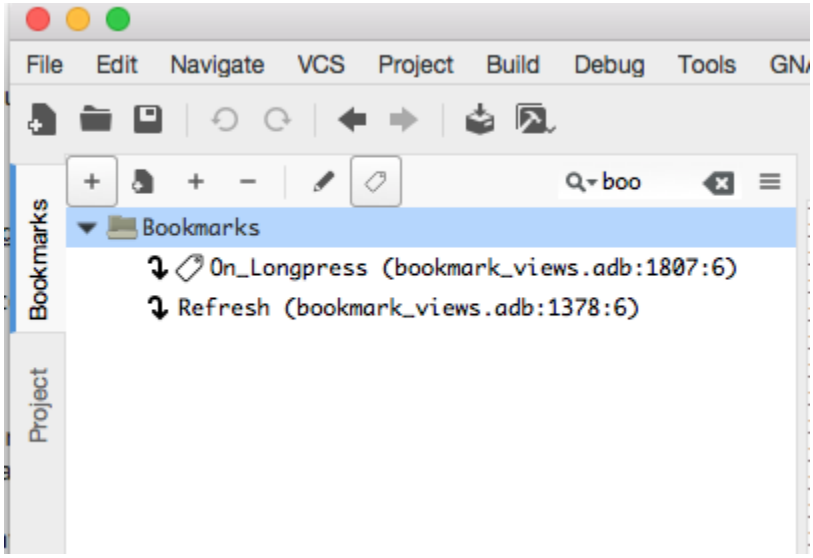
Sometimes, though, you want to associate code with the note (i.e. the bookmark should not only point to a location, but you also want to remember the code that was in that location). The simplest to do this is to select the text in the editor, and then drag and drop the selected text directly onto the bookmark. This will create a note (if needed) or add to the existing note the full selected text.



In the tooltips, we use a non-proportional font, so that the code is properly rendered and alignment preserved.

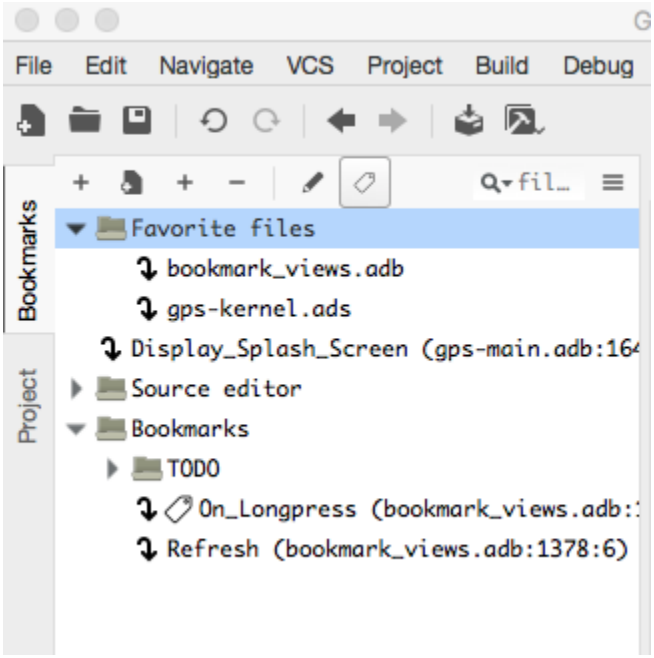
1.17.6 Filtering bookmarks

If you start creating a lot of bookmarks, and even if you have properly organized them into groups, it might become difficult to find them later on. So we added a standard filter in the local toolbar, like was done already for a lot of other views. As soon as you start typing text in that filter, only the bookmarks that match (name, location or note) are left visible, and all the others are hidden.



1.17.7 Favorite files

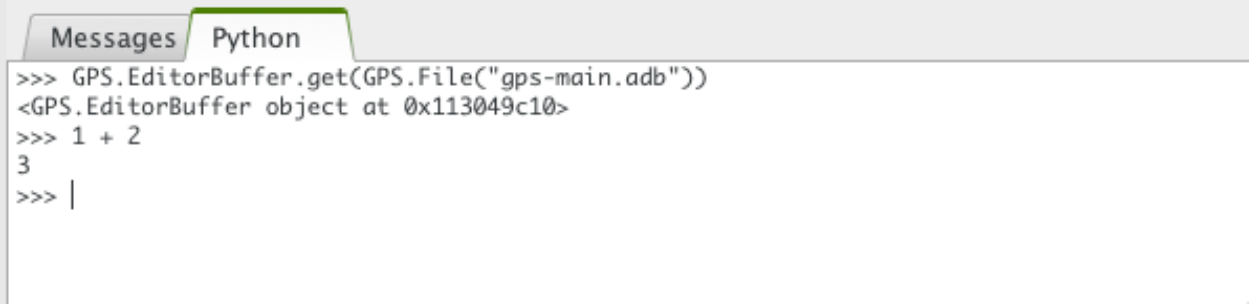
GPS provides a large number of ways to navigate your code, and in particular to open source files. The most efficient one is likely the omni-search (the search field at the top-right corner).



But some users like to have a short list of favorite files that they go to frequently. The *Bookmarks* view can be used to implement this.

Simply create a new group (here named *Favorite* files), and create one new bookmark in this group for each file you are interested in. I like to create the bookmark on line 1, but I always remove the line number indication in the name of the bookmark since the exact line is irrelevant here.

1.18 The *Python* Console



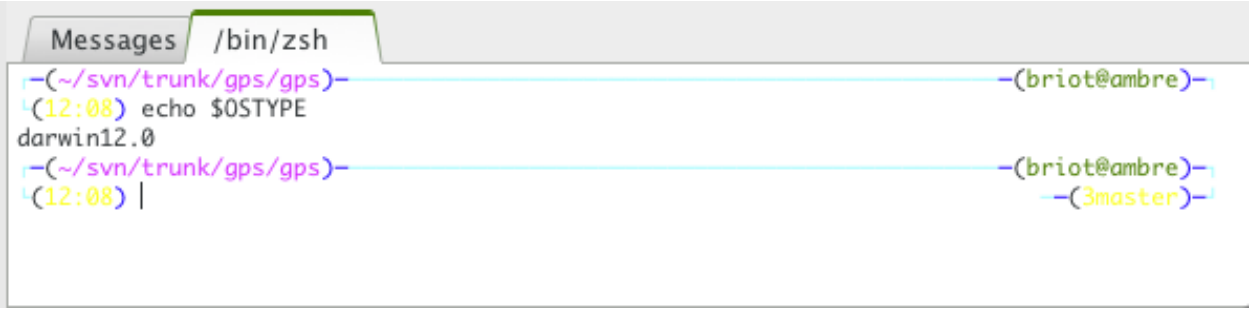
These consoles provide access to the various scripting languages supported by GPS, allowing you to type interactive commands such as editing a file or compiling without using the menu items or the mouse.

The menu *Tools* → *Consoles* → *Python* opens the python console. Python is the preferred language to customize GPS (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.

Both consoles provide a history of previously typed commands. Use the `up` and `down` keys to navigate through the command history.

1.19 The OS Shell Console



GPS also provides an OS shell console, providing an access to the underlying OS shell (as defined by the *SHELL* or *COMSPEC* environment variables).

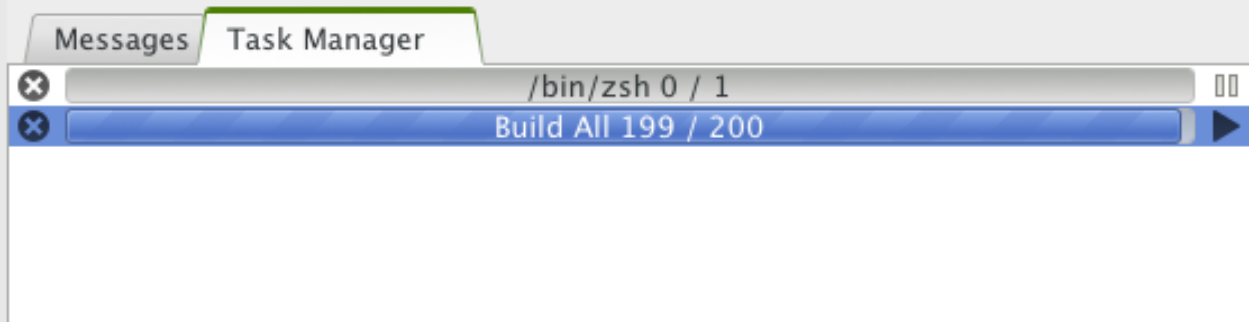
Open this console via the *Tools* → *Consoles* → *OS Shell* menu, which is available only if the plugin `shell.py` was loaded in GPS (the default). Check the documentation of that plugin, which lists a few settings that might be useful.

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

1.20 The Execution window

When a program is launched using the *Build* → *Run* menu, GPS creates a new execution window allowing input and output for the program. To allow post-mortem analysis and copy/pasting, GPS does not close execution windows when the program terminates; you must close them manually. If you try to close the execution window while the program is still running, GPS displays a dialog window asking if you want to kill it.

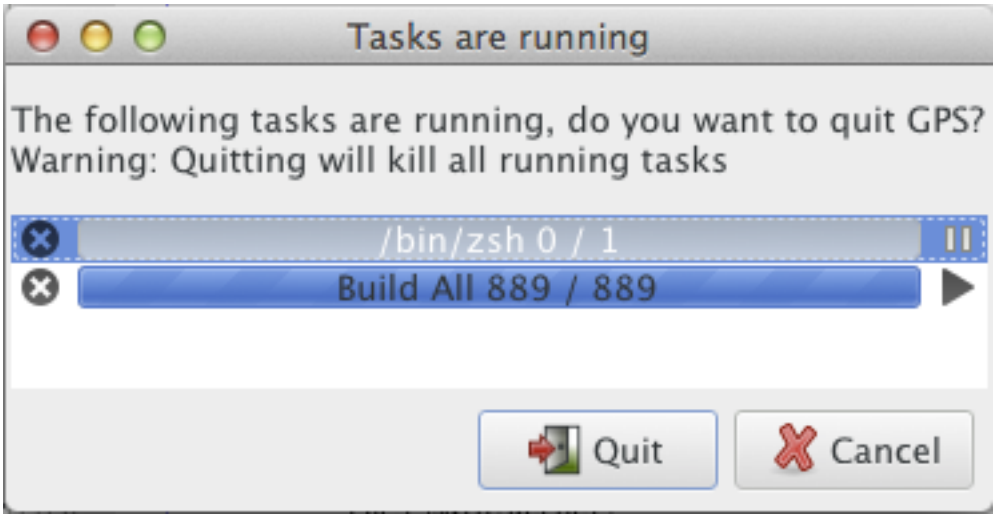
1.21 The *Tasks* view



The *Tasks* view displays all running GPS operations currently running in the background, such as builds, searches, or VCS commands.

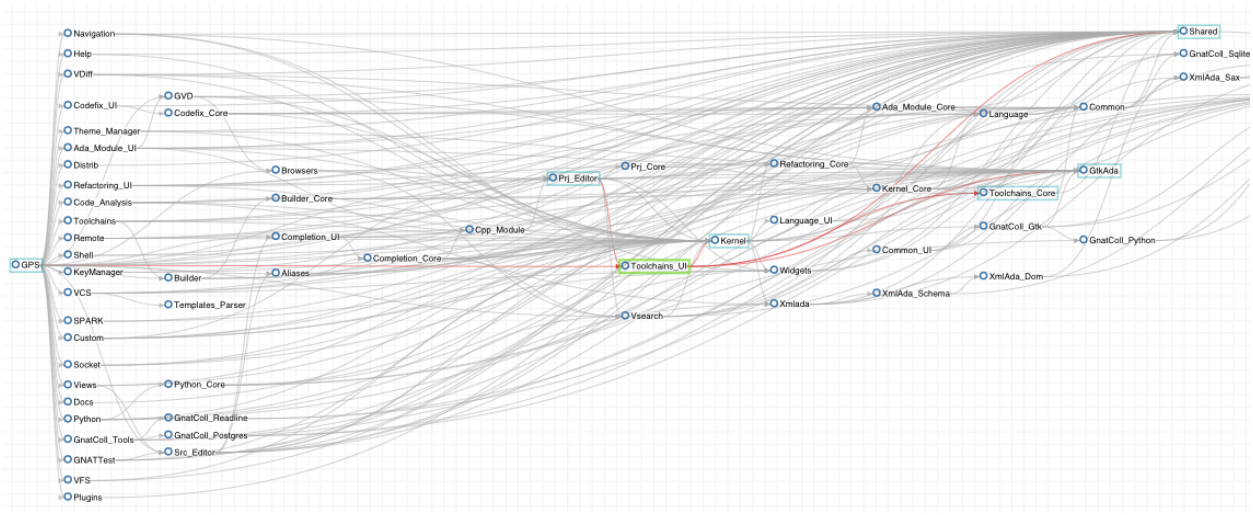
For each task, the *Tasks* view displays its status and current progress. Suspend the execution of a task by clicking the small *pause* button next to the task. Or kill a task by clicking the *interrupt* button.

Open the *Tasks* view by double clicking on the progress bar in the main toolbar or using the *Tools* → *Views* → *Tasks* menu. You can move it placed anywhere on your desktop.



If there are tasks running when exiting GPS, it displays a window showing those tasks. You can kills all remaining tasks and exit by pressing the confirmation button or continue working in GPS by pressing the *Cancel* button.

1.22 The *Project Browser*



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

Access it through the contextual menu in the *Project* view by selecting the *Show projects imported by...* menu when right-clicking on a project node.

Click on the left arrow in the title bar of a project to display all projects that import that project. Click on the right arrow to display all projects imported by that project.

Right-clicking on a project brings up a menu containing several items. Most are added by the project editor and provide direct access to such features as editing the properties of the project, adding dependencies.

Some items in the menu are specific to the *Project Browser*:

- *Locate in Project View*

  Switch the focus to the *Project* view and highlight the first project node matching the project. This is a convenient way to get information such as the list of directories or source files for a project.

- *Show projects imported by...*

  Like the right arrow in the title bar, displays all the projects in the hierarchy that are directly imported by the selected project.

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

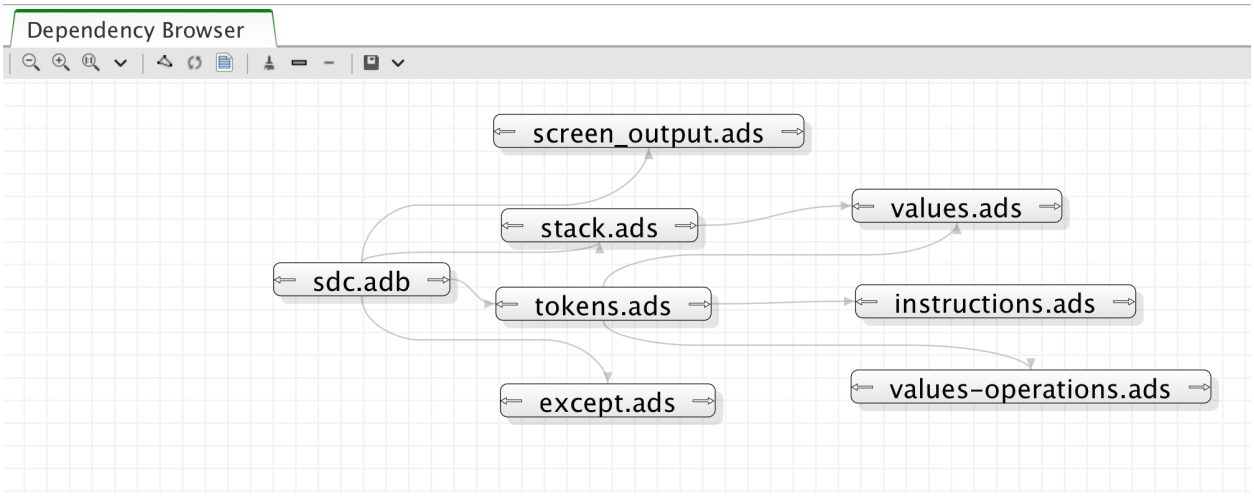
  Display all dependencies recursively for the project (i.e., the projects it imports directly and the projects they import).

- *Show projects importing...*

  Like the left arrow in the title bar, display all the projects that directly import the selected project.

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

1.23 The *Dependency Browser*



The dependency browser displays dependencies between source files. Each item in the browser represents one source file. Click on the right arrow in the title bar to display the list of files the selected file depends on. A file depends on another if it explicitly imports it (**`with`** statement in Ada, or **`#include`** in C/C++). Implicit dependencies are currently not displayed in this browser since you can access that information by opening the direct dependencies. Click on the left arrow in the title bar to 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 entries:

- *Show dependencies for ...*

  Like clicking on the right arrow for a file already in the browser, displays the direct dependencies for that file.

- *Show files depending on ...*

  Like clicking on the left arrow for a file already in the browser, displays 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...*

  Display an external dialog where you can select the name of a file to analyze.

- *Recompute dependencies*

  Check that all links displays in the dependency browser are still valid. Any that not are removed. The arrows in the title bar are also reset if new dependencies were added for the files. Also recompute the layout of the graph and changes the current position of the boxes. However, the browser is not refreshed automatically, since there are many cases where the dependencies might change.

- *Show system files*

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

- *Show implicit dependencies*

  Indicates whether implicit dependencies should also be displayed for files. Implicit dependencies are ones required to compile the selected file but not explicitly imported through a **`with`** or **`#include`** statement. For example, the body of a generic in Ada is an implicit dependency. Whenever an implicit dependency is modified, the selected file should be recompiled as well.

The contextual menu available by right clicking on an item also contain these entries:

- *Analyze other file*

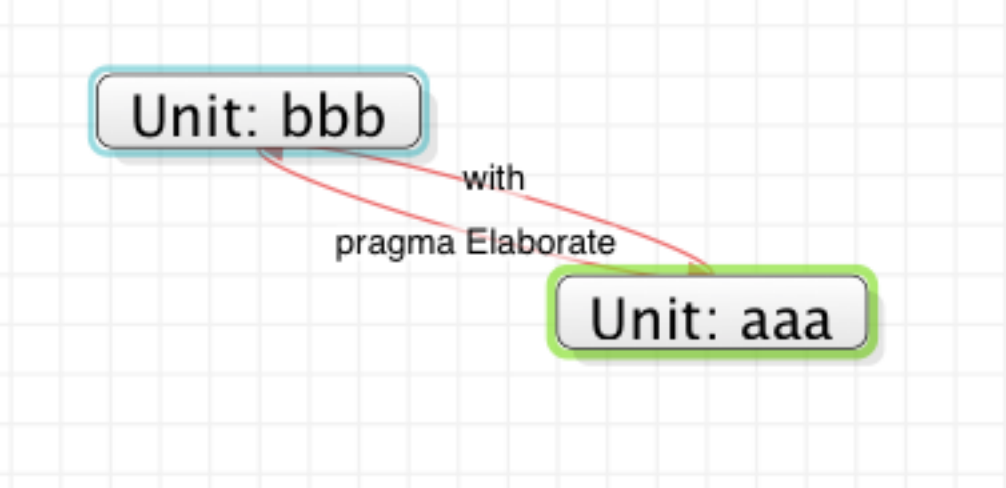
  Open a new item in the browser, displaying the files associated with the selected one. In Ada, this is the body if you clicked on a spec file, or vice versa. In C, it depends on the naming conventions you specified in the project properties, but it generally goes from a `.h` file to a `.c` file and back.

- *Show dependencies for ...*

  These have the same function as in the project view contextual menu

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

1.24 The *Elaboration Circularities* browser



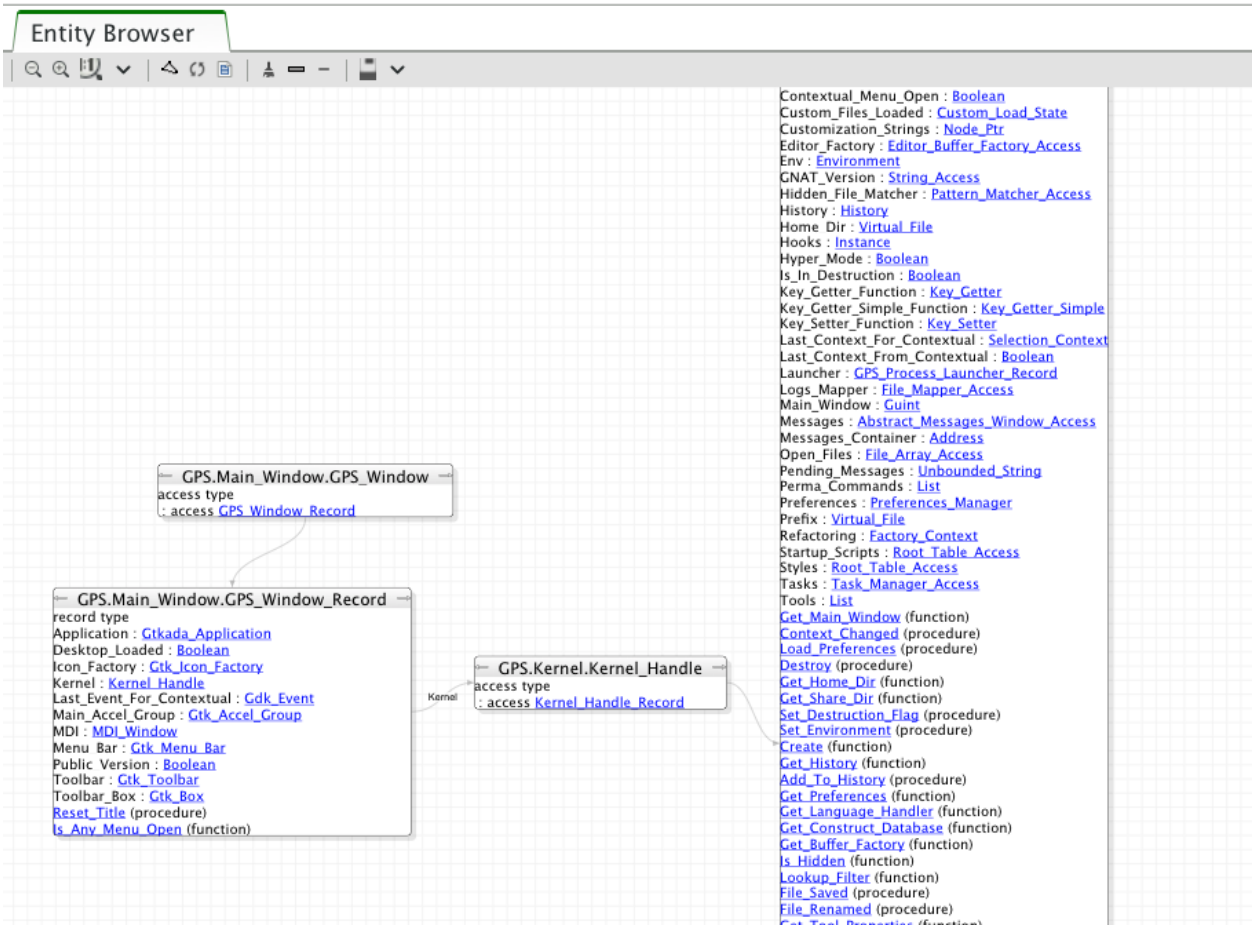
GPS detects elaboration cycles reported by build processes and constructs a visual representation of elaboration dependencies in an *Elaboration Cycles* browser.

This visual representation depicts program units as items in the browser 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 and connected by links labeled "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 browsers* for more capabilities of GPS browsers.

1.25 The *Entity* browser



The *Entity* browser displays static information about any source entity. What is displayed for each entity depends on the type of the entity, but are normally other entities. For example:

- `Ada record / C struct`

  The list of fields is displayed.

- `Ada tagged type / C++ class`

  The list of attributes and methods is displayed.

- `Subprograms`

  The list of parameters is displayed

- `Packages`

  The list of all the entities declared in that package is displayed

Access this browser via the *Browsers* → *Examine entity* contextual menu in the project view and source editor when clicking on an entity.

Most entities displayed are clickable (by default, they appear as underlined blue text). Clicking on one opens a new item in the entity browser for the selected entity.

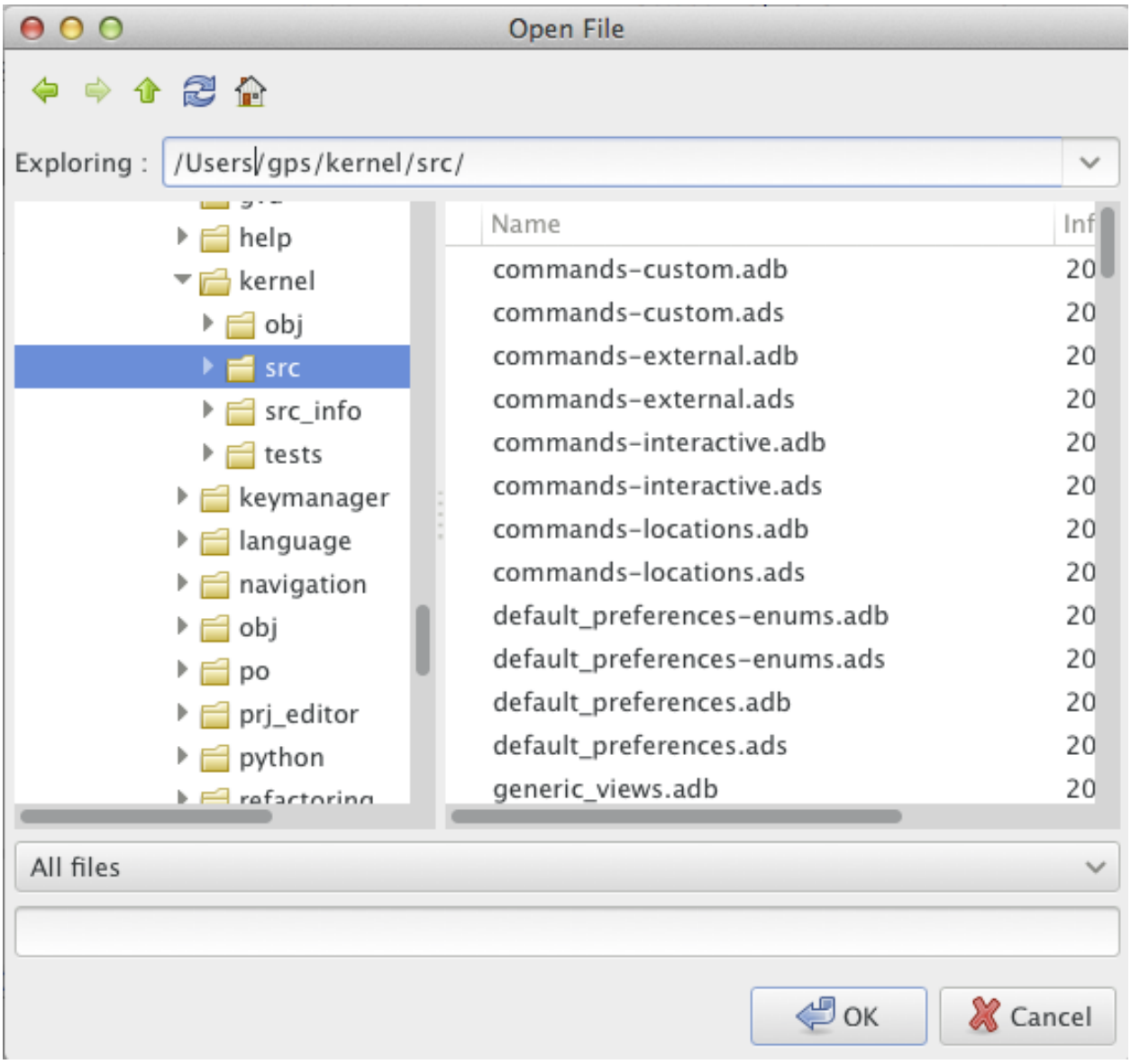
You can display the parent entities for an entity. For example, in a C++ class or Ada tagged type, this is the type it derives from. Display the parent by clicking on the up arrow in the title bar of the entity.

Similarly, you can display child entities (for example, types that derive from the item) by clicking on the down arrow in the title bar.

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

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

1.26 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 provides the following contents:

- A tool bar on the top consists of five buttons:
  - *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. Modify the current directory by modifying the text entry and pressing `Enter` or by clicking on the right arrow and choosing a previous directory in the pop down list displayed.
- A directory tree. 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 next directory, + and − to expand and collapse the current directory, and `backspace` to select the parent directory.
- A filter area. Depending on the context, one of several filters are available to select only a subset of files to display. The filter *All files* is always available and displays all files in the selected directory.
- A file list. This area lists the files contained in the selected directory. If you specified a filter, only the matching files are displayed. Depending on the context, the list of files may include additional information about the files such as the type of file or its size.
- A file name area. This area displays the name of the current file, if any. You can also type a file or directory name, with file completion provided by the `Tab` key.
- A button bar with the *OK* and *Cancel* buttons. When you have selected the desired file, click *OK* to confirm or click *Cancel* at any time to cancel the file selection.

CHAPTER

TWO

MULTIPLE DOCUMENT INTERFACE

All windows (whether editors or views) 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 where related windows are put into a larger window which is itself under control of the system or the windows manager.

By default, no matter how many editors and views you opened, your system still sees only one window (on Windows systems, the task bar shows only one icon). You can organize the GPS windows in whatever way you want, all inside the GPS main window. This section describes the capabilities GPS provides to help you do this.

2.1 Window layout

The GPS main window is organized into various areas. The most important distinction is between the central area (which usually occupies the most space) and the side areas (which include the top and bottom areas). Some windows in the GPS area are restricted to either the central or the side areas. You can split each area into smaller areas, as described below. Each area can contain any number of windows, organized into notebooks with tabs, possibly displaying names. Right-clicking on a tab displays a contextual menu providing capabilities showing the *Tabs location* and *Tabs rotation*, which allows you to display the tabs on any side of the notebook or make them vertical if you want to save screen space.

The central area can contain the source editors (which can only go there) as well as larger views like browsers. The contents of the central area are preserved when switching perspectives (see below).

2.2 Selecting Windows

Only one window is selected in GPS (the **active window**) at a time. Select a window by clicking on its tab, which becomes a different color, or selecting its name in the *Window* menu. Or use the *Windows* view (see *The Windows view*), which also provides a convenient mechanism for slecting multiple windows at once.

Finally, you can select windows with the omni-search, the search field in the global toolbar. One of the contexts for the search is the list of opened windows. 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*).

Any window whose name contains the specified letter matches the search field. For example, if you are currently editing the files `unit1.adb` and `file.adb`, pressing `t` leaves only `unit1.adb` selectable.

2.3 Closing Windows

Wherever a window is displayed, you can close it by clicking the small *X* icon in its tab or selecting the window by clicking on its tab and selecting the *Window* → *Close* menu.

When you close a window, the focus is set to the window in the same notebook that previously had the focus. If you open an editor as a result of a cross-reference query, close that editor to go back to where you were.

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

There is a *Close all other editors* menu in the notebook tab when you are in an editor, which closes most windows except a single editor, the one you are using.

2.4 Splitting Windows

You can split windows horizontally and vertically in any combination. To do this requires at least two windows (for example text editors or browsers) present in a given notebook. Select either the *Window* → *Split Horizontally* or *Window* → *Split Vertically* menus to split the selected window. In the left (respectively, top) pane, the currently selected window is put on its own. The rest of the previously superimposed windows are put in the right (respectively, bottom) pane. You can further split these remaining windows to achieve any desired layout.

You can resize any split windows by dragging the handles that separate them.

You may want to bind the key shortcuts to the *Window* → *Split Horizontally* and *Window* → *Split Vertically* menus using the key manager. If you want to achieve an effect similar to standard Emacs behavior (where `control-x 2` splits a window horizontally and `control-x 3` splits a window vertically), use the key manager (see *The Key Shortcuts Editor*).

Moving Windows shows how to split windows using drag-and-drop, which is the fastest way.

You can put several editors or browsers in the same area. In that case, they are grouped together in a notebook; select any of them by clicking on the corresponding tab. If there are many windows, two small arrows appear on the right of the tabs. Click these arrows to show the remaining tabs.

GPS changes the color and size of the title (name) of a window in the notebook tab to indicate that the window content has been updated but the window is not visible. This commonly occurs when new messages have been written in the *Messages* or *Console* views.

2.5 Floating Windows

You may prefer to have several top-level windows under direct control of your system's window manager. For example, you want to benefit from some options your system might provide such as virtual desktops, different window decoration depending on the window's type, transparent windows, and/or multiple screens.

You can make any window currently embedded in the MDI a **floating window** by selecting the window and selecting the *Window* → *Floating* menu. The window is detached and you can move it anywhere on your screen, even outside GPS's main window.

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

The second method assumes you have set the preference *Destroy Floats* in the *Edit* → *Preferences* menu to false. If so, you can close the floating window by clicking the close button in the title bar; the window is put back in GPS's main windows. If you want to close the window, you need to click the cross button in its title bar a second time.

GPS provides a mode where all windows are floating and the MDI area in the main window is invisible. You may want to use this if you rely on windows handling facilities supported by your system or window manager that are not available in GPS, for example if you want to have windows on various virtual desktops and your window manager supports this.

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

2.6 Moving Windows

Change the organization of windows at any time by selecting a notebook containing several editors or browsers and selecting one of the *Split* entries in the *Window* menu.

You can also drag and drop the window within GPS. Select an item to drag by selecting the notebook tab. In that case, you can also reorder the windows within the notebook: select the tab, then start moving left or right to the window's new position. Your mouse must remain within the tab area or GPS will drop the window into another notebook.

Here are the various places where you can drop a window:

- Inside the MDI

  While the mouse button is pressed, the target area is highlighted and shows where the window would be put if you release the mouse button. The background color of the highlight indicates whether the window will be preserved or not when changing perspectives (for example, when starting a debug session). You can drag a window to one side of a notebook to split that notebook.

  If you drop a window all the way on a side of the area, the window will occupy the full width (or height) of the area.

  GPS will however restrict where windows can be placed: editors and most browsers, for instance, must go into the central area (the part that stays common when switching perspectives), whereas other views must stay on the sides (left, right, bottom or top) of that central area. The color of the highlight during a move (blue or brown) will indicate where the window can be dropped.

- System window

  If you drop a window outside of GPS (for example, on the background of your screen), GPS floats the window.

Keeping the `shift` key pressed while dropping the window results in a copy operation instead of a simple move, if possible. For example, if you drop an editor, a new view of the same editor is created, resulting in two views: the original one at its initial location and a second at the new location.

If you keep the `control` key pressed while dropping the window, all the windows in the same notebook are moved, instead of just the one you selected. This is the fastest way to move a group of windows to a new location.

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.

You can switch to a different perspective for different types of activities you want to perform (such as debugging or version control operations). For example, when using the debugger, the default perspective consists of windows containing the call stack, data window, and the debugger console, each at the location you have set. When you start the debugger again, you do not have to reopen these windows.

Each perspective has a name. Switch perspectives by selecting the *Window* → *Perspectives* menu. Create a new perspective by selecting the *Window* → *Perspectives* → *Create New* menu.

The most convenient way to change perspective, though, is to simply click on the button to the right of the main toolbar. By default, it shows the label "Default", which is the name of the default perspective. Selecting any item in the popup window will switch to that perspective.

GPS sometimes automatically changes perspectives. For example, if you start a debugger, it switches to the perspective called *Debug* if one exists. When the debugger terminates, you are switched back to the *Default* perspective, if one exists.

When you leave a perspective, GPS automatically saves its contents (including which windows are opened and their location) so when you return to the same perspective you see the same layout.

When GPS exits, it saves the layout of all perspectives to a file `perspectives6.xml` so it can restore them when you restart GPS. This behavior is controlled by the *General* → *Save desktop on exit* preference, which you can disable.

One difficulty in working with perspectives is knowing which windows are preserved when you switch to another perspective and which are hidden. To help you determine this, there's a central area where you can find all preserved windows. It usually only contains editors (including those that you have split side by side). If you drop another window on top or to one side of an editor, that window is preserved when changing perspectives unless it is already in the new perspective. The color of the highlight appearing on the screen while you drag tells you whether the window (if dropped at the current location) will be visible or hidden in other perspectives.

CHAPTER

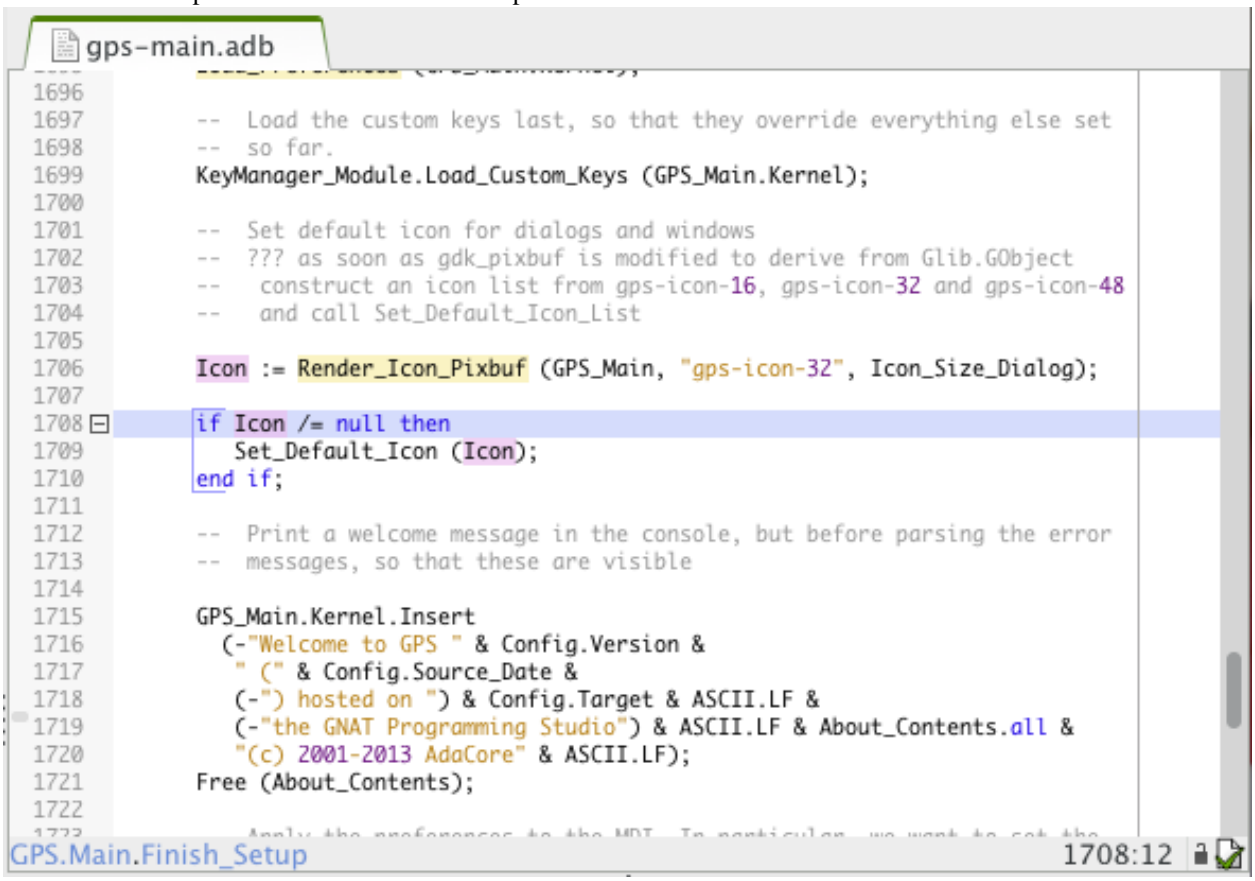
THREE

EDITING FILES

3.1 General Information

Source editing is one of the central parts of GPS. It allows access to many other functionalities, including extended source navigation and source analysis tools. You can have as many editor windows as you need. Each editor window receives annotations from other components in GPS, such as a debugger.

We use the term “pointer” to refer to the mouse pointer and “cursor” to refer to the text cursor.



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

Multi cursors

> You are not limited to edition via a single cursor in GPS. You can create multiple cursors that will all forward the text actions that you enter via your keyboard. This allows you to automate simple repetitive actions, in a similar way to what you would do with text macros, but in a simpler fashion.

Most of the text actions described in this documentation will be handled transparently by multi cursors, so you can delete several words at once, or select several pieces of text at once, for example.

At any time during edition with multiple cursors, you can press `Escape` to remove every cursor but the main one, so that you are back to single cursor edition. Using the mouse to move the cursor will have the same effect.

Title bar

Displays the full name of the file including path information in the title bar of the editor window.

Line number information

Located to the left of the source editor, Line numbers can be disabled using the *Editor → Display line numbers* preference. This area also displays additional information in some cases, such as the current line of execution when debugging or VCS annotations.

Scrollbar

Located to the right of the editor, this allows scrolling through the source file. The highlighted area of the scrollbar corresponds to the visible portion of the file. While you are scrolling, the editor displays a tooltip showing the file, line number, and subprogram corresponding to the center of the visible portion.

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*. Use the *Editor → Speed column policy* preference to control the display of this area. It can sometimes be convenient to keep it visible at all times (to avoid resizing the editors when new information becomes available) or to hide it automatically when not needed to save space on the screen.

Status bar

Gives information about the file. It is divided in two sections, one each on the left and 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 *Editor → Display subprogram names* preference controls this display.
- The right section contains multiple items:
  - The box displays the position of the cursor in the file as a line and column number. When you have made a selection in the editor, this area also displays the size of the selection (number of lines and characters).
  - Next to the box is an icon showing whether the file is writable or readonly. Change this state by clicking on the icon, which toggles between *Writable* and *Read Only*. This does not change the permissions of the file on disk: it only changes the writability of the view in the source editor.

    When you try to save a readonly file, GPS asks for confirmation, and if possible, saves the file, keeping its readonly state.
  - If the file is maintained under version control and version control is supported and enabled in GPS, the next icon shows VCS information for the file: the VCS kind (e.g. CVS or subversion) followed by the revision number and, if available, the file's status.

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 plugins. Most languages supported by GPS provide syntax highlighting in the editor.

Automatic indentation

When enabled, lines are automatically indented each time you press the `Enter` key or the indentation key, which, by default, is `Tab`. Change it in the key manager dialog. See *The Key Shortcuts Editor*.

If you have selected a list of lines when you press the indentation key, GPS indents all the lines.

Tooltips

When you place the pointer over a word in the source editor, GPS displays a small window if there is relevant contextual information to display about that word. The type of information displayed depends on the current state of GPS.

In normal mode, the editor displays the entity kind and location of the declaration when this information is available, i.e., 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 editor displays documentation for the entity, if available. This is the block of comments immediately before or after the entity's declaration (without any intervening blank lines). For example, the editor displays the following documentation for Ada:

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

B : Integer;
--  A comment for B

C : Integer;

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

When comments appear both before and after the entity, GPS choses the one given by the preference *Documentation → Leading documentation*. In debugging mode, the editor shows the value of the variable under the pointer if the variable is known to the debugger.

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.

Text completion is useful when editing a file using the same words repeatedly where it provides automatic word completion. When you type the `Ctrl-/` key combination (customizable through the key manager dialog) after a partial word, GPS inserts the next potential completion. Typing this key again cycles through the list of potential completions. GPS searches for text completions in all currently open files.

Delimiter highlighting

When the cursor is placed before an opening delimiter or after a closing delimiter, GPS highlights both delimiters. The following characters are considered delimiters: ()[]{}. Disable highlighting of delimiters with the preference *Editor → Highlight delimiters*.

Jump to a corresponding delimiter by invoking the *jump to matching delimiter* action (which can be bound to a key in the key shortcuts editor). Invoking this action a second time returns the cursor to its original position.

Current line highlighting

Configure the editor to highlight the current line with a specified color (see the preference *Editor → Fonts & Colors → Current line color*).

Current block highlighting

If the preference *Editor → Block highlighting* is enabled, GPS highlights the current block of code, e.g. the current **`begin...end`** block or loop statement, by placing a vertical bar to its left.

Block highlighting also takes into account the changes made in your source code and is recomputed to determine the current block when needed. This capability is currently implemented for the Ada, C, and C++ languages.

Block folding

When the preference *Editor → Block folding* is enabled, GPS displays – icons on the left side corresponding to the beginning of each block. If you click on one of these icons, all lines corresponding to this block are hidden except the first. Like block highlighting, these icons are recomputed automatically when you modify your sources.

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

Auto save

GPS will by default periodically save your work in temporary files. This can be configured via the *Edit → Preferences* dialog).

Automatic highlighting of entities

When the pointer is positioned on an entity in the source editor, GPS will highlight all references to this entity in the current editor. When the pointer is moved 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.

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

GPS also integrates with existing third party editors such as **`emacs`** or **`vi`**. See *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-2-0`	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 to the end of the current word.
`Ctrl-backspace`	Delete to the beginning of the current word.
`Shift-Alt-down`	Add a cursor to the current location and go down one line
`Shift-Alt-up`	Add a cursor to the current location and go up one line
`Ctrl-Alt-N`	jump the main cursor to the next occurence of the selection
`Shift-Ctrl-N`	Add a cursor selecting the current selection and jump the main cursor to the next occurence of the selection

3.3 Menu Items

This section describes the main menus that give access to extended functionality related to source editing.

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 asks you to enter the name of the file. If you have started typing Ada code, GPS tries to guess a name for the new file based on the first main entity in the editor and the current naming scheme.

- *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 different parts of the same file, for example a function spec and its body.

  You can also create a new view by holding the `shift` key down while dropping the editor (see *Moving Windows*). This second method is better because you can specify where you want to put the new view. The default

when using the menu puts the new view on top of the current editor.

- *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...*

  Move the focus to the *The omni-search* field, where you can immediately start typing part of the file name you want 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*. If you have access to a remote host file system, you can specify a file which can be edited in GPS. When you press the *save* button or menu item, the file is saved on the remote host.

  See *Using GPS for Remote Development* for a more efficient way to work locally on remote files.

- *File → Recent*

  Open a submenu containing a list of the ten most recent files opened in GPS.

- *File → Save*

  Save the file corresponding to current source editor, if there are changes.

- *File → Save As...*

  Save the current file under a different name, using the file selector dialog. See *The File Selector*.

- *File → Save More*

  Give access to additional save capabilities:

  - *File → Save More → All*

    Save all items, including projects.

  - *File → Save More →Desktop*

    Save the desktop to a file. The desktop includes information about files and graphs and their window sizes and positions in GPS. One desktop is saved per top level project so that when you reload the same project you get back to the same state you were in when you left GPS. If you load a different project, either another desktop or the default desktop is loaded. Request GPS to automatically save this desktop when you quit with the preference *General → Save Desktop On Exit*.

- *File → Change Directory...*

  Open a directory selection dialog that lets you change the current working directory.

- *File → Locations*

  This submenu gives access to functionalities related to the *Locations* window.

  - *File → Locations → Export Locations to Editor*

    List the contents of the *Locations* view in an editor.

- *File → Print*

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

  On Windows, if no command is specified in the preferences, GPS displays the standard Windows print dialog box, allowing you 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. 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 at the current cursor position.

- *Edit → Paste previous*

  GPS stores a list of all the text that was previously copied to the clipboard through the use of *Copy* or *Cut*.

  By default, if you press *Paste*, the newest text will be inserted at the cursor's current position. If you pres *Paste Previous* (one or more times) immediately after that, you can instead paste the text that was copied to the clipboard the previous time.

  For example, if you use *Edit → Copy* to copy the text `First`, then copy the text `Second`, select *Edit → Paste* to insert `Second` at the current cursor position. If you then select *Edit → Paste Previous*, `Second` is replaced by `First`. When reaching the end of this list, GPS starts from the beginning and again inserts the text that was copied to the clipboard most recently.

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

  For more information, see *The Clipboard view*.

- *Edit → Undo*

  Undo previous insertion or deletion in the current editor.

- *Edit → Redo*

  Redo previous insertion or 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 behavior depends on whether or not there is a current selection.

  If there is no selection, the set of lines modified begins with the current line and includes all adjacent lines that have at least one digit in the same column as the cursor. In the following example, 'I' marks the place where the cursor starts:

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

Only the first two lines are modified and 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 of each line that had digits in the same column of the first line are modified. Starting from the original example above, if you select all three lines, the replacement becomes:

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

Only the fifth and sixth columns are modified since only those columns contained digits in the first line.

This feature assumes you selected a relevant set of lines. But it is designed most specifically for modifying blank parts of lines. For example, if you start with:

```
AAA 1
BBB
CCC
```

it becomes:

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

- *Edit → Select all*

  Select the entire contents of the current source editor.

- *Edit → Insert File...*

  Open a file selection dialog and insert the contents of that file in the current source editor at the current cursor position.

- *Edit → Insert Shell Output...*

  Open an input window at the bottom of the GPS window where you can specify any external command. If the command succeeds, the output of the command is inserted at the current cursor position, or, 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. See *Edit → Preferences* for preferences related to source formatting.

- *Edit → Smart completion*

  Complete the identifier prefix under the cursor and list the results in a pop-up window. When used with Ada sources, this takes 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, you must have the smart completion enabled, which causes the computation of the entity database at GPS startup.

Support for C and C++ is not as powerful as the support for Ada since it relies completely on the cross-reference information files generated by the compiler, does not take 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, you must enable the smart completion preference and build your C/C++ application with **`-fdump-xref`**.

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

Typing additional letters reduces the range of possibilities, as long as possibilities remain. Once you have selected the expected completion, confirm it by pressing `Enter`. Typing any character which cannot be used in identifiers, including control characters, also confirms the current selection.

GPS can also automatically complete subprogram parameters or dotted notation for child and nested packages. For example, if you type:

```
with Ada.
```

a smart completion window appears, listing all child and nested packages of Ada. You can configure the time interval after which the completion window appears in the preferences dialog.

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

```
with Ada.Text
```

Pressing the completion key offers you Text_IO.

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

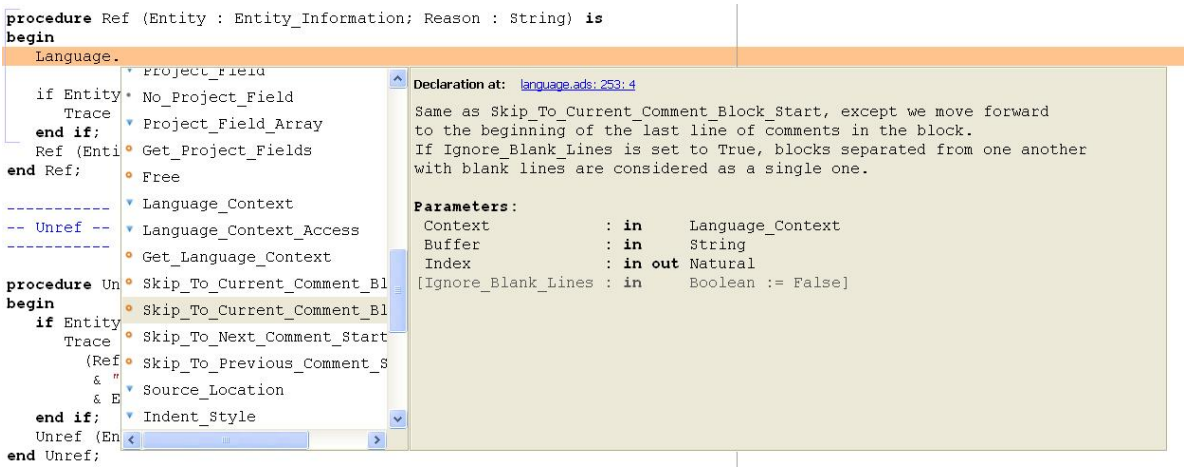
```
 declare
   type R is record
      Field1 : Integer;
      Field2 : Integer;
   end record;

   V : R;
begin
   V.
```

Completing V. proposes Field1 and Field2.

Smart completion also lists the possible parameters of a call you are 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 proposes completing with the named parameters "B =>", "C =>" or completing with all the remaining parameters, in this case "B =>, C =>)".



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 cross-reference information generated by the compiler. Therefore, GPS has no knowledge of 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 and by special characters such as '(', '->', or the C++ operator '::' and may propose too many candidates since it does not have knowlege of the C/C++ syntax context. Typing new letters reduces the range of possibilities, as long as possibilitites remain.
- Smart completion of subprogram parameters, fields and dotted notation are not yet available 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 at the cursor. This command cycles through all identifiers starting with the specified 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*

    Make the current selection or line into a comment based on the current programming language syntax.

  - *Edit → Selection → Uncomment lines*

    Remove the comment delimiters from the current selection or line.

- *Edit → Selection → Refill*

  Rearrange line breaks in the selection or current line so that line lengths do not exceed the maximum length, as set in the *Right margin* preference.

- *Edit → Selection → Sort*

  Sort the selected lines alphabetically. This is particularly useful when editing files that are not source code or for specific parts of code, such as **`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 is passed to the current selection as input. If the command succeeds, the selection is replaced by the output of the command.

- *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 left.

- *Edit → Fold all blocks*

  Collapse all blocks in the current file.

- *Edit → Unfold all blocks*

  Uncollapse all blocks in the current file.

- *Edit → Create bookmark*

  Creates a new *Bookmark* at cursor position. For more information, see *The Bookmarks view*.

- *Edit → Pretty Print*

  Pretty-print the current source editor by calling the external tool **`gnatpp`**. Specify **`gnatpp`** switches in the switch editor. See *The Switches Editor*.

- *Edit → Generate Body*

  Generate an Ada body stub for the current source editor by calling the external tool **`gnatstub`**.

- *Edit → Edit with external editor*

  See *Using an External Editor*.

- *Edit → Aliases*

  Display the Aliases editor. See *Defining text aliases*.

- *Edit → Preferences*

  Show the preferences dialog.

3.4 Rectangles

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

Select a rectangle using the standard selection mechanism. Either use the mouse to highlight the proper region, use `shift` and the cursor keys to extend the selection, or use the Emacs selection (with the mark and the current cursor location) if you have activated the Emacs key themes in the Key Shortcuts editor (*The Key Shortcuts Editor*).

Visually, a selected rectangle appears exactly the same as the standard selection. In particular, the characters after the last column on each line are also highlighted. Whether a selection is interpreted as full text or a rectangle depends on the entry you use to manipulate the selection.

If you use one of the entries from the *Edit → Rectangles* menu, the rectangle extends from the top-left corner to the bottom-right corner. All characters to the right of the right-most column, although highlighted, are not considered part of the rectangle.

Consider for example the following text:

```
package A is
   procedure P;

   procedure Q;
end A;
```

and assume you have selected from the character "p" in "procedure P" down to the character "c" in "procedure Q".

You can then use one of the following entries (either from the menu or key shortcuts assigned to them via the usual *Edit → Key shortcuts* menu).

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

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

  ```
  package A is
     edure P;

     edure Q;
  end A;
  ```

- *Edit → Rectangles → Copy*

  Copies the contents of the rectangle into the clipboard without affecting the current editor.

- *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 was initially in) and existing text in the editor lines is shifted to the right. If, for example, you now place the cursor in the first column of the second line and paste, you end up with:

  ```
  package A is
  proc   edure P;

  proc   edure Q;
  end A;
  ```

- *Edit → Rectangles → Clear*

  Replaces the contents of the selected rectangle with spaces. If you start from our initial example, you end up with the following. Note the difference between this and *Edit → Rectangles → Delete* menu:

```
package A is
       edure P;

       edure 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 between this and the *Edit → Rectangles → Clear* menu:

```
package A is
       procedure P;

       procedure Q;
end A;
```

- *Edit → Rectangles → Replace With Text*

  Similar to *Edit → Rectangles → Clear* but the rectangle is replaced with user-defined text. The lines are shifted left or right if the inserted text is shorter (respectively, longer) than the width of the rectangle. This command affects lines that are empty in the initial rectangle. If, for example, you replace our initial rectangle with the text "TMP", you end up with the following. Note that the character "c" has disappeared, since "TMP" is shorter than our rectangle width (4 characters):

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

- *Edit → Rectangles → Insert Text*

  Inserts text to the left of the rectangle on each line. The following example inserts `TMP`. Note the difference between this command and *Edit → Rectangles → Replace With Text*. This command also inserts the text on lines that are empty in the initial rectangle:

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

- *Edit → Rectangles → Sort*

  Sorts the selected lines according to the key which starts and ends on the 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, the lines are sorted as follows:

```
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 you want to repeat the action of pressing a single key, 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), entering the number of times you wish to repeat, and then pressing the key whose action you want to repeat.

  For example, the sequence `control-u 79 -` inserts 79 characters of '-' in the current editor. This is often useful to insert separators.

  If you are using the Emacs mode (see *Edit → Preferences → Key Shortcuts* menu), you can also use the sequence `control-u 30 control-k` to delete 30 lines.

- Recording macros

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

  First, tell GPS that it should start recording the keys you are pressing via the *Tools → Macros → Start Keyboard Macro* menu. As its name indicates, this only records keyboard events, not mouse events. GPS keeps recording the events until you select the *Tools → Macros → Stop Macro* menu.

  In Emacs mode, macro actions are bound to `control-x (`, `control-x )` and `control-x e` key shortcuts. For example, 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, for example, the right button on your mouse) on a source file, you get access to a number of entries, which are displayed or hidden depending on the current context.

These menu entries include the following categories:

Source Navigation

See *Contextual Menus for Source Navigation*.

Dependencies

See *The Dependency Browser*.

Entity browsing

See *The Entity browser*.

Project view

See *The Project view*.

Debugger

See *Using the Source Editor when Debugging*.

Case exceptions

See *Handling of casing*.

Refactoring

See *Refactoring*.

In addition, an entry *Properties...* is always visible in this contextual menu. When you select it, a dialog allows you to override the language or the character set used for the file. This is useful when opening a file that does not belong to the current project but where you want to benefit from the syntax highlighting, which depends on knowing the file's language.

Do not override the language for source files belonging to the current project. Instead, use the *Project* → *Edit Project Properties* menu and change the naming scheme as appropriate. This provides better consistency between GPS and the compiler in the way they manipulate the file.

3.7 Handling of casing

GPS maintains a dictionary of identifiers and a corresponding casing that are used by all case-insensitive languages. When editing or reformatting a buffer for such a language, the dictionary is checked first. If GPS finds an entry for a word or a substring of a word, it is used; otherwise the specified default casing for keywords or identifiers is used. A substring is defined as a part of the word separated by underscores.

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

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

- *Casing* → *Lower *entity**

  Set the selected entity to be in lower case.

- *Casing* → *Upper *entity**

  Set the selected entity to be in upper case.

- *Casing* → *Mixed *entity**

  Set the selected entity to be in mixed case (the first letter and letters before an underscore are in upper case and all other letters are in lower case).

- *Casing* → *Smart Mixed *entity**

  Set the selected entity as smart mixed case, which is the same as above except that upper case letters are kept unchanged.

- *Casing → Add exception for *entity**

  Add the current entity into the dictionary.

- *Casing → Remove exception for *entity**

  Remove the current entity from the dictionary.

To add or remove a substring from the dictionary, first select the substring in the editor. Then, the last two contextual menu entries will be:

- *Casing → Add substring exception for *str**

  Add the selected substring into the dictionary.

- *Casing → Remove substring exception for *str**

  Remove the selected substring from the dictionary.

3.8 Refactoring

GPS includes basic facilities to refactor your code. "Refactoring" is the term used to describe manipulation of source code that does not affect the behavior of the code but helps reorganize it to make it more readable, more extendable, or make other similar improvements. Refactoring techniques are generally things that programmers have done by hand, but which can be done faster and more securely when done automatically by a tool.

A basic recommendation when you refactor your code is to recompile and test your application regularly to make sure each small modification you made did not change the behavior of your application. This is particularly true with GPS, since it relies on the cross-references information generated by the compiler. If some source files have not been recompiled recently, GPS prints warning messages indicating that the operation might be dangerous and/or only partially performed.

One of the reference books used in the choice of refactoring methods for GPS 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 opens a dialog asking for the new name of the entity. GPS renames all instances of the entity in your application, including the definition of the entity, its body, and all calls to it. No comments are updated so you should probably manually check that the comment for the entity still applies.

GPS handles primitive operations by also renaming the operations it overrides or that override it, so any dispatching call to that operation is also renamed, allowing the application to continue to work properly. If you are renaming a parameter to a subprogram, GPS also renames parameters with the same name in overriding or overridden subprograms.

You can specify the behavior for read-only files: by default, GPS wil not do any refactoring in these files and instead displays a dialog listing them. However, you can choose to make them writable just as if you had clicked on the *Read-Only* button in the status bar of the editor and 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 displays a *Refactoring → Name parameters* contextual menu, which replaces all unnamed parameters by named parameters, for example:

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

3.8.3 Extract Subprogram

This refactoring moves some code into a separate subprogram to simplify the original subprogram by moving part of its code elsewhere. Here is an example from the "Refactoring" book. The refactoring takes place in the body of the package `pkg.adb`, but the spec is needed so you can compile the source code (a preliminary, but mandatory, step 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;
```

An initial implementation for this is the following:

```
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;
```

Suppose we feel 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.

First, we move 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 *Refactoring → Extract Subprogram* contextual menu. GPS removes those lines from the subprogram **Print_Owing** and creates a new procedure **Print_Banner** (the name is specified by the user; GPS does not try to guess a name). 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 the relevant 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, 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 also automatically decides whether to create `in`, `out` or `in out` parameters. If there is a single `out` parameter, GPS automatically creates a function instead of a procedure.

GPS uses the same name for the local variable for the parameters. Often, it makes sense to recompile the new version of the source and apply the *Refactoring → Rename Entity* refactoring to have more specific names for the parameters, or the *Refactoring → Name Parameters* refactoring so that calls 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 balance. When this code is moved, the variables **E** and **Each** become dead in **Print_Owing** and are moved into the new subprogram (which we call **Get_Outstanding**). The initial selection should include the blank lines before and after the code to keep the resulting **Print_Owing** simpler. GPS automatically ignores those blank lines. Here is the result of that last refactoring

```
... 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;
```

The final version of **Print_Owing** is not perfect. For example, passing the initial value 0 to **Get_Outstanding** is useless and, in fact, it should probably be a function with no parameter. But GPS already saves a lot of time and manipulation even given these imperfections.

Finally, a word of caution: this refactoring does not check that you are starting with valid input. For example, 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 *Editor → External editor* preference.

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 providing a full integration in GPS, since any extended lisp command can be sent to the Emacs server.

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

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

  See http://www.hpl.hp.com/personal/ange/gnuserv/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 reuses an existing Emacs sessions, so it is extremely fast to open a file.

- **emacs**

  This clients 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. The only reason to use this external editor is if your system does not support **emacsclient**.

- **vim**

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

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

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

- **vi**

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

- **custom**

  Specify any external editor by choosing this entry. Specify the complete command line used to call the editor in the *Editor → Custom editor command* preference.

- **none**

  No external editor is used and the contextual menus do not appear.

In the cases that require an Emacs server, the project file currently used in GPS is 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 a new one is spawned by GPS with the appropriate project.

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

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

3.10 Using the Clipboard

This section is of interest to 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 to the current selection.

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

By default, GPS overrides the X mechanism. To prevent this, add the following line:

```
OVERRIDE_MIDDLE_CLICK_PASTE = no
```

to your `traces.cfg` file (typically in `~/.gps/`). Note 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.

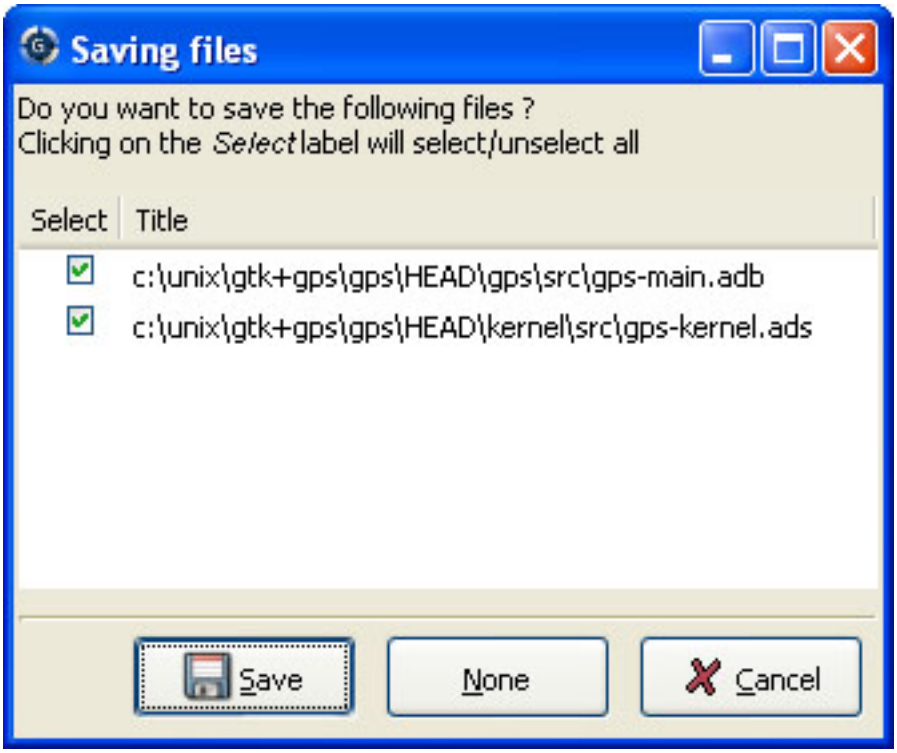
3.11 Saving Files

After you have finished editing your files, you need to save them. Do so by selecting the *File* → *Save* menu, which saves the currently selected file.

Use the *File* → *Save As...* menu if you want to save the file with another name or in another directory.

If you have multiple files to save, use the *File* → *Save More* → *All* menu, which opens a dialog listing all the currently modified editors. Select which ones should be saved and click on *Save* to save those editors.

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 the external command sees your changes. If the preference is enabled, editors are saved automatically.



Conveniently select or unselect all the files at once by clicking on the title of the first column (labeled *Select*). This toggles the selection status of all files.

If you press *Cancel* instead of *Save*, nothing is saved and the action that displayed this dialog is also canceled. Such actions can be, for example, starting a compilation command, a VCS operation, or quitting GPS with unsaved files.

3.12 Printing Files

GPS lets you configure how printing is performed, via its *External Commands/Print command* preference.

This program is required for Unix systems, and is set to **`lp`** by default. Other popular choices include **`a2ps`** which provides pretty-printing and syntax highlighting.

On Windows, this preference is optional and the preference is empty by default since GPS provides built-in printing. If you specify an external tool, such as the **`PrintFile`** freeware utility available from http://www.lerup.com/printfile/descr.html, GPS uses that.

CHAPTER

FOUR

SOURCE NAVIGATION

4.1 Support for Cross-References

GPS provides cross-reference navigation for program entities defined in your application such as types, procedures, functions, and variables. This support relies on compiler-generated cross-reference information, so you need to compile your project before being able to navigate within it. Similarly, if you have modified your sources, you need to rebuild and recompute the xref information it you want your changes to be taken into account by GPS.

Here is language specific information about source navigation:

Ada

By default, GPS uses the GNAT compiler to generate the cross-reference information it needs. However, if you are using the **-gnatD** or **-gnatx** switches, no cross reference information is available to GPS.

If you need to navigate through sources that do not compile (such as after modifications or while porting or initially developing an application), GNAT generates partial cross-reference information if you specify the **-gnatQ** switch. Using this along with the **-k** switch of **gnatmake** generates as much relevant information from your non-compilable sources as possible.

Sometimes GPS cannot find the external files (called `ALI files`) containing the cross-reference information. Most likely, this is either because you have not compiled your sources yet or because the sources changed since the `ALI files` were generated. Another possibility is that you have not included the object directories that contain the `ALI files` in the project.

In addition, GPS cannot automatically handle one special case, when you have separate units whose file names have been crunched by the **gnatkr** command.

C/C++

You need to use the GCC C and C++ compilers that come with GNAT to generate the cross-references information needed by GPS and to call them with the **-fdump-xref** switch, so you need to first add that switch to your project's switches for C and C++ sources and compile your application before you browse through the cross-references. If your sources have been modified, recompile the modified files.

4.1.1 Ada cross-reference heuristics

GPS provides basic navigation support for Ada, C, and C++ sources even in the absence of information coming from the compiler by using a built-in parser, parsing the files both at startup and when they are modified. This provides basic navigation in simple cases.

In this mode, GPS can navigate to an entity body from the declaration and vice versa. For other references, GPS can navigate to the declaration only if the heuristics provides the necessary information without ambiguity, which may not be the case with overloaded entities.

GPS also uses this parser to provide the Ada outline view, code completion and entity view, but these heuristics are not used in global reference searching operations or to generate call graphs.

4.1.2 The cross-reference database

GPS parses the cross-reference information generated by the compiler (the `.ali` and `.gli`) files into one or several **`sqlite`** databases (e.g: if your project uses Ada and C). These database files can become quite large and should preferably be on a fast local disk.

By default, GPS places these database files in the object directory of the currently-loaded root project, or, if specified, in the directory designated by the relative or absolute path given by the *Artifacts_Dir* attribute of the *IDE* package of your project file:

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

   package IDE is
      for Artifacts_Dir use "artifacts";
      --  All the artifacts generated by GPS (including the xref datatabases)
      --  will be put in the /home/user1/work/artifacts/ directory.
      --
      --  We could also have specified an absolute path here
      --  (e.g: for Artifacts_Dir use "/home/user1/work/artifacts/").
   end IDE;
end Default;
```

If you want to have more advanced control regarding the naming of the Ada cross-references database file, you can use the *Xref_Database* in the *IDE* package of your project file, either as an absolute path or a path relative to the location of the project file. We recommend this path to be specific to each use, 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 drawback in altering the default location is that **`gprclean`** will not remove these database files when you clean your project. But it might speed up GPS if your project is not on a fast local disk and you can put the databases there.

WARNING: You should not store this file in a directory that is accessed via a network filesystem, like NFS, or Clearcase's MVFS. If your obj directory is on such a filesystem, be sure to specify a local directory for IDE'Artifacts_Dir project attribute or, if you project only uses Ada, a custom local file path for the IDE'Xref_Database project attribute.

4.1.3 Cross-references and partially compiled projects

The cross-reference engine works best when the cross-reference information generated by the compiler (the `.ali` files) is fully up to date.

If you start from such a state and then modify the spec or body of an Ada package and recompile only that file, any reference to entities declared in that spec in other packages might no longer be found (until you recompile those other packages, as **`gprbuild`** would).

This is because GPS has no way to know for sure whether an entity `Foo` in the spec is the same entity as before or is a new one with the same name. It uses an approximate algorithm where the references are only preserved if an entity with the same name remains at precisely the same location in the new version of the source. But if a blank line in the file will change the declaration line for all entities declared further in the file, so those will lose their references from other source files.

4.1.4 Cross-reference and GNAT runtime

By default, GPS does not parse the GNAT runtime files because there is a large number of them and doing so would significantly slow down GPS, while producing only a minimal gain for most users. However, the location of subprograms in those runtime files is available from the `.ali` files corresponding to the sources of your project.

From your own sources, you can navigate to one of the runtime files (for example, if you have a reference to `Put_Line()`, you will jump to its declaration in `a-textio.ads`). But you cannot perform cross-reference queries from a runtime file itself.

If you need this capability, enable the preference *Project/Cross References in Runtime Files*.

4.2 The Navigate Menu

- *Navigate → Find or Replace...*

  Open the find and replace dialog. See *Searching and Replacing*.

- *Navigate → Find Next*

  Find next occurrence of the current search. See *Searching and Replacing*.

- *Navigate → Find Previous*

  Find previous occurrence of the current search. See *Searching and Replacing*.

- *Navigate → Find All References*

  Find all the references to the current entity in the project. This is not a simple text search, but is based on the semantic information extracted from the sources. The result of the search is displayed in the *Location* view. See *The Locations View*.

- *Navigate → Goto declaration*

  Go to the declaration (spec) of the current entity. You can also access this entry through the editor's contextual menu. This requires the availability of cross-reference information. See *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. You can also access this entry through the editor's contextual menu. This requires the availability of cross-reference information. See *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 and jump to that line in the current source editor. This entry is also available by clicking on the location at the bottom of editors.

- *Navigate → Goto entity*

  Moves the focus to the *The omni-search* view. You can the enter the name (or part of the name) for any entity defined in your project. Clicking on one of the results takes you to its declaration.

- *Navigate → Goto file spec<->body*

  Open the corresponding spec file if the current edited file is a body file, or the body file otherwise. You can also access this entry through the editor's contextual menu. This requires support for cross-references.

- *Navigate → Start of statement*

  Move the cursor to the start of the current statement or the start of the enclosing statement if the cursor is already at the start of a statement.

- *Navigate → End of statement*

  Move the cursor to the end of the current statement or the end of the enclosing statement if the cursor position is already at the end of a statement.

- *Navigate → Previous subprogram*

  Move the cursor to the start of the previous procedure, function, task, protected record, or entry.

- *Navigate → Next subprogram*

  Move the cursor to the start of the next procedure, function, task, protected record, or entry.

- *Navigate → Previous tag*

  Go to previous tag or location. *The Locations View*.

- *Navigate → Next tag*

  Go to next tag or location. *The Locations View*.

- *Navigate → Back*

  Each time you use one of the navigation features in GPS, it stores the current location in a history. This entry allows you to navigate backward in the history, going 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 selected text, the contextual menu applies to the selection or entity. Most of these menus requires support for cross-references.

- *Goto declaration of *entity**

  Go to the declaration (spec) of *entity*.

- *Goto declarations of *entity**

  This entry appears when clicking on a dispatching subprogram call. In that case, GPS cannot know what subprogram will actually be called at run time, so it 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` plugin (enabled by default) which, given an object, lists all its primitive operations in a contextual menu so you can easily jump to them. See also the *References → Find References To...* contextual menu, which allows you to find all calls to a subprogram or one of its overriding subprograms.

- *Goto full declaration of *entity**

  This entry appears for a private or limited private types. Go to the full declaration (spec) of *entity*.

- *Goto type declaration of *entity**

  Go to the type declaration of *entity*.

- *Display type hierarchy for *entity**

  This entry appears for derived or access types. Put the type hierarchy for *entity* into the *Location* view.

- *Goto body of *entity**

  Go to the body (implementation of *entity*.) If *entity* is the declaration of an Ada subprogram imported from C, go to the location where the C function is defined.

- *Goto bodies of *entity**

  Similar to *Goto declarations of*, but applies to the bodies of entities.

- *Goto file spec/body*

  Open the corresponding spec file if the current edited file is a body file, or the body file otherwise. This entry 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 many references. See *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 correponding *Browsers/* submenu if you expect many references. See *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...*

  Similar to the entry above except you can select more precisely what kind of reference should be displayed. You can also specify the scope of the search and whether the context (or caller) at each reference should be displayed.

  The option *Include overriding and overridden operations* includes references to overridden or overriding entities. This is particularly useful if you need to know whether you can easily modify the profile of a primitive operation or method since you can see which other entities would also be changed. If you select only the *declaration* check box, you see the list of all related primitive operations.

  This dialog allows you to determine which entities are imported from a given file or unit. Click on any entity from that file (for example on the **`with`** line for Ada code) and select the *All entities imported from same file* toggle, which displays in the *Location* view the list of all entities imported from the same file.

Selecting the *Show context* option produces a list of all the references to these entities within the file. If it is not selected, 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 entry is only visible if you activated the plugin `methods.py` (the default) and when you click on a tagged type or an instance of a tagged type. It lists all the primitive operations or methods of that type, allowing you to jump to the declaration of any of these operations or methods.

- *Browsers → *Entity* calls*

  Open or raise the *Callgraph* browser on the specified entity and display all the subprograms called by it. See *Callgraph browser*.

- *Browsers → *Entity* calls (recursively)*

  Open or raise the *Callgraph* 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). See *Callgraph browser*.

- **Entity* is called by*

  Open or raise the *Callgraph* browser on the specified entity and display all the subprograms calling *entity*. See *Callgraph browser*.

- *Expanded code*

  Present for Ada files only. Generates a `.dg` file by calling the GNAT compiler with the *:index:command:-*gnatGL'' switch and displaying the expanded code. Use this when investigating low-level issues and tracing how your 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.

- *Open *filename**

  When you click on a filename (for example, a C **`#include`**, or an error message in a log file), this entry opens that file. If the file name is followed by `:` and a line number, the cursor points to that line.

4.4 Navigating with hyperlinks

When you press the `Control` key and start moving the mouse, entities in the editors under the pointer become hyperlinks and the form of the pointer changes.

Left-clicking on a reference to an entity opens a source editor on the declaration of the entity and left-clicking on an entity declaration opens an editor on the implementation of the entity. Left-clicking on the Ada declaration of a subprogram imported from C opens a source editor on the definition of the corresponding C entity. This capability requires support for cross-references.

Middle-clicking on either a reference to an entity or the declaration of an entity jumps to the implementation (or type completion) of the entity.

For efficiency, GPS may create hyperlinks for some entities which have no associated cross reference. In this case, clicking has no effect even though a hyperlink is displayed.

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

4.5 Highlighting dispatching calls

By default, GPS highlights dispatching calls in Ada and C++ source code via the `dispatching.py` plugin. Based on the cross-reference information, this plugin highlights (with a special color you can configure in the preferences dialog) all Ada dispatching calls or calls to virtual methods in C++. A dispatching call in Ada is a subprogram call where the actual subprogram called is not known until run time and is chosen based on the tag of the object.

Disable this highlighting (which may be slow if you have large sources) by using the *Edit* → *Preferences* → *Plugins* menu and disabling the `dispatching.py` plugin.

CHAPTER

FIVE

PROJECT HANDLING

The discussion of the *Project* view (see *The Project view*) gave a brief overview of what the projects are and the information they contain. This chapter provides more in-depth information and describes how you create and maintain projects.

5.1 Description of the Projects

5.1.1 Project files and GNAT tools

The projects used by GPS are the same as the ones used by GNAT: all command-line GNAT tools are project aware. Projects files are text (with the extension `.gpr`), which you can edit with any text editor or through GPS's interface. GPS can load any project file, even those you created or edited manually, and you can manually edit project files created by GPS. Most features of project files can be accessed without using GPS.

The detailed syntax and semantics of project files is fully described in the GNAT User's Guide and GNAT Reference Manual. Read these sections if you want to use the more advanced capabilities of project files that are not supported by GPS's graphical interface.

You usually will not need to edit project files manually, since GPS provides several graphical tools such as the project wizard (see *The Project Wizard*) and the properties editor (see *The Project Properties Editor*).

GPS does not preserve the layout or comments of projects you created manually after you have edited them in GPS. For example, multiple case statements in the project are merged into a single case statement. GPS needs to do this normalization to be able to preserve the previous semantics of the project in addition to supporting the new settings.

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

- absolute paths
- relative paths

  These paths, when used in a **`with`** line as described below, are relative to the location of the project containing the **`with`**.

- *ADA_PROJECT_PATH*

  If set, an environment variable containing a colon-separated (semicolon under Windows) list of directories to be searched for project files.

- *GPR_PROJECT_PATH*

  If set, an environment variable containing a colon-separated (semicolon under Windows) list of directories to be searched for project files.

- predefined project path

  The compiler internally defines a predefined project path in which standard libraries can be installed, for example XML/Ada.

5.1.2 Contents of project files

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

A project file can contain comments, which have the same format as in Ada: they start with "–" and extend to the end of the line. You can add comments when you edit the project file manually. GPS attempts to preserve them when you save the project through the menu, but this is not always possible. GPS is more likely to preserve them if the comments are put at the end of the line:

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

Often, one project file is not enough to describe a complex system. If so, you will create and use a project hierarchy, with a root project importing subprojects. Each project and subproject is responsible for its own set of sources (including compiling them with the appropriate switches and putting the resulting files in the correct directories).

Each project file contains the following information (see the GNAT User's Guide for the full list):

- `List of imported projects`

  When compiling sources from this project, the builder first makes sure it correctly recompiled all the imported projects and that they are up to date. This properly handles dependencies between source files.

  If one source file of project A depends on some source files from project B, B must be marked as imported by A. If this is not done, the compiler will complain that those source files cannot be found.

  Each source file name must be unique in the project hierarchy (i.e., a file cannot be under control of two different projects), ensuring that the file will be found no matter what project is managing it.

- `List of source directories`

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

- `Object directory`

  When sources of the project are compiled, the resulting object files are put in this directory. There must be exactly one object directory for each project. If you need to split the object files across multiple directories, you must create multiple projects importing each other.

  When sources from imported subprojects are recompiled, the resulting object files are put in the subproject's own object directory and not the parent's object directory.

- `Exec directory`

  When the object files are linked into an executable, this executable is put in the exec directory specified by this attribute. If it is omitted, the builder puts the executable into the object directory.

- `List of source files`

  Each project is responsible for managing its set of source files. These files can be written in any programming language, but the graphical interface supports only Ada, C, and C++.

  By default, these source files are found by taking all the files in the source directories that follow the naming scheme (see below) for each language. You can also edit the project file manually to provide an explicit list of source files.

This attribute cannot be modified graphically.

- `List of main units`

  The main units of a project (or main files in some languages) are the units containing the main subprogram of the application. The name of the file is generally related to the name of the executable.

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

- `Naming schemes`

  The naming scheme refers to the way files are named for each language used by your project. GPS uses this to choose the language to use when you open a source file and what tools to use to compile or otherwise manipulate a source file.

- `Embedded targets and cross environments`

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

  GPS also supports embedded targets such as VxWorks by specifying alternate names for the build and debug tools.

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

- `Tools`

  Project files provide a simple way of specifying the compiler and debugger commands to use.

- `Switches`

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

5.2 Supported Languages

Other information stored in the project file is the list of languages used by the project. GPS supports any language, each with a name you choose, but advanced support is only provided by default for some languages (Ada, C, and C++). You can specify other properties of the languages through customization files (see *Adding support for new languages*).

The graphical interface only allows you to choose languages currently known to GPS, either through built-in support or your customization files. Supporting a languages means syntax highlighting in the editor, and possibly the *Outline* view. Other languages have advanced cross-references facilities available. You can 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 to allow GPS to associate files with that language. For example, you could specify that all `.adb` files are Ada, all `.txt` files are standard text files, etc.

Only files that have an associated known language are displayed in the *Project* view and available for selection through the *File* → *Open From Project* menu. Similarly, only these files are shown in the Version Control System interface. It is important to properly set up your project to make these files conveniently available in GPS although you can still open any file through the *File* → *Open* menu.

If your project includes `README` files, or other text files, you should add "txt" as a language (the name is arbitrary) and ensure these files are associated with that language in the *Project* → *Properties...*.

5.3 Scenarios and Configuration Variables

You can further tailor the behavior of project by using scenarios.

You can specify the value of all attributes of a project except its list of imported projects based on the value of external variables, each of which comes from either the host computer environment or is specifically set in GPS. The interface to manipulate these scenarios is the *Scenario* view, which you display by selecting the menu *Tools* → *Views* → *Scenario* (*The Scenario View*). You may want to drop this window above the *Project* view so you can see both at the same time.

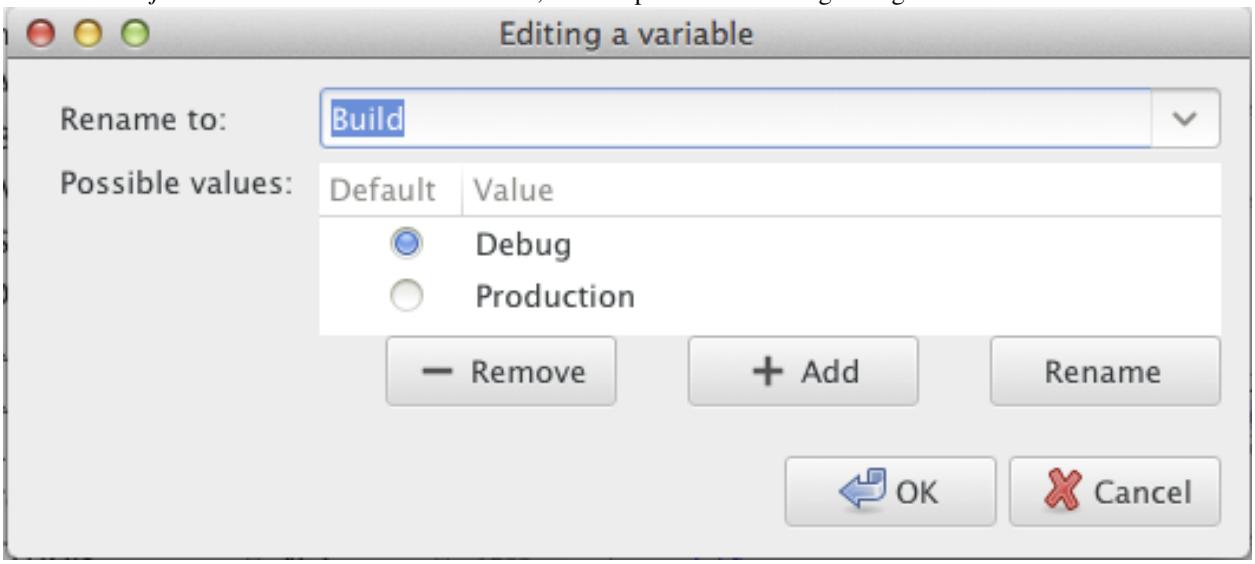
The *Scenario* view allows you to select new values for the scenario variables defined in your project and thus dynamically change the view GPS has of your project and its source files.

For example, you can use this facility to compile all the sources either in debug mode (so 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 that scenario, most of the attributes (such as source directories and tools) remain the same, but compilation switches differ. You could also maintain a completely separate hierarchy of projects, but it is much more efficient to create a new configuration variable and edit the switches for the appropriate scenario (see *The Project Properties Editor*).

There is one limitation on what GPS can do with scenario variables: although **`gnatmake`** and **`gprbuild`** can use scenario variables whose default value is something other than static string (for example, a concatenation or the value of another scenario variable), GPS cannot edit such a project graphically, though such projects load correctly.

5.3.1 Creating new scenario variables

Create a new scenario variable through the contextual menu (right-click) in the *Project* or *Scenario* views themselves. Select the *Project* → *Add Scenario Variable* menu, which opens the following dialog:



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

- It is displayed in the *Scenario* view
- It is the name of the environment variable from which the initial value is read. When GPS starts, all configuration variables are initialized from the host computer environment, although you can later change their values inside GPS. Selecting a new value for the scenario variable does not change the 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`** you can also specify the value they should use for the scenario variable by using a command line switch, typically **`-X`**.

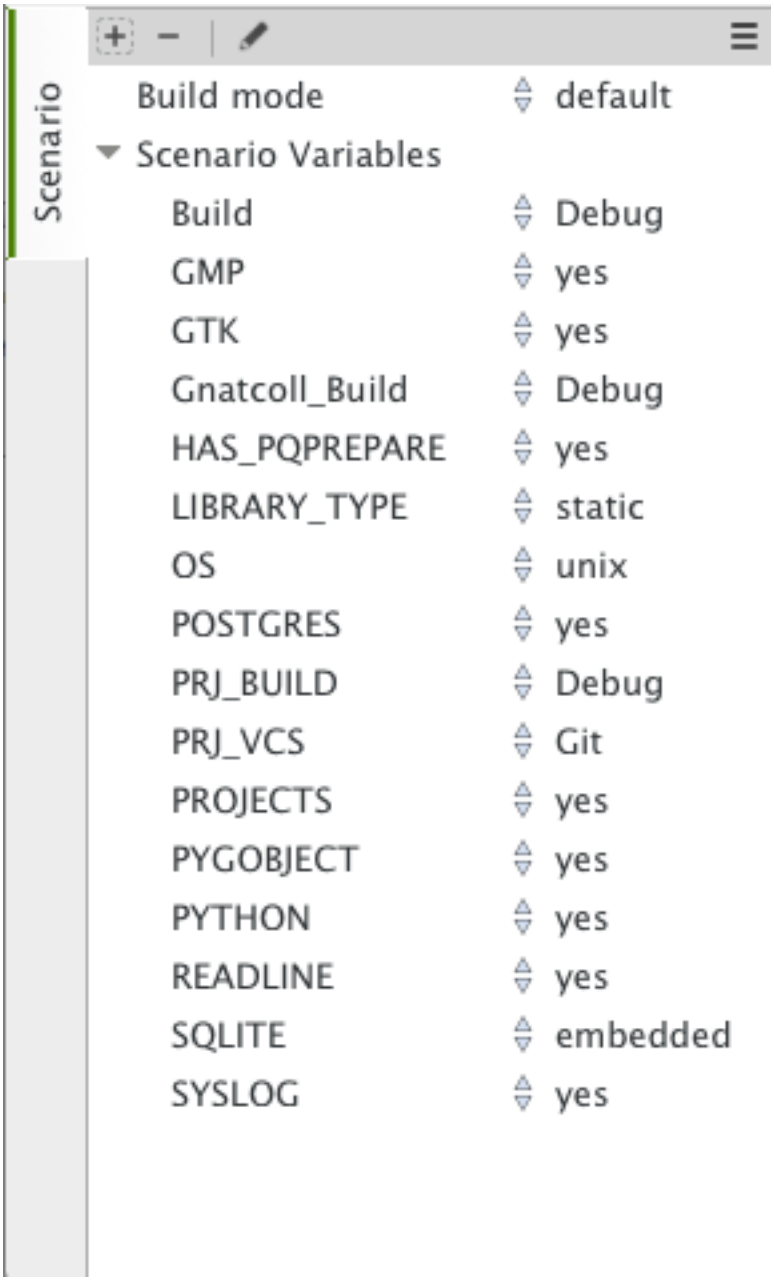
Click on the arrow on the right of the name area to display the list of all currently-defined environment variables. However, you can choose any variable; the environment variable need not exist when you start GPS.

The second area in this dialog is the list of possible values for this variable. GPS generates an error and will not load the project if you specify any other value. One of these values is the default (the one whose button in the *Default* column is selected). If the environment variable is not defined when GPS starts, it behaves as if it had this default value.

You can edit the list of possible values by right-clicking on the name of the variable and selecting either *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 contains something similar to:



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

As soon as a new value is selected, GPS recomputes the *Project* view (in case source directories, object directories or list of source files have changed). GPS also updates other items such as the list of executables in the *Compile*, *Run*, and *Debug* menus.

Because it can be time consuming and costly of system resources, GPS does not recompute the contents of the various browsers, such as the call graph and dependencies, for this updated project. You must explicitly request that they be updated if you want them recomputed.

Change the list of possible values for a configuration variable at any time by clicking on the *edit* button in the local toolbar. This pops up the same dialog used to create new variables, and also allows you to change the name of the scenario variable (the same name as the environment variable used to set the initial value of the scenario variable).

To remove a variable, select it and click the *remove* button in the local toolbar. GPS displays a confirmation dialog. When the variable is removed, GPS acts as if the variable always had the value it had when it was removed.

5.4 Extending Projects

5.4.1 Description of project extensions

Project files are designed to support large projects, with several hundred or even several thousand source files. In such contexts, one developer will generally work on a subset of the sources. Such a project may often take several hours to be fully compiled. Most developers do not need to have the full copy of the project compiled on their own machine.

However, it can still be useful to access other source files from the application. For example, a developer may need to find out whether a subprogram can be changed, and where it is currently called.

Such a setup can be achieved through project extensions. 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 those inherited from the original project.

When compiling such projects, the compiler puts the newly created project files in the extension project's directory and leaves the original directory untouched. As a result, the original project can be shared read-only among several developers (for example, the original project is often the result of a nightly build of the application).

5.4.2 Creating project extensions

The project wizard allows you to create extension projects. Select an empty directory (which is created if it does not exist), as well as a list of initial source files (new files can be added later). GPS copies the selected source files to the directory and creates a number of project files there. It then loads a new project, with the same properties as the previous one, except that some files are found in the new directory and object files resulting from the compilation are put into that directory instead of the object directory of the original project.

5.4.3 Adding files to project extensions

Once you load a project extension in GPS, most things are transparent to the extension. If you open a file through the *File* → *Open From Project* dialog, the files found in the local directory of the extension project are picked up first. Build actions create object files in the project extensions' directory, leaving the original project untouched.

You may want to work on a source file you did not put in the project extension when you created it. You could edit the file in the original project (provided, of course, you have write access to it). However, it is generally better to edit it in the context of the project extension, so the original project can be shared among developers. Do this by clicking the file in the *Project* view and selecting the *Add To Extending Project* menu. You will see a dialog asking whether you want GPS to copy the file to the project extension's directory. GPS may also create some new project files in that directory, if necessary, and automatically reload the project as needed. From that point on, if you use the menu *File* → *Open From Project*, GPS uses the file from the project extension. Open editors will still edit the same files they previously contained, so you should open the new file in them if needed.

5.5 Aggregate projects

Aggregate projects are a convenient way to group several independent projects into a single project that you can load in GPS. Using an aggregate project has several advantages:

- There is no restriction on duplicate names within aggregate sources and projects. There can be duplicate file names between the aggregate projects or duplicate projects. For example, if you have a project `liba.gpr` containing a library used by both `projectA.gpr` and `projectB.gpr`, you can still aggregate the latter two projects. A source file is also permitted to belong to both `projectA.gpr` and `projectB.gpr`.
- You can use **gprbuild** to build the main units of all aggregate projects with a single command.
- The aggregated project can contain attributes to setup your environment, in particular you can use `External` to set the value of the scenario variables and `Project_Path` to set the project path to be used to load the aggregated projects.

Here is a short example of an aggregate project:

```
aggregate project BuildAll is
    --  "liba.gpr" as described above, is automatically imported, but
    --  not aggregated so its main units are not build
    for Project_Files use ("projecta/projecta.gpr",
                           "projectb/projectb.gpr");

    --  Set environment variables
    for External ("BUILD") use "Debug";
end BuildAll;
```

GPS helps you use aggregate projects in the following ways:

- Since a source file can now belong to several projects, each editor is associated with a specific project. If the `common.ads` file is part of multiple projects, you may end up with two editors, one for `common.ads` in the context of `projectA.gpr`, and the other in the context of `projectB.gpr`. The project matters when doing cross-reference queries, since a *with C;* in `common.ads` could point to different files depending on which project owns that editor.

  To help with this, GPS shows the name of the project in the notebook tabs.
- The omni-search (at the top-right corner of the GPS window) may list the a file several times, once per each project that owns it. So you need to select the one you are interested in.
- After you perform a cross-reference (*Navigate* → *Goto declaration*), the newly opened editor automatically selects the proper project.

5.6 Disabling Editing of the Project File

You should generally consider project files part of the sources and put them under the control of a version control system. This will prevent accidental editing of the project files, either by you or someone else using the same GPS installation.

One way to prevent such accidents is to change the write permissions of the project files themselves. On Unix systems, you could also change the owner of the file. When GPS cannot write a project file, it reports an error to the user. However, the above does not prevent a user from trying to make changes 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 a special startup switch, typically by creating a short script that spawns GPS with these switches. Use the following command line:

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

This prevents the loading of the two GPS modules responsible for editing project files. However, this also has an impact on the Python functions that are exported by GPS and thus could break some plugins. Another possible

solution is to hide the corresponding project editing menus and contextual menus. You could do this by enabling the `prevent_project_edition.py` plugin via the *Edit* → *Preferences* → *Plugins* menu.

5.7 The Project Menu

The menu bar item *Project* contains several entries that act on the whole project hierarchy. To act on only a single project, use the contextual menu in the *Project* view.

GPS loads a single project hierarchy at any one time. Some of these entries apply to the currently selected project. Which project is considered currently selected depends on what window is currently active in GPS: if it is the *Project* view, the selected project is either the selected node (if a project) or its parent project (for a file or directory, for example). If the currently active window is an editor, the selected project is the one containing that file. If none of those are the case, it is the root project of the hierarchy.

These entries are:

- *Project* → *New...*

  Open the project template wizard, allowing you to create a new project using one of the project templates defined in GPS. See *Adding project templates*.

- *Project* → *Open*

  Open a file selection dialog, allowing any existing project to be loaded into GPS. The newly loaded project replaces the currently loaded project hierarchy.

- *Project* → *Recent*

  Switch back to the last project loaded into GPS.

- *Project* → *Properties...*

  Open the project properties dialog for the currently selected project.

- *Project* → *Save All*

  Save all the modified projects in the hierarchy.

- *Project* → *Edit File Switches*

  Open a new window in GPS listing all the source files for the currently selected project along with the switches used to compile them. See *The Switches Editor*.

- *Project* → *Reload project*

  Reload the project to take into account modifications done outside of GPS. In particular, take into account new files added to the source directories externally. If all modifications were made though GPS, you do not need to do this.

- *Project* → *Project View*

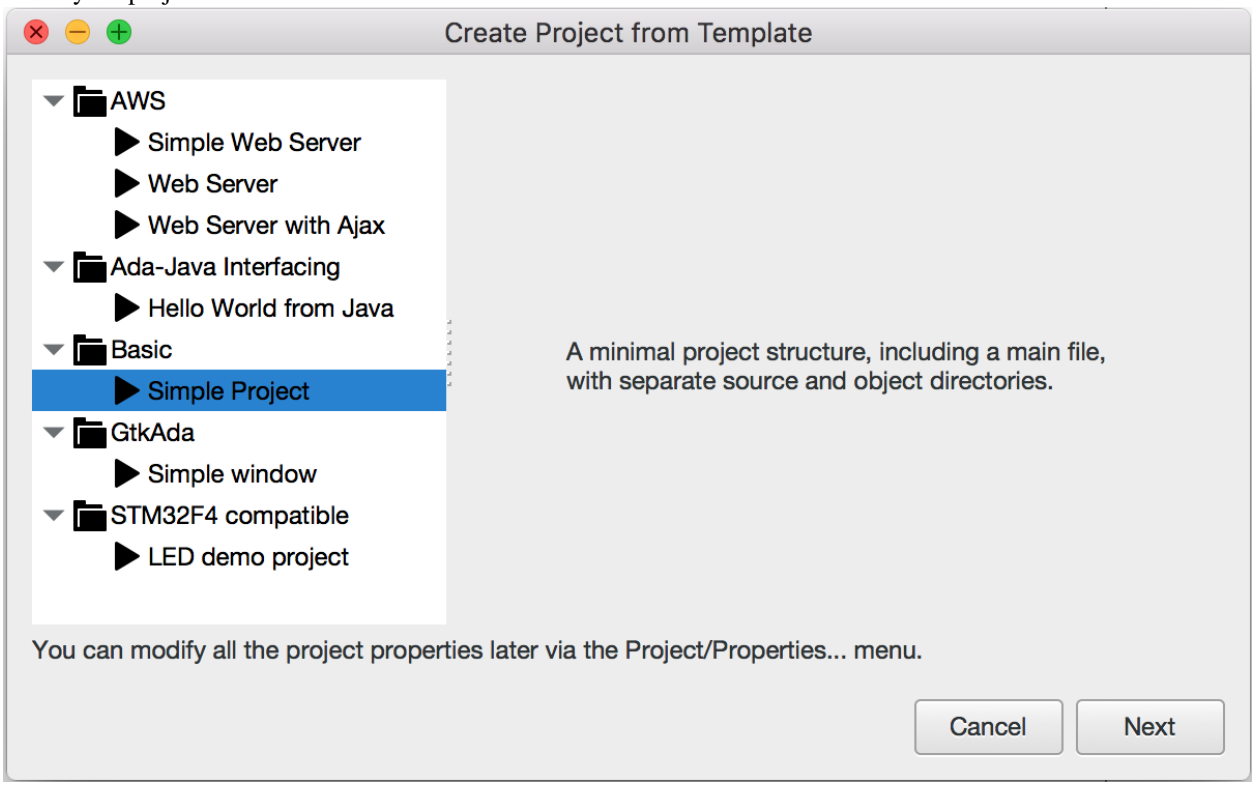
  Open (or raise if it is already open) the *Project* view on the left side of the GPS window.

5.8 The Project Wizard

The project wizard lets you create a new project file in a few steps. It contains a number of project templates, making it easy to create projects that rely on a particular technology (e.g: GtkAda).

You normally access this wizard through the *Project* → *New...* menu.

The first page of the wizard lists the various project templates. Selecting one of them and clicking on the *Next* button will show a page allowing you to modify the project template settings. Once modified, click on *Apply* to actually create your project.

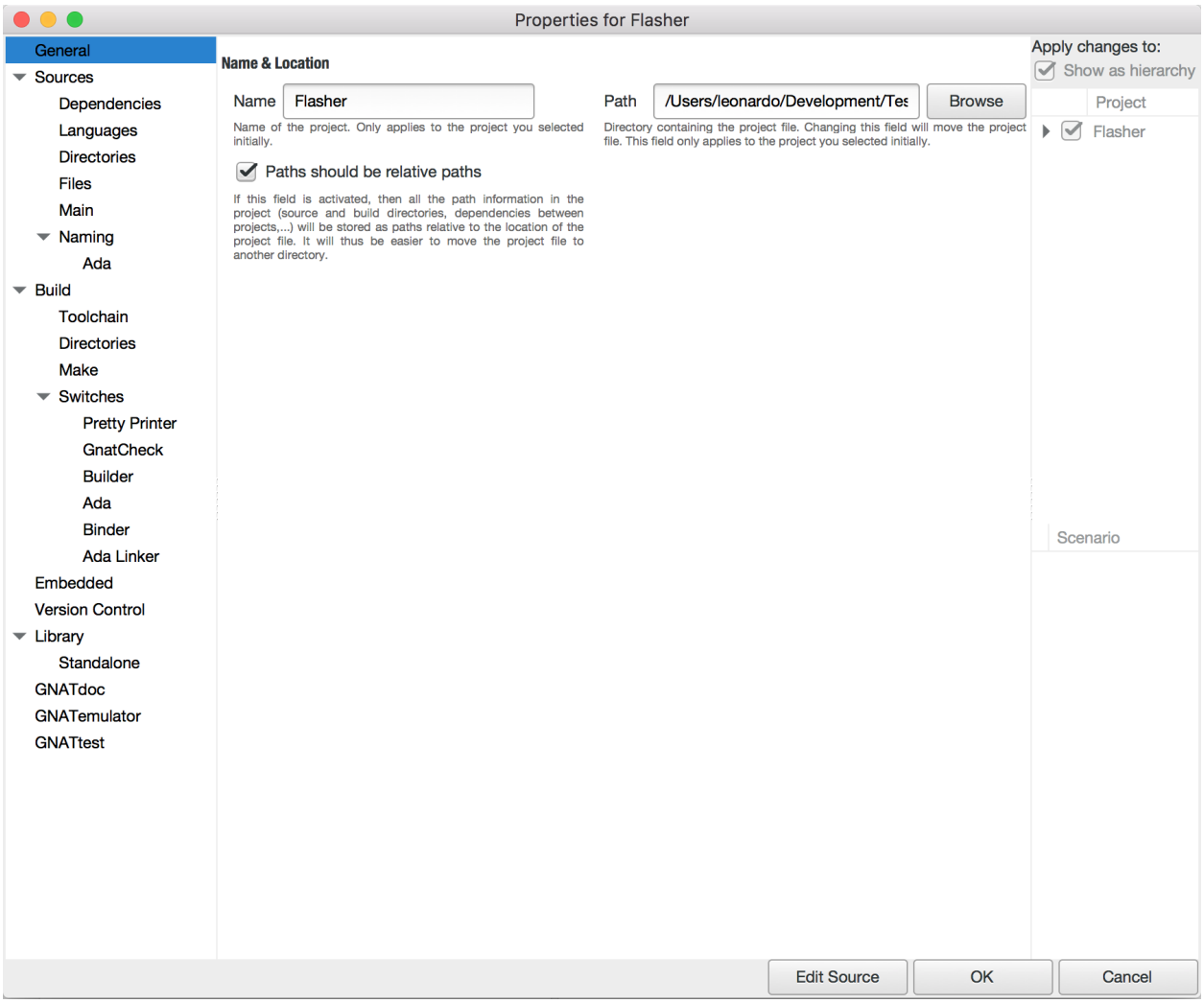


5.9 The Project Properties Editor

Use the *Project Properties* editor at any time to access the properties of your project through the *Project* → *Properties...* menu or the contextual menu *Properties* on any project item, e.g. from the *Project* views or the *Project* browser.

In some cases, GPS cannot edit your project graphically. It will still display a read-only version of the *Project Properties* dialog. This is the case, among others, when:

- the project loaded with errors, such as invalid syntax or missing directories;
- you are editing an aggregate project;
- the project was written manually before and uses advanced features like variables (`Var := ...`).



The *Project Properties* editor is divided into three parts:

The attributes editor

The contents of this editor are very similar to that of the project wizard (see *The Project Wizard*). In fact, all pages but the *General* page are exactly the same; read their description in the project wizard section.

See also *Working in a Cross Environment* for more info on the *Cross environment* attributes.

The project selector

This area, the top-right corner of the properties editor, displays a list of all 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 sorts the projects in ascending or descending order. Clicking on the left title bar (untitled) selects or deselect all the projects.

This selector has two different possible presentations, chosen by the toggle button on top: either a sorted list of all the projects, each appearing only once, or the same project hierarchy displayed in the *Project* view.

The scenario selector

This area, the bottom-right corner of the properties editor, displays all scenario variables declared in

the project hierarchy. By selecting some or all of their values, you can chose to which scenario the modifications in the attributes editor apply.

Clicking on the left title bar (untitled, on the left of the *Scenario* label) selects or deselects all values of all variables.

To select all values of a given variable, click on the corresponding check button.

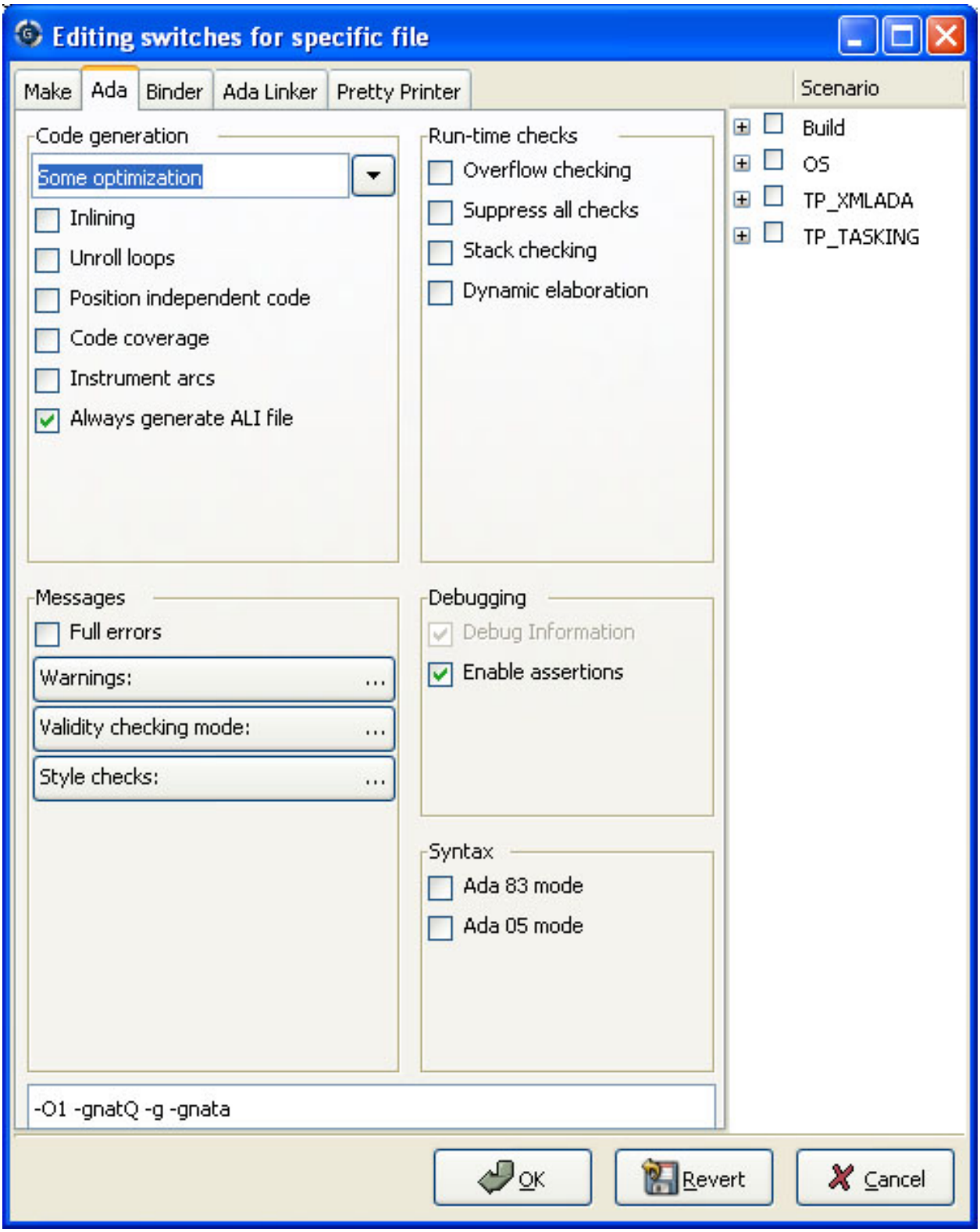
5.10 The Switches Editor

The switches editor, available through the *Project* → *Edit Switches* menu, displays all source files associated with the selected project.

For each file, it lists the compiler switches for that file. These switches are displayed in gray if they are the default switches defined at the project level (see *The Project Properties Editor*) and in black if they are specific to that file.

Edit the switches for the file by double-clicking in the switches column. You can 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 containing switches, GPS opens a new dialog allowing you to edit the switches specific to the selected files. This dialog has a button titled *Revert*, which cancels any file-specific switch and reverts to the default switches defined at the project level.


Editing switches for specific file
Make
Ada
Binder
Ada Linker
Pretty Printer
Scenario
Code generation
Some optimization
Inlining
Unroll loops
Position independent code
Code coverage
Instrument arcs
Always generate ALI file
Run-time checks
Overflow checking
Suppress all checks
Stack checking
Dynamic elaboration
Build
OS
TP_XMLADA
TP_TASKING
Messages
Full errors
Warnings:
Validity checking mode:
Style checks:
Debugging
Debug Information
Enable assertions
Syntax
Ada 83 mode
Ada 05 mode
-O1 -gnatQ -g -gnata
OK
Revert
Cancel

CHAPTER

SIX

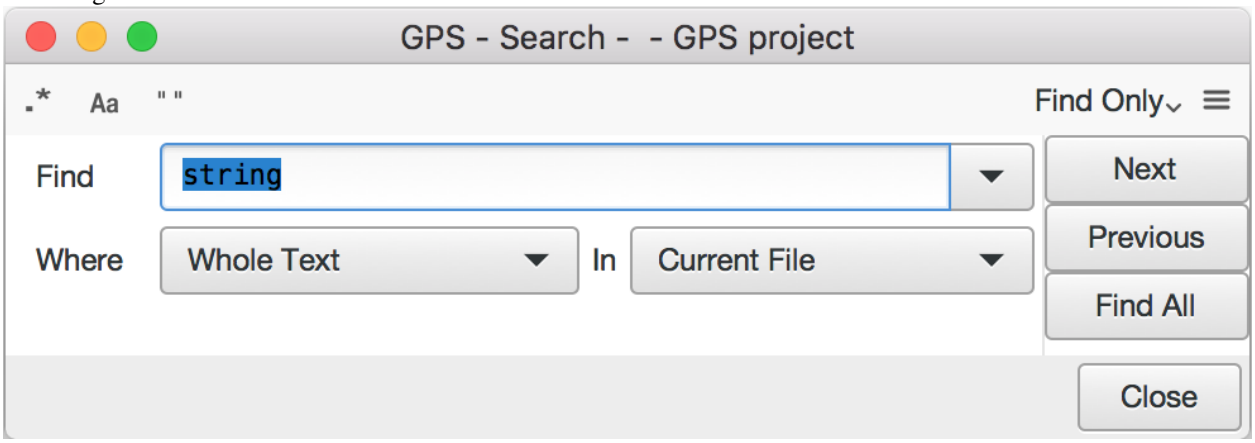
SEARCHING AND REPLACING

GPS provides extensive search capabilities in different contexts. For example, you can search in the currently edited source file or in all source files belonging to the project, even those that are not currently open. You can also search in the project view (on the left side of the main GPS window).

All of these search contexts are merged into a single graphical window that you can open either through the *Navigate* → *Find* menu or the shortcut `Ctrl-F`.

6.1 Searching

By default, the search window is floating and appears as a dialog on top of GPS. Put it inside the multiple document interface for easier access by selecting the *Window* → *Floating* menu and dropping the search window into a new location (for example, above the *Project* view). Selecting either option pops up a dialog on the screen similar to the following:



This dialog's toolbar contains several buttons that enable some specific options:

- *Regexp*

  Toggles between strings and regular expressions. Or you can select the arrow to the right of the *Search for:* field. The grammar used by 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, use the *open from project* menu (*File* → *Open From Project...*) and type `g-regpat.ads`.

- *Whole Word*

  Force the search engine to ignore substrings. For example, "sensitive" no longer matches "insensitive".

- *Case Sensitive Search*

By default, patterns are case insensitive (upper-case letters and lower-case letters are considered equivalent). Change this behavior by clicking this check box.

In addition, the dialog's local menu contains more general options used to control the behavior of the Search view:

- *Incremental search*

  Enable the incremental mode. In this mode, a search will be automatically performed whenever the search pattern is modified, starting from the current location to the next occurrence in the current file.

- *Close on Match*

  This button only appears if the search window is floating. If pressed, the search window is automatically closed when an occurrence of the search string is found.

- *Select on Match*

  Gives the focus to the editor containing the match. If not selected, the focus remains on the search window. If so, press `Enter` to search for the next occurrence.

By default, the search view contains three searching related widgets:

Search Type the string or pattern to search for.

The combo box provides a number of predefined patterns. The top two are empty patterns that automatically set the appropriate strings or regular expression mode. The other regular expressions are language-specific and match patterns such as Ada type definitions or C++ method declarations.

Where Used restrict the search to a set of language constructs. For example, use this to to avoid matching comments when you are only interested in actual code or to only search strings and comments, but not code.

In The context in which the search should occur.

GPS automatically selects the most appropriate context when you open the search dialog by looking at the component that 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*), the last one you used is selected.

Change the context to a different one by clicking on the arrow on the right, which displays the list of all possible contexts, including:

- **`Open Files`**

  Search all files currently open in the source editor.

- **`Files...`**

  Search a specified set of files. An extra *Files* box is displayed where you specify the files using standard shell (Unix or Windows) regular expressions (such as `*.ad?` for all files ending with `.ad` and any trailing character). The directory specifies where the search starts and the *Recursive search* button whether subdirectories are also searched.

- **`Files From Projects`**

  Search all files from the current project, including files from project dependencies.

- **`Files From Current Project`**

  Search all files from the current project, defaulting to the root project if none. 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).

- **`Files From Runtime`**

  Search all specification files from GNAT runtime library

- **`Current File`**

  Search the current source editor.

Normally, GPS sets the default value for *In* that matches the currently selected window. For example, if you are in an editor and open the search dialog, the context is set to *Current File*. Optionally, GPS can remember the last context that was set (see the preference *Search* → *Preserve Search Context*). In that case, if an editor is selected, GPS remembers whether the last time you started a search from an editor you decided to search in (for example) *Current File* or *Files From Project*.

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.

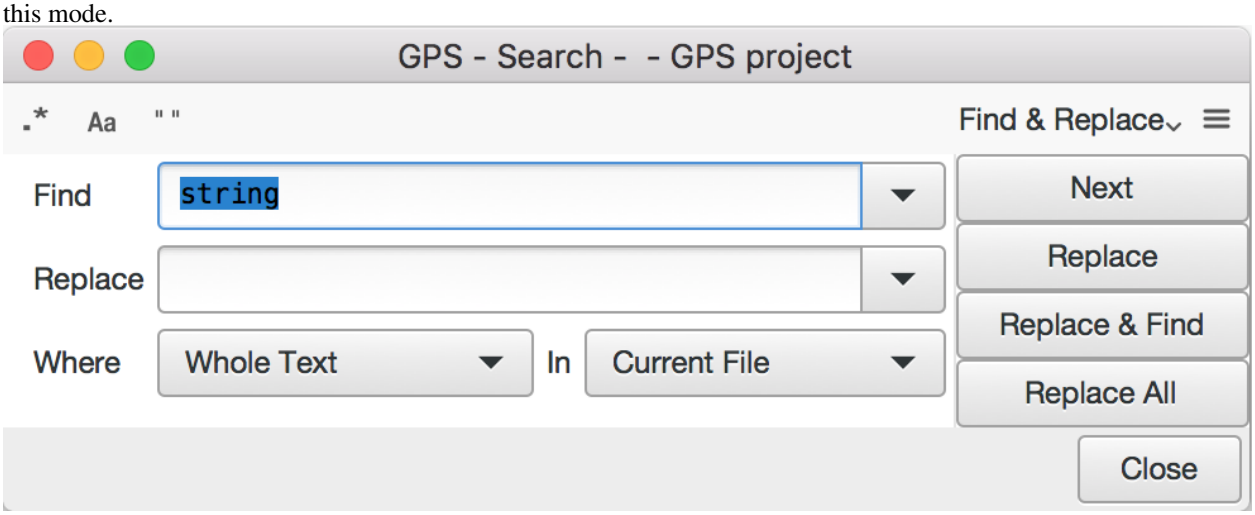
The right part of the dialog is a row of three buttons, used to navigate among the search results.

Press the *Find* or *Previous* button to perform an interactive search, which stops as soon as one occurrence of the pattern is found. At that point, the *Find* button is renamed to *Next*, which you press (or type the equivalent shortcut `Ctrl-N`) to go to the next occurrence.

The *Find all* button starts a search for all occurrences and puts the results in a view called *Locations* view, see *The Locations View*.

6.2 Replacing

The combo box present in the toolbar is used to switch the search view's mode: switch to *Find & Replace* to enable replacing capabilities. You can also use the *Navigate* → *Replace* menu or the `Ctrl-Shift-F` shortcut to switch to this mode.



In this mode, an additional field is displayed:

Replace Contains the string to replace the occurrences of the pattern. The combo box provides a history of previously used replacement strings. If a regular expression is used for search, special escapes in this field are used as:

- `\1`, `\2` .. `\9` refer to the corresponding matching subexpressions.
- `\0` refers to the complete matched string.
- `\i`, `\i(start,step)` refers to the sequentially increasing number (starting from start and increased by step on each replace).

The *Replace* and *Replace & Find* buttons are grayed out if no occurence of the pattern is found. To enable them, start a search, for example by pressing the *Find* button. Pressing *Replace* replaces the current occurrence (grays out the two

buttons) and *Replace & Find* replaces the occurrence and jumps to the next one, if any. If you do not want to replace the current occurence, jump to the next one by pressing *Next*.

The *Repl all* button replaces all occurences found. By default, a popup is displayed asking for confirmation. You can disable this popup by either checking the box *Do not ask this question again* or going to the *Search* panel of the preferences pages and unchecking *Confirmation* for *Replace all*.

Like most GPS components, the search window is under control of the multiple document interface and can be integrated into the main GPS window instead of being an external window. To do this, open the *Window* → *Search* menu in the list at the bottom of the menu, and either select *Window* → *Floating* or *Window* → *Docked*.

If you save the desktop (*File* → *Save More* → *Desktop*), GPS automatically reopens the search dialog in its new place when it is next started.

6.3 Searching in current file

The dialog we described above is convenient when you want to search in multiple files, or even in files that are not opened in GPS. However, the most frequent context is to search in the current file. GPS provides a number of facilities just for this:

- Use the *Incremental search* option

  When this option is enabled, GPS automatically jumps to the next match for the word you are currently typing.

- Use the omni-search

  At the top-right corner of the GPS window, the search field is able to search in all the sources of your project. But it can also search just in the current source. The recommended approach is once again to define a new key shortcut via *Edit* → *Key Shortcuts*, for the action *Global Search in context: current file*. Whenever you press that shortcut from now on, GPS will move the keyboard focus to the global search box, and when you type some text, a popup window will show all occurrences of that text within the current file.

CHAPTER

SEVEN

COMPILATION/BUILD

This chapter describes how to compile files, build executables, and run them. Most capabilities can be accessed through the *Build* top-level menu or through the *Build* and *Run* contextual menu items, as described below.

When GPS detects compiler messages, it adds entries to the *Locations* view, allowing you to easily navigate through the compiler messages (see *The Locations View*) and even to automatically correct some errors or warnings (see *Code Fixing*).

In source editors, compiler messages also appear as icons on the side of each line that has a message. When the pointer is placed on these icons, a tooltip appears, listing the error messages posted on the corresponding line. When GPS can automatically correct the errors, clicking the icon applies the fix. These icons are removed when the corresponding entries are removed from *The Locations View*.

7.1 The Build Menu

Use the build menu to access capabilities related to checking, parsing, and compiling files as well as creating and running executables. This menu is fully configurable via the *Targets* settings; what is documented here 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* view if no file is currently selected.

- *Build* → *Check semantic*

  Check the semantics of the current source file. Display an error message in the *Messages* view if no file is currently selected.

- *Build* → *Compile file*

  Compile the current file. By default, displays 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* view 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. To remove the highlighting on these lines, remove the files from the *Locations* view using either the *Remove category* contextual menu item or by closing the *Locations* view.

- *Build* → *Project* → *<main>*

  List all main units defined in your project hierarchy, if any. Each menu item builds the selected main. The list is either a flat list when there is a small number of mains, or they are grouped into projects when there is a large number of the root project is an aggregate project.

- *Build → Project → Build All*

  Build and link all main units defined in your project. If no main unit is defined in your project, build all files defined in your project and subprojects recursively. For a library project, 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 allowing you to enter any external command. Use this item when you already have existing build scripts, make files or similar and want to invoke them from GPS. If the `SHELL` environment variable is defined (to, e.g. `/bin/sh`), the syntax used to execute the command is the one for that shell. Otherwise, GPS spawns the command without any shell interpretation.

- *Build → Clean → Clean all*

  Remove all object files and other compilation artifacts associated with all projects related to the current one. This allows restarting a complete build from scratch.

- *Build → Clean → Clean root*

  Remove all object files and other compilation artifacts associated to the root project but do not clean objects from other related projects.

- *Build → Makefile*

  If the **`make`** utility is in your PATH and you have a file called `Makefile` in the same directory as your project file or if you have set the *makefile* property in the *Make* section of the project properties (see *The Project Properties Editor*), this menu is displayed, giving access to all the targets defined in your makefile.

- *Build → Ant*

  If the **`ant`** utility is in your PATH and you have a file called `build.xml` in the same directory as your project file or if you have set the *antfile* property in the *Ant* section of the project properties (see *The Project Properties Editor*), this menu is displayed, giving access to all the targets defined in your ant file.

- *Build → Run → Project → <main>*

  For each main source file defined in your top level project, displays an entry to run the executable associated with that file. Running an application first opens a dialog where you can optionally specify command line arguments to your application. You can also specify whether the application should be run within GPS (the default) or using an external terminal.

  When running an application within GPS, a new execution view is added to the bottom area to display input and output of the application. This view is not closed automatically, even when the application terminates, so you still have access to the application's output. If you explicitly close an execution window while an application is running, GPS displays a dialog window to confirm whether the application should be terminated.

  When using an external terminal, GPS launches an external terminal utility to perform your application's execution and input/output. Configure this external utility in the *External Commands → Execute command* preferences dialog.

  The GPS execution views have several limitations that external terminals do not. In particular, they do not handle signals like `ctrl-z` and `control-c`. If you are running an interactive application, we strongly encourage you to run it in an external terminal.

Similarly, the *Run* contextual menu item of a project entity contains the same entries.

- *Build* → *Run* → *Custom...*

  Similar to the option above, except 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 that shell. Otherwise, GPS spawns the command directly without any shell interpretation.

- *Build* → *Settings* → *Targets*

  Opens the Target Configuration Dialog. See *The Target Configuration Dialog*.

- *Build* → *Settings* → *Toolchains*

  Opens a dialog allowing GPS to work with two compilation toolchains. This is particulary useful when you need to compile a project with an old compiler but want up-to-date functionality from the associated tools (for example, **`gnatmetric`** and **`gnatcheck`**). See *Working with two compilers*.

- *Tools* → *Interrupt*

  Interrupts the last compilation or execution command. Once you interrupted the last operation, you can interrupt the previous one by selecting the same menu item again.

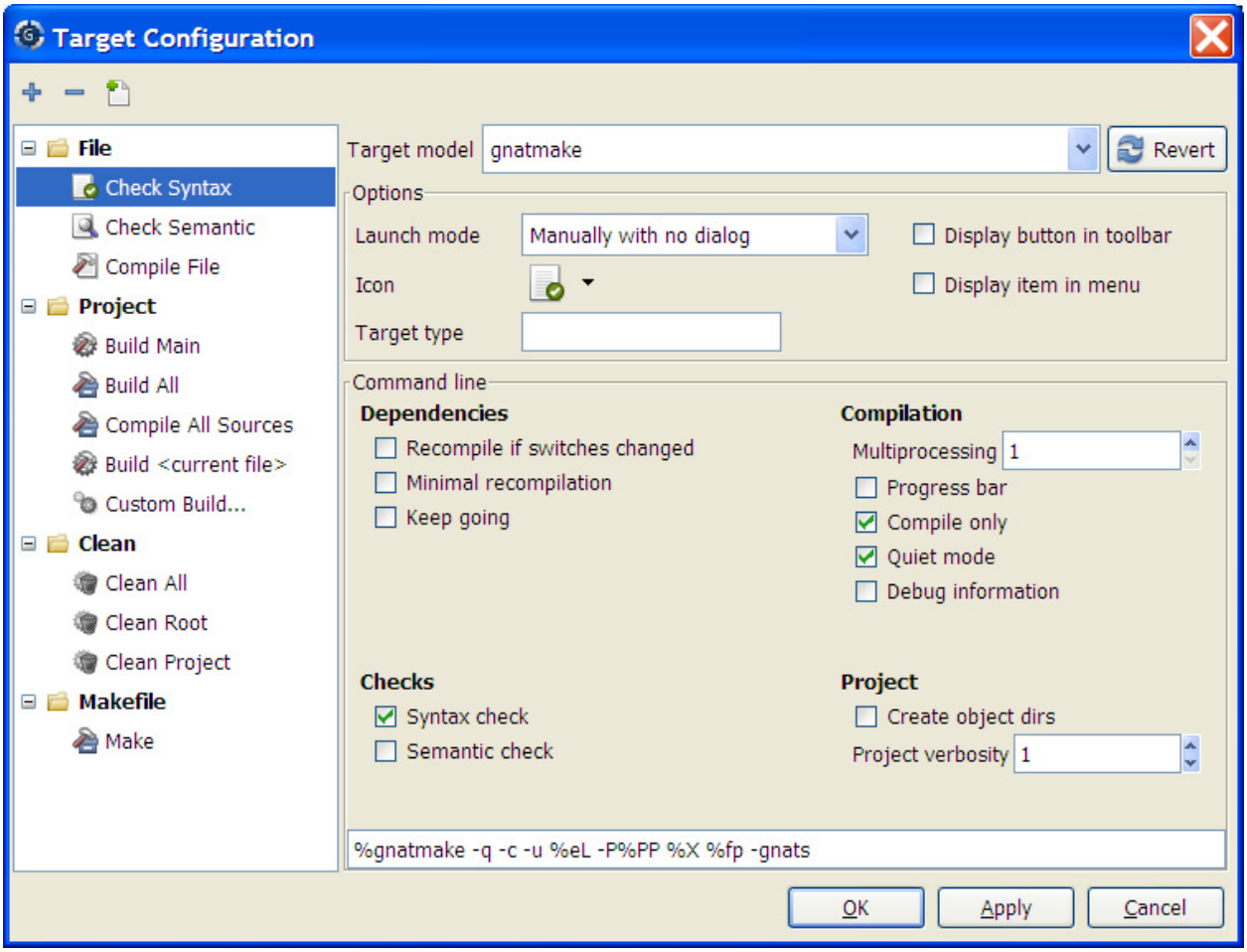
- *Tools* → *Views* → *Tasks*

  The easiest way to interrupt a specific operation, whether it was the last one started or not, is to start the *Tasks* view via the *Tools* → *Views* → *Tasks* menu, which shows one line for each running process. Right-clicking on any of these lines allows interrupting that process.

If your application is built through a `Makefile`, you should probably load the `makefile.py` startup script (see the *Edit* → *Preferences* → *Plugins* menu).

7.2 The Target Configuration Dialog

GPS provides an interface for launching operations such as building projects, compiling individual files, and performing syntax or semantic checks. These operations all 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. See *Customizing Build Targets and Models*.



This dialog is divided in two areas: on the left is a tree listing Targets and in the main area is 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 that is identical to the currently selected target.

7.2.2 The configuration panel

From the top of the configuration panel, you can select the Target model. That 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 available for all Targets.

- The Launch mode selects the way the target is launched:
  - Manually:

    The target is launched when you click on the corresponding icon in the toolbar or activate the corresponding menu item. In the latter case, a dialog is displayed, allowing final modifications of the command line.
  - Manually with dialog:

    Same as Manually, but the dialog is always displayed.
  - Manually with no dialog:

    Same as Manually, but the dialog is never displayed.
  - 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*.
- Icon:

  The icon to use for representing this target in the menus and in the toolbar. To use one of your icons, register icons using the `<stock>` XML customization node. (See *Adding custom icons*). Then use the "custom" choice and enter the ID of the icon into the text field.
- Target type:

  Type of target described. If empty or set to "Normal", it represents a simple target. If set to another value, it 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 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 that can be used 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 example 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 that are specific to the Model of this Target.

The full command line is displayed at the bottom. It may contain Macro Arguments. For example, 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 can launch compilation targets in the background. This means GPS launches the compiler on the current state of the file in the editor.

Error messages resulting from background compilations are not listed in the *Locations* or *Messages* views. The full list of messages are shown in the *Background Build* console, accessible from the *Tools* → *Consoles* → *Background Builds* menu. Error messages that 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. In both places, tooltips show the contents of the error messages.

Messages from background compilations are removed automatically when either a new background compilation has finished or a non-background compilation is launched.

GPS launches background compilations for all targets that have a *Launch mode* set to *In background* after you have made modifications in a source editor. Background compilation is mostly useful for targets such as `Compile File` or `Check Syntax`. For targets that operate on Mains, the last main 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 (i.e. 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 (see *Scenario view*).

When the mode is set to "default", GPS performs the build using the switches defined in the project. When the mode is set to another value, specialized parameters are passed to the builder. For example, the **`gcov`** mode adds all the compilation parameters needed to instrument the generated objects and executables to work with the **`gcov`** tool.

In addition to changing the build parameters, changing the mode changes the output directory for objects and executables. For example, 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.

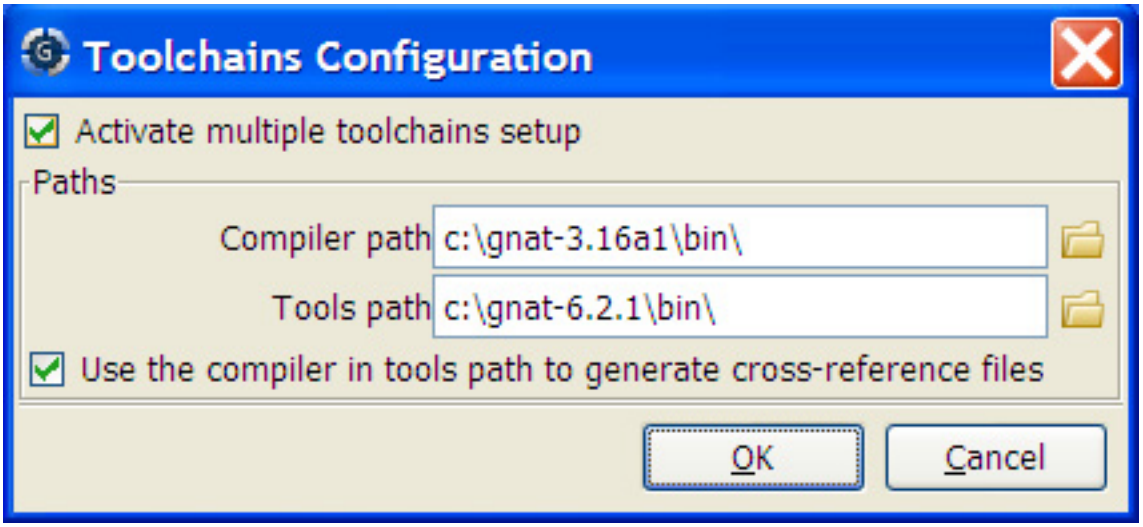
You can define new modes using XML customization, see *Customizing Build Targets and Models*.

The Build Mode only affects 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 if your projects need to be compiled with a specific (old) version of the GNAT toolchain while you still need 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, or computing metrics.

To configure GPS to handle two compiler toolchains, use the *Build* → *Settings* → *Toolchains* menu. This opens a dialog from which 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 is used to compile the code, while the second is used to run up-to-date tools in order to get more functionality or more accurate results. GPS only enables the *OK* button when the two paths are set to different locations, as that is the only case where it makes sense to enable the multiple toolchains mode.

You can also activate an automated cross-reference generation from this dialog. The cross-reference files are the `.ali` files generated by the GNAT compiler together with the compiled object files. The `.ali` files are used by GPS for several purposes, such as cross-reference browsing and documentation generation. Having those `.ali` files produced by a recent compiler provides more accurate results for those purposes but might cause serious problems if an old compiler were to also read those `.ali` files when compiling a project.

If you activate the automated cross-reference generation, GPS generates those `.ali` files using the compiler found in the tools path and places 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 the old toolchain's `.ali` files remain untouched.

Cross-reference files generation does not output anything in the *Messages* view so as not to be confused with the output of the regular build process. If needed, you can see the output of the cross-ref generation command with 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 wanting up-to-date tools on the local side, GPS does the following when both a remote compilation server is defined and the multiple toolchains mode is in use:

- The compiler path is ignored when a remote build server is defined. All compilation actions are performed normally on the build server.
- The tools path is used and all related actions are performed on the local machine using this path.
- The cross-reference files are handled **`rsync`** so they do not get overwritten during local and remote host synchronizations. Otherwise, they would cause build and cross-reference generation actions to occur at the same time on the local machine and on remote server.

CHAPTER

EIGHT

DEBUGGING

GPS also serves as a graphical front-end for text-based debuggers such as GDB. If you understand the basics of the underlying debugger used by GPS, you will better understand how GPS works and what kind of functionality 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 other components of GPS. For example, you can edit files and navigate through your sources while debugging.

To start a debug session, click on the *Debug* button in the main toolbar or go to the *Debug* → *Initialize* menu and choose either the name of your executable, if you specified the name of your main program(s) in the project properties, or start an empty debug session using the <*no main file*> menu. You can then load any file to debug, by using the *Debug* → *Debug* → *Load File...* menu.

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).

Create multiple debuggers by using the *Debug* → *Initialize* (or the corrsponding toolbar button) menu several times: this creates a new debugger each time. All debugger-related actions (e.g. stepping, running) are performed in the current debugger, represented by the current debugger console. To switch to a different debugger, select its corresponding console. Setting breakpoints, though, will be done for all debuggers, to help debug when you work on multiple executables that share code.

After the debugger has been initialized, you have access several new views: the debugger console (in a new page, after the *Messages* window), the *Breakpoints* views and the *Variables* view.

You can now access any of the menus under *Debugger*, and you also have access to additional contextual menus, in particular in the source editor where you can easily display variables, set breakpoints, and get automatic displays (via tooltips) of object values.

To exit the debugger without quitting GPS, use the *Debug* → *Terminate Current* menu, which terminates your current debug session, or the *Debug* → *Terminate* menu which terminates all of your current debug sessions.

8.1 The Debug Menu

The *Debug* entry in the menu bar provides operations acting at a global level. Key shortcuts are available for the most common operations and are displayed in the menus. Here is a detailed list of the items 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 execution should stop at the beginning of the main subprogram. If you confirm by clicking the *OK*

button, GPS starts the program with the arguments you entered. Note, the user should do the quoting himself if he has some special characters inside arguments.

- *Debug → Step*

  Execute the program until it reaches the next source line.

- *Debug → Next*

  Execute the program until it reaches the next source line, stepping over subroutine calls.

- *Debug → Step Instruction*

  Execute the program until it reaches the next machine instruction.

- *Debug → Next Instruction*

  Execute the program until it reaches the next machine instruction, stepping over subroutine calls.

- *Debug → Finish*

  Execute the program until the subprogram running in the selected stack frame returns.

- *Debug → Continue*

  Continue execution of the program being debugged.

- *Debug → Interrupt*

  Asynchronously interrupt the program being debugged. Depending on the state of the program, it may stop in low-level system code that does not have debug information or, in some cases, even a coherent state. You should use breakpoints instead of interrupting programs, if possible. However, interrupting programs is nevertheless required in some situations, for example when the program appears to be in an infinite (or at least very long) loop.

- *Debug → Terminate Current*

  Terminate the current debug session by terminating the underlying debugger (e.g, **gdb**) used to handle the low level debugging. Control what happens to the windows through the *Debugger → Debugger Windows* preference.

- *Debug → Terminate*

  Terminate all your debug sessions. This is the same as *Debug → Terminate Current* if you only have one debugger open.

8.1.1 Initialize

This menu contains one item per main unit defined in your project. Selecting that item starts a debug session and loads the executable associated with the main unit selected and, if relevant, all corresponding settings: a debug session opens the debug perspective and associated debug properties (e.g. saved breakpoints and data display).

- *Debug → Initialize → <No Main File>*

  Initializes the debugger with no executable. Then use one of the other menu entries such as *Debug → Debug → Load File* or *Debug → Debug → Attach*.

8.1.2 Debug

- *Debug → Debug → Connect to board*

  Opens a dialog to connect to a remote board. This option is only relevant for cross debuggers.

- *Debug → Debug → Load File...*

  Opens a file selection dialog allowing 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*

  Adds the symbols from a given file. This corresponds to the **`gdb`** command **`add-symbol-file`**. This menu is particularly useful under VxWorks targets, where modules can be loaded independently of the debugger. For example, if a module is independently loaded on the target using **`windshell`**, you must use this functionality for the debugger to work properly.

- *Debug → Debug → Attach...*

  Instead of starting a program to debug, attach to an already running process. To do so, 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. Like *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; the executable continues to run independently. Use the *Debug → Debug → Attach To Process* menu to later re-attach to this process.

- *Debug → Debug → Debug Core File*

  Opens a file selection dialog allowing you to debug a core file instead of a running process. 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

Most items in this menu need to access the underlying debugger when the process is stopped, not when it is running, so you first need to stop the process at a breakpoint or interrupt it before using the following items. Failure to do so will result in empty windows.

- *Debug → Data → Data Window*

  Displays the *Data* browser. If it already exists, it is raised so it becomes visible.

- *Debug → Data → Variables*

  Displays the *Variables* view, or raise an already existing one. See *The Variables View* for more details.

- *Debug → Data → Call Stack*

  Displays the *Call Stack* view. See *The Call Stack View* for more details.

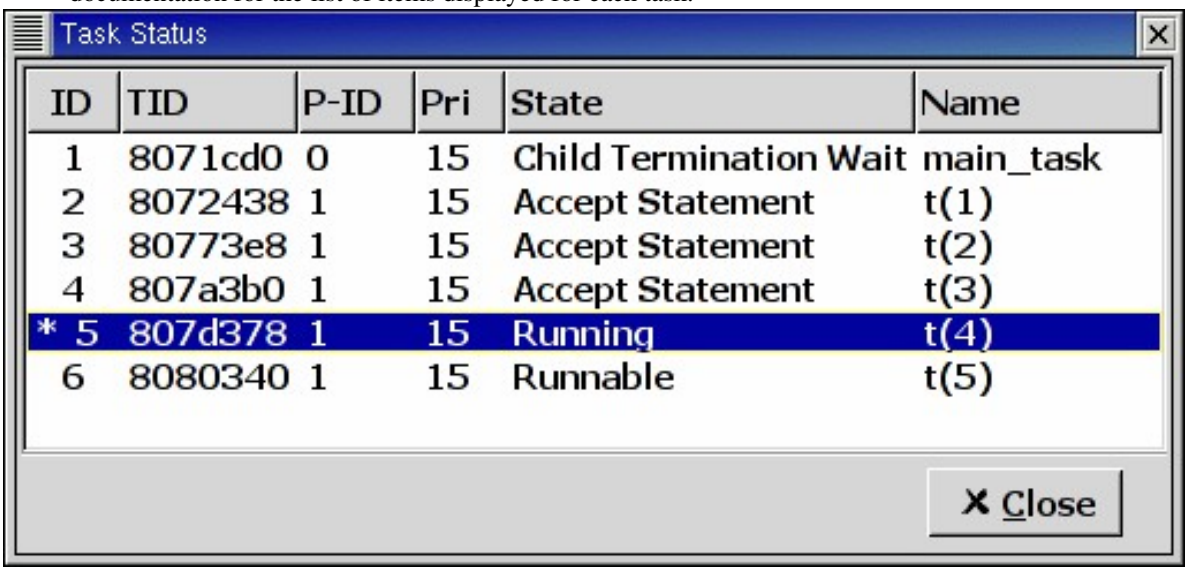
- *Debug → Data → Threads*

  Opens a new window containing the list of threads currently present in the executable as reported by the underlying debugger. For each thread, it gives language- and debugger-dependent information such as the internal identifier, name, and status. Refer to the underlying debugger's documentation for more details. Like other similar commands, the process being debugged needs to be stopped before using this. If not, GPS will display an empty list.

  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, opens a new window containing the list of Ada tasks currently present in the executable. Just like 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.



- *Debug → Data → Protection Domains*

  For VxWorks AE only, opens 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 $\star$ character indicates the current protection domain.

- *Debug → Data → Assembly*

  Opens a new window displaying an assembly listing of the current code being executed. See *The Assembly Window* for more details.

- *Debug → Data → 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 and editor. See *The Memory View* for more details.

- *Debug → Data → Command History*

  Opens a dialog with the list of commands executed in the current session. Select any number of items in this list to replay the selection.

- *Debug → Data → Display Local Variables*

  Opens an item in the *Data* browser containing all local variables in the current frame.

- *Debug → Data → Display Argument*

  Opens an item in the *Data* browser containing the arguments for the current frame.

- *Debug → Data → Display Registers*

  Opens an item in the *Data* browser containing the current value of the machine registers for the current frame.

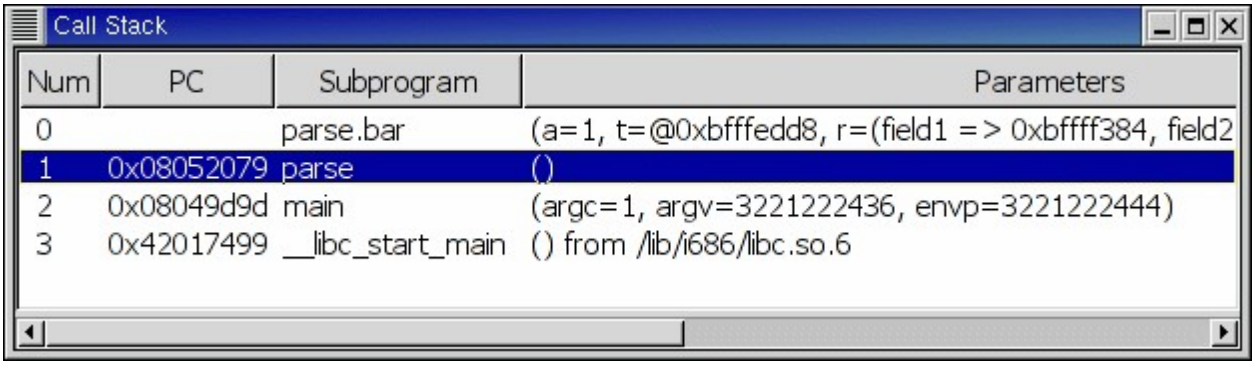
- *Debug → Data → Display Any Expression...*

Opens a small dialog letting you specify an arbitrary expression in the *Data* browser. This expression can be a variable name or a more complex expression following the syntax of the underlying debugger. (See the debugger documentation for more details on the syntax.) Enable the check button *Expression is a subprogram call* 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 items displayed in the *Data* browser.

8.2 The Call Stack View



The call stack view lists the frames corresponding to the current execution stack for the current thread or 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). Click on any frame to switch to that caller's context; this updates the display in the source window. Use the up and down buttons in the tool bar to go up and down one frame in the call stack.

The local configuration menu 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 machine address corresponding to the function's entry point.

- *Subprogram Name*:

  The name of the subprogram.

- *Parameters*:

  The parameters to the subprogram.

- *File Location*:

  The filename and line number information.

By default, only the subprogram name is displayed. Hide the call stack view by closing it and show it again using the menu *Debug* → *Data* → *Call Stack* menu.

Showing extra information like the value for parameters requires more work from the debugger, and thus will be slower.

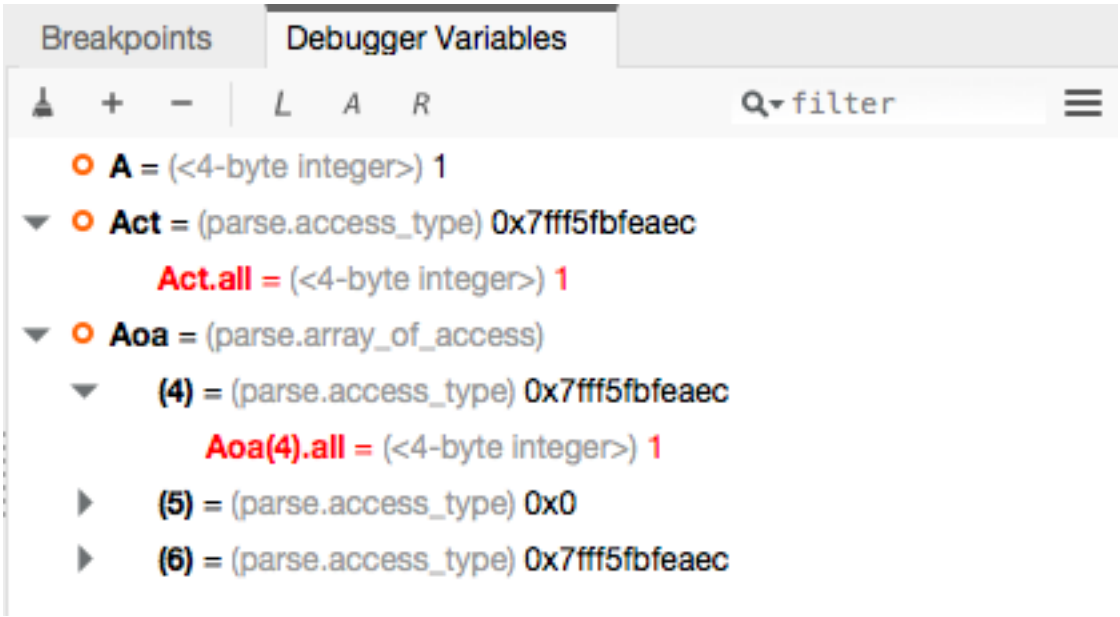
8.3 The Variables View

The *Variables* view displays the value of selected variables or debugger command every time the debugger stops. The display is done in a tree, so that for instance the fields of a record are displayed in child nodes (recursively).

Access types (or pointers) can also be expanded to show the value they reference.

Values that have been modified since the debugger last stopped are highlighted in red.

This value is very similar to *The Data Window*.



8.4 The Data Window

8.4.1 Description

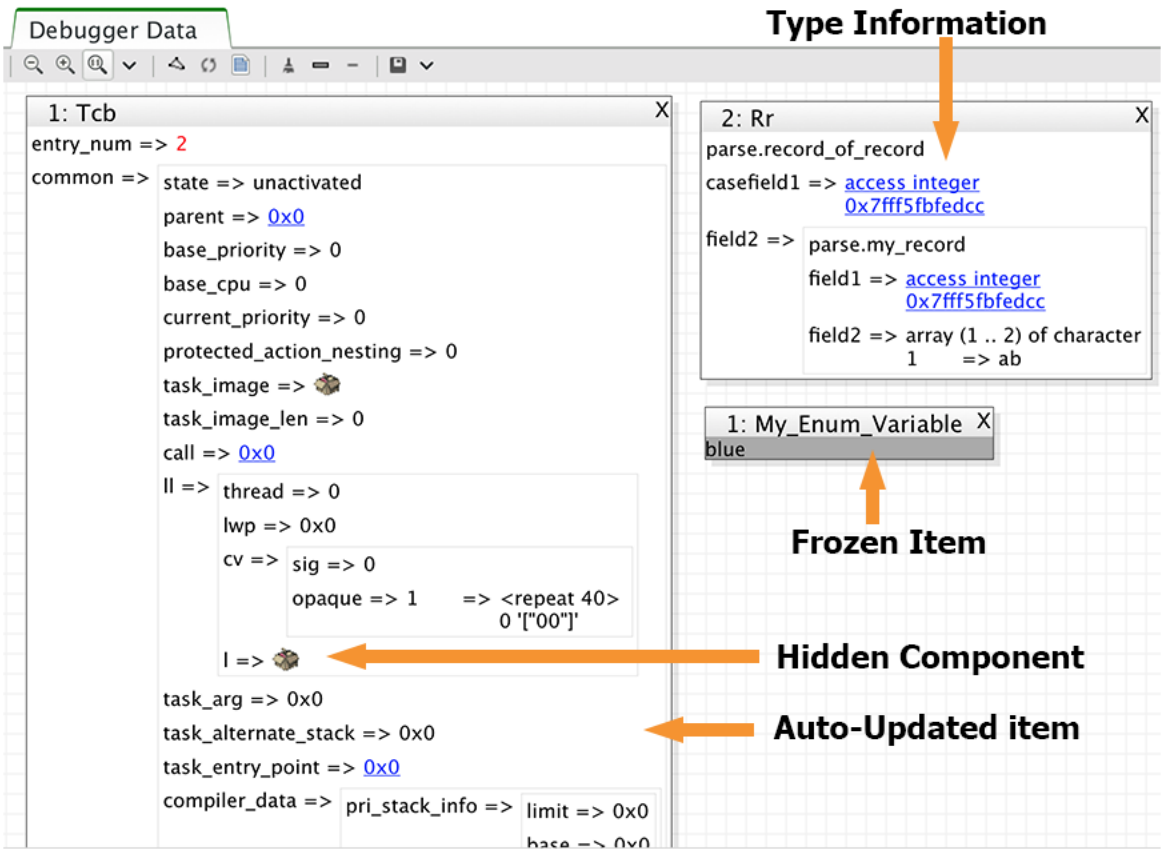
The Data browser is the area in which various information about the process being debugged is displayed. This includes the value of selected variables, the current contents of registers, and local variables.

This browser is open by default when you start the debugger. Force it to display through the menu *Debug* → *Data* → *Data Window*.

By default, the contents of the data browser is preserved whenever you close it: if you reopen it either during the same debugger session or automatically when you start a debugger on the same executable, it displays the same items as previously. This behavior is controlled by the *Debugger* → *Preserve State on Exit* preference.

The data browser contains all the graphic boxes that can be accessed using the *Debug* → *Data* → *Graph Display** menus, the data browser *Display Expression...* contextual menu, the editor *Display* contextual menu items, and the *graph* item in the debugger console.

In each of these cases, a box is displayed in the data browser with the following information:



- A title bar containing:
  - The number of this expression: 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 flashlight or a lock.

    This is a clickable icon that changes the state of the box from automatically updated (the flashlight icon) to frozen (the lock icon). When frozen, the value is grayed out and does not change until you change the state. When updated, the value of the box is recomputed each time an execution command is sent to the debugger (e.g step, next).
  - An icon representing an 'X'. Click on this to close and delete any box.
- A main area.

  The main area displays the data value hierarchically in a language-sensitive manner. The browser knows about data structures of various languages such as C, Ada, and C++ and organizes them accordingly. For example, each field of a record, struct, or class or each element of an array is displayed separately. For each subcomponent, a thin box is displayed to separate it from other components.

A contextual menu, that takes into account the current component selected by the pointer, gives access to the following menus:

- *Close *component**

  Closes the selected item.
- *Hide all *component**

Hides all subcomponents of the selected item. To select a particular field or element in a record or array, move the pointer over the name of the component (not over the box containing its values).

- *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**

  Displays the memory view dialog and explores memory at the address of the component.

- *Set value of *component**

  Sets the value of a selected component. This opens an entry box allowing you to enter the new value of a variable or component. The underlying debugger 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 on program execution (e.g step, next).

The *Data* browser has a local menu bar containing 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, the address of all items already displayed on the canvas are compared with the address of a new item to display. If they match (for example, if you tried to dereference a pointer to an object already displayed), GPS will display a link instead of creating a new item.

Zoom in

Redisplays the items with a bigger font.

- *Zoom out*

  Displays the items with smaller fonts and pixmaps. Use this when you have several items in the browser and you cannot see all of them at the same time (for example, a tree whose structure you want to see clearly).

- *Zoom*

  Choose the zoom level directly from a menu.

- *Clear*

  All the boxes currently displayed are removed.

8.4.2 Manipulating items

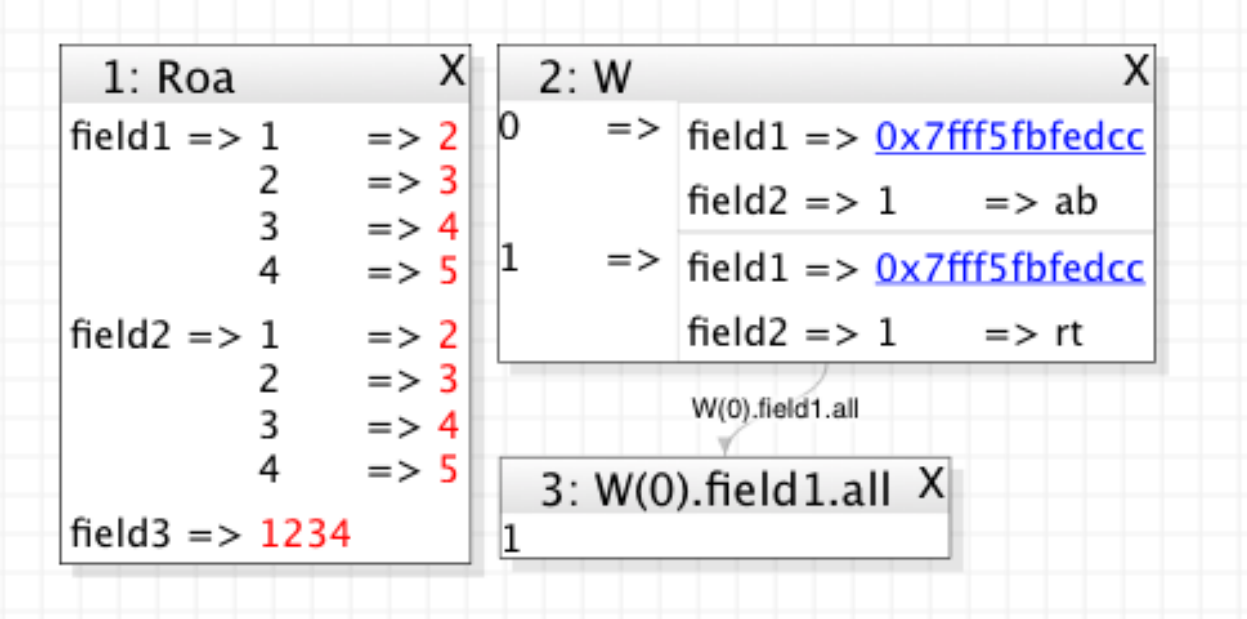
Moving items

You can manipulate all items with your mouse, and you can move them anywhere within the browser. If you try to move an item outside of the visible area of the browser, GPS scrolls the browser to make the new position visible.

GPS also provides automatic scrolling if you move the pointer while dragging an item near the borders of the browser. While the pointer remains close to the border and the mouse is pressed while hovering on the item, GPS scrolls the browser and moves the item. 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. The default meaning of each colors is as follows (the colors can be changed through the preferences dialog):



black

The default color used to print the value of variables or expressions.

blue 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 dereference these (that is to say see the value pointed to) by double-clicking on the blue text itself.

red

Used for variables and fields whose value has changed since the data window was last displayed. For example, if you display an array in the data browser and then select the *Next* button in the tool bar, the elements of the array whose value has just changed appear in red.

As another example, if you choose to display the value of local variables in the data window (*Display* → *Display Local Variables*), only the variables whose value has changed are highlighted; the others remain black.

Icons

Several different icons can be seen when displaying items. They convey the following special meanings:

trash bin icon

Indicates the debugger could not get the value of the variable or expression. For example, because the variable is currently not in scope (and thus does not exist) or might have been optimized away by the compiler. In all cases, the display is updated as soon as the variable's value is known again.

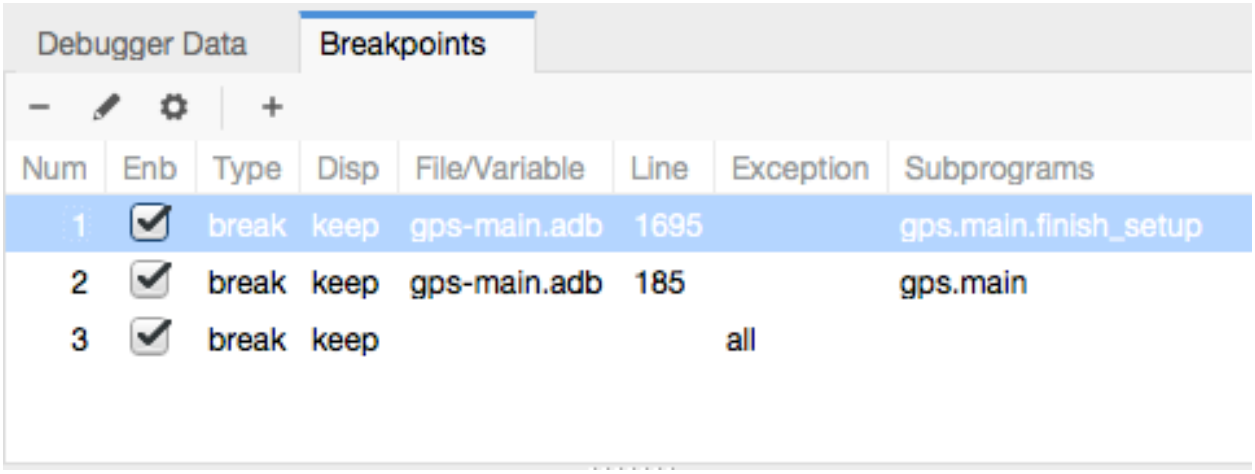
package icon

Indicates part of a complex structure is currently hidden. Manipulating huge items in the data window (for example 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 icon expands the hidden part and clicking on any subrectangle in the display of the variable hides that part and replaces it with this icon.

See also the description of the contextual menu to automatically show or hide all the contents of an item. An alternative to hiding subcomponents is to clone them in a separate item (see the contextual menu).

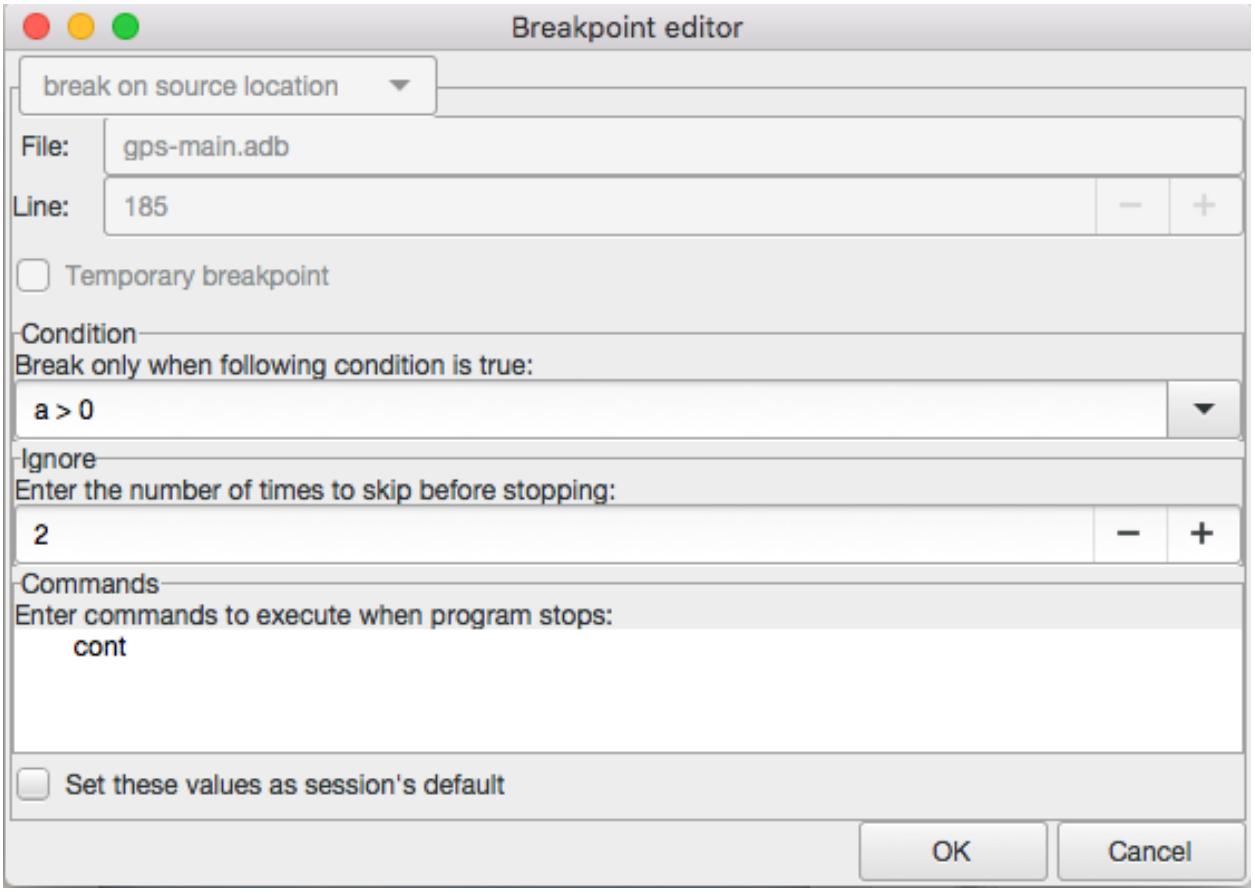
8.5 The Breakpoint Editor



Access the breakpoint editor from the *Debug* → *Data* → *Breakpoints* menu. It allows you to manipulate the various kinds of breakpoints: those at a source location, on a subprogram, at an executable address, on memory access (watchpoints), or on Ada exceptions.

This view lists the existing breakpoints that are currently set in the debugger. You can quickly and conveniently enable or disable breakpoints by clicking on the checkboxes directly in the list.

Select a breakpoint in the list and click on the *View* button in the toolbar to shows the corresponding editor at that location. You can alternatively double-click on the breakpoint.



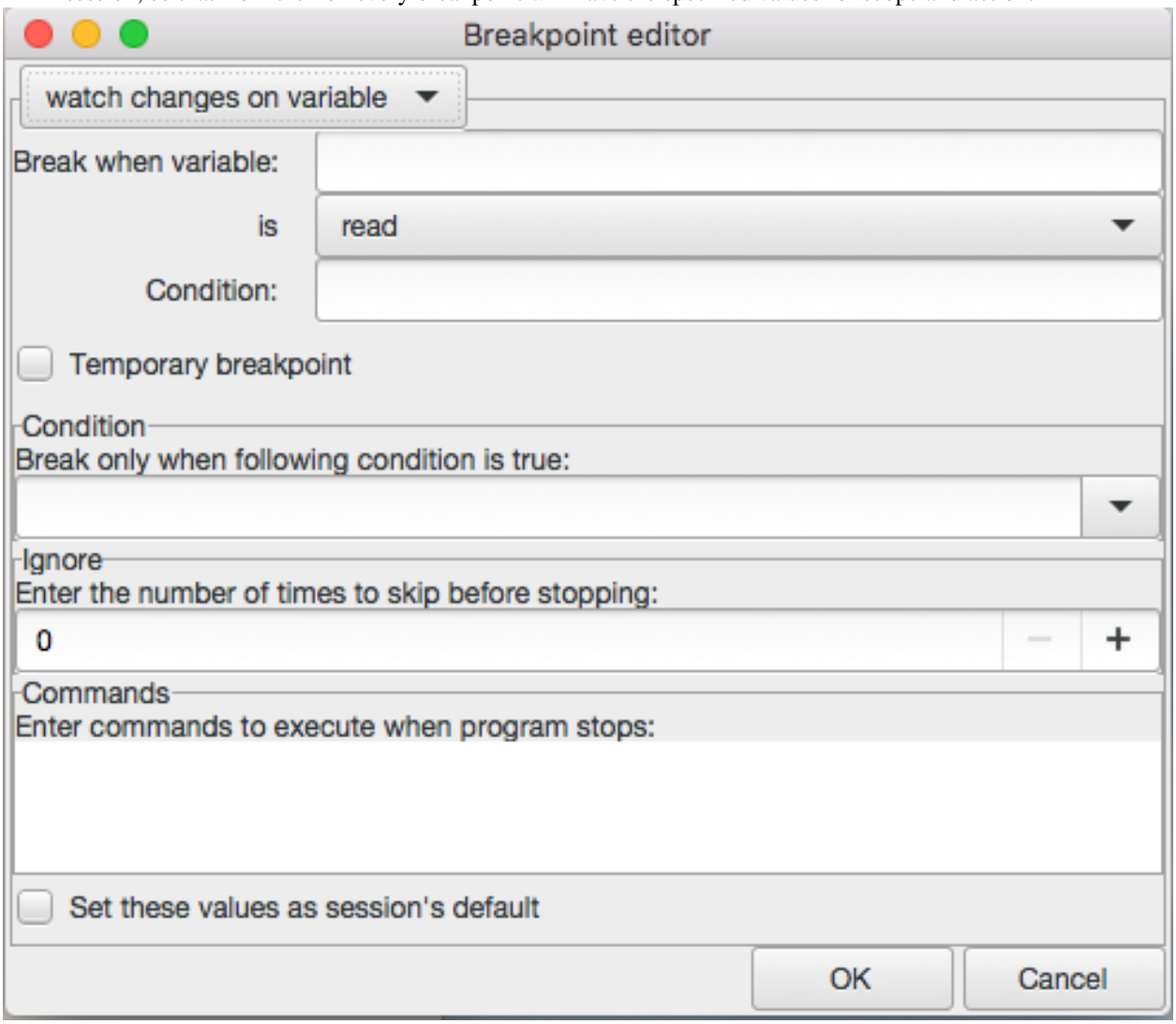
To view the details of a breakpoint, select it in the list and click on the *Edit* button in the toolbar. You can also do a long click on the breakpoint (keep your mouse pressed for a short while).

This opens up a separate dialog that shows the various attributes:

- Details on where the breakpoint is set: the file and line, the specific address in memory, or the name of the exception which will stop the debugger when raised. These are not editable, so to change this you must create a new breakpoint instead;
- The conditions to be met for the debugger to stop at that location. Such conditions can refer to variables valid at that location, and for instance test the value of specific variables;
- The number of times that the breakpoint should be ignored before the debugger actually stops. This is useful when you know the error occurs after the 70th time hitting the breakpoint;
- Debugger commands to execute when reaching the breakpoint.
- When running VxWorks AE, this dialog also lets you two extra properties:
  - The **scope** indicates which tasks will be stopped. Possible values are:
    - task: The breakpoint only affects the task that was active when the breakpoint was set. If the breakpoint is set before the program is run, the breakpoint affects the environment task.
    - pd: Any task in the current protection domain is affected by the breakpoint.
    - any: Any task in any protection domain is affected by the breakpoint. This setting is only allowed for tasks in the Kernel domain.
  - The **action** indicates which tasks are stopped when the breakpoint is hit:
    - task: only the task that hit the breakpoint.

* pd: all tasks in the current protection domain.
* all: all stoppable tasks in the system.

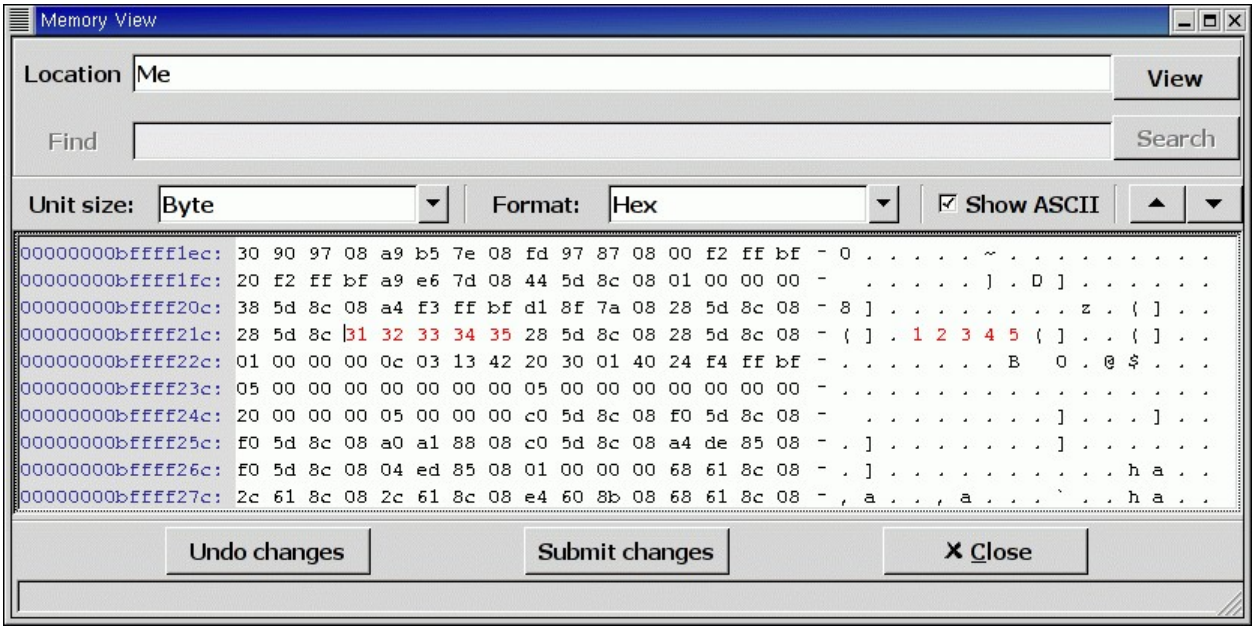
Both of these properties can either be configured for the specific breakpoint, or configured as the default for the session, so that from then on every breakpoint will have the specified values for scope and action.



To create new breakpoints, click on the *Add* button in the toolbar. This opens up the same dialog as above, but lets you edit the top section (file, line, exception, address,...). Select the type of the breakpoint or watchpoint at the top.

If you enabled the preference *Debugger* → *Preserve state on exit*, GPS automatically saves the currently set breakpoints and restores them the next time you debug an executable in the same project. This allows you to immediately start debugging your application without having to set the breakpoints every time. These breakpoints will be reused for all executables in the same project.

8.6 The Memory View



The memory view allows you to display the contents of memory by specifying either an address or a variable name.

To display memory contents, enter either the address using the C hexadecimal notation (0xabcd) or the name of a variable in the *Location* text entry. (If a variable is entered, the underlying debugger computes its address.) Then either press `Enter` or click the *View* button. GPS displays the memory with the corresponding addresses in the bottom text area.

Specify the unit size (*Byte*, *Halfword* or *Word*) and 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 allow you to walk through the memory in order of ascending or descending addresses respectively.

Finally, modify a memory area by clicking on the location you want to modify and entering the new values. Modified values appear in a different color (red by default) and are only written to the target when you click on the *Submit changes* button. Clicking on *Undo changes* or going up or down in the memory also undoes your editing.

Clicking on *Close* closes the memory window, canceling your last pending changes, if any.

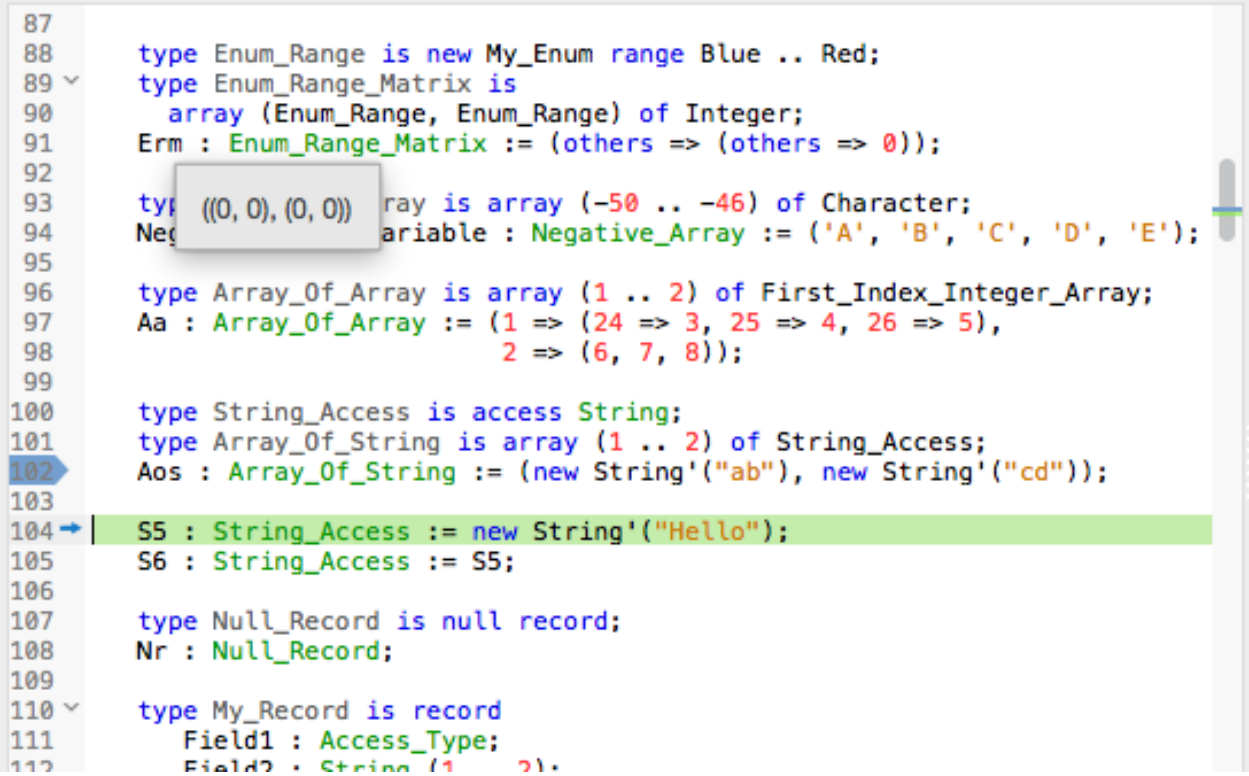
8.7 Using the Source Editor when Debugging

When debugging, the left area of each source editor provides the following information:

Current line executed

The line about to be executed by the debugger is highlighted in green (by default), and a green arrow is displayed on its left side.

Lines with breakpoints The line number (if present, otherwise the first few pixels) is highlighted with a background color for lines where breakpoints have been set. Add or delete breakpoints by clicking on the line number. These breakpoints can be set or unset even when no debugger is running.



The second area in the source editor is a text window on the right that displays the source files, with syntax highlighting. If you hold the pointer over a variable, GPS displays a tooltip showing the value of that variable. Disable these automatic tooltips using the preferences menu.

At all times, the contextual menu of the source window contains a *Debug* submenu providing some or all of the entries below. These entries are dynamic and apply to the entity under the pointer (depending on the current language). In addition, if you have made a selection in the editor, the text of the selection is used instead. This allows you to easily display complex expressions (for example, you can add comments to your code with expressions you want to display in the debugger).

- *Debug → Graph Display *selection**

  Displays the selection (or by default the name under the pointer) in the data window. GPS automatically refreshes this value each time the process state changes (e.g after a step or a next command). To freeze the display, click on the corresponding icon in the browser or use the contextual menu for that item (see *The Data Window*).

- *Debug → Graph Display *selection*.all*

  Dereferences the selection (or by default the name under the pointer) and displays the value in the data browser.

- *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 pointer. This menu is always enabled, even when no debugger is started. This means that you can prepare breakpoints while working on the code, before you even start the debugger.

- *Set Breakpoint on *selection**

  Sets a breakpoint at the beginning of the subprogram named *selection*. This menu is always enabled, even when no debugger is started. This means that you can prepare breakpoints while working on the code, before you even start the debugger.

- *Continue Until Line *xx**

  Continues execution (the program must have been started previously) until it reaches the specified line.

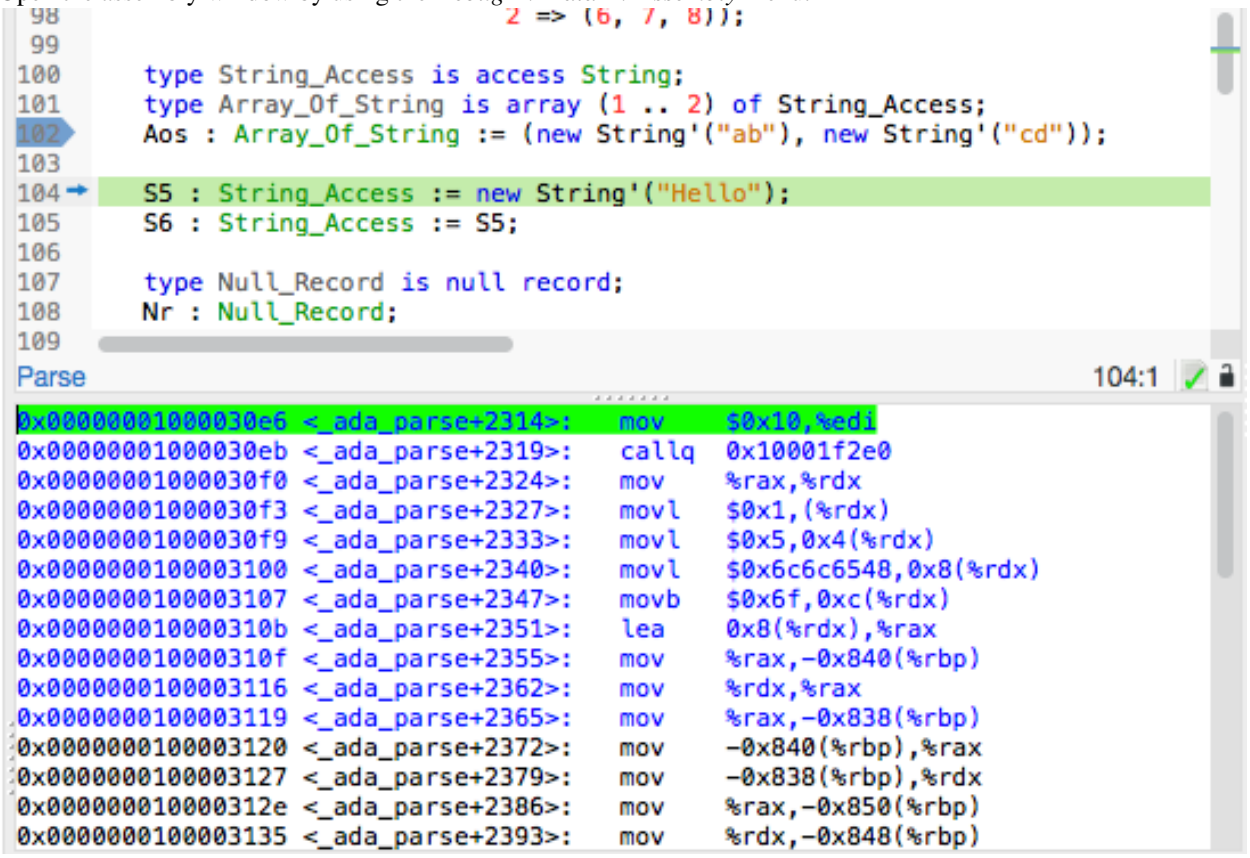
- *Show Current Location*

  Jumps to the current line of execution. This is particularly useful after navigating through your source code.

8.8 The Assembly Window

It is sometimes convenient to look at the assembly code for the subprogram or source line you are currently debugging.

Open the assembly window by using the *Debug* → *Data* → *Assembly* menu.



The current assembler instruction is highlighted on the left with a green arrow. The instructions corresponding to the current source line are highlighted (by default in red). This allows you to easily see where the program counter will point after you press the *Next* button on the tool bar.

Move to the next assembler instruction using the *Nexti* (next instruction) button in the tool bar. If you choose *Stepi* instead (step instruction), it steps into any subprogram being called by that instruction.

For efficiency purposes, GPS only displays a small part of the assembly code around the current instruction. Specify how many instructions are displayed in the preferences dialog. Display the instructions immediately preceding or following the currently displayed instructions by pressing one of the `Page up` or `Page down` keys or using the contextual menu in the assembly window.

A convenient complement when debugging at the assembly level is the ability to display the contents of machine registers. When the debugger supports it (as **gdb** does), select the *Debug* → *Data* → *Display Registers* menu to get

an item in the data browswer that shows the current contents of each machine register and that is 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* menu, enter something like:

```
output /x $eax
```

in the field and select toggle button *Expression is a subprogram call*. This creates a new browser item that is refreshed every time the value of the register (in this case **eax**) changes.

8.9 The Debugger Console

The debugger console is the text window located at the bottom of the main window. It gives you direct access to the underlying debugger, to which you can send commands (you need to refer to the underlying debugger's documentation, but usually typing "help" will gives you an overview of the available commands).

If the underlying debugger allows it, pressing `Tab` in this window provides completion for the command being typed (or its arguments).

Additional commands are defined here 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):

- tree display expression

  This command displays the value of the expression in the *Variables* view. The `expression` should be the name of a variable, or any expression matching the source language of the current frame (for instance `A(0).Field`).

- tree display *command*

  This command executes the gdb command, and displays the result in the *Variables* view. The `command` should be an internal debugger command, for instance `info local`.

graph (print|display) expression [dependent on display_num] [link_name name] [at x, y] [num num]

Create a new item in the browser showing the value of `Expression`, which is the name of a variable, or one of its fields, in the current scope for the debugger. The command *graph print* creates a frozen item, one that is not automatically refreshed when the debugger stops, while `graph display` displays an item that is automatically refreshed.

The new item is associated with a number displayed in its title bar. This number can be specified with the `num` keyword and can be used to create links between the items, using the second argument to the command, `dependent on`. By specifying the third argument, the link itself (i.e. the line) can be given a name that is also displayed.

graph (print|display) 'command'

Similar to the above, except you use it to display the result of a debugger command in the browser. For example, using **gdb**, if you want to display the value of a variable in hexadecimal rather than the default decimal, use a command like:

```
graph display `print /x my_variable`
```

This evaluates the command between back-quotes every time the debugger stops and displays the result in the browser. The lines that have changed are automatically highlighted (by default, in red).

graph (enable|disable) display display_num [display_num ...]

Change the refresh status of items in the canvas. As explained above, items are associated with a number visible in their title bar.

The **graph enable** command forces the item to be refreshed automatically every time the debugger stops and **graph disable** freezes the item, preventing its display from being changed.

graph undisplay display_num

Remove an item from the browser.

8.10 Customizing the Debugger

GPS is a high-level interface to several debugger backends, in particular **gdb**. Each backend has its own advantages, but you can enhance the command line interface to these backends through GPS by using Python.

This section provides a short such example whose goal is to demonstrate the notion of an "alias" in the debugger console. For example, if you type just "foo", it executes a longer command, such as one displaying the value of a variable with a long name. **gdb** already provides this feature through the **define** keywords, but here we implement that feature using Python in GPS.

GPS provides an extensive Python API to interface with each of the running debuggers. In particular, it provides the function "send", used to send a command to the debugger and get its output, and the function "set_output", 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 types 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 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 is sent to the debugger through **execute_alias**.

The real work is done by *debugger_commands*. If you execute the **alias** command, it defines a new alias. Otherwise, if you type the name of an alias, we want to execute that alias. And if not, we let the underlying debugger handle that command.

After you copied this example in the `$HOME/.gps/plug-ins` directory, 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** or **tree display** commands so the value of the variable is displayed in the data window, for example:

```
(gdb) graph display `foo`
(gdb) tree display `foo`
```

You can also program other examples. You could write complex Python functions, which would, for example, query the value of several variables and pretty-print the result. You can call any of these complex Python functions from the debugger console or have it called automatically every time the debugger stops via the **graph display** command.

8.11 Command line interface

GPS is still running the standard gdb underneath. So any command that you might be used to run in gdb can also be executed from the *Debugger Console*.

In particular, gdb has a feature where it reads initialization commands from a *.gdbinit* configuration file. Here are some pieces of information if you would like to use such files:

- When **gdb** starts, the current directory (which is where you should put your *.gdbinit* file is the environment's current directory. GPS doesn't override it. In general, this will also be the directory from which you started GPS itself. You can type:

  ```
  (gdb) pwd
  ```

  in the debugger console to find out exactly what the directory is.

- **gdb** always loads the global configuration *.gdbinit* in your home directory. It can also load a *.gdbinit* from the current directory, but this feature is disabled by default for security reasons to avoid malicious scripts.

  To enable the local *.gdbinit*, you will need to create the global one as well, with a contents similar to:

  ```
  add-auto-load-safe-path  <your directory>
  set auto-load local-gdbinit
  ```

  If you feel safe, you can replace "<your directory>" with "/" to always allow it on your system.

CHAPTER

NINE

VERSION CONTROL SYSTEM

Version control systems (VCS) are used to keep previous versions of your files, so that you can refer to them at any point in the future.

GPS provides integration with a number of such systems. It tries to provide a similar GUI interface for all of them, while preserving their specific vocabulary and features.

9.1 Setting up projects for version control

GPS does not come with any version control system. Instead, it expects that you already have such a system installl on your machine. In some cases, it is able to automatically recognize them. In other cases, you will need to edit your project file as described below.

GPS has built in support for the following VCS systems:

- *None*

  Disables version control support in GPS:

```
project Default is
   package IDE is
      for VCS_Kind use "None";
   end IDE;
end Default;
```

- *Auto* (default)

  Let GPS guess the correct version control system:

```
project Default is
   package IDE is
      for VCS_Kind use "Auto";
   end IDE;
end Default;
```

- *CVS*

  The Concurrent Version System. To use this, you must have a **patch** tool, which usually comes with CVS. GPS is automatically able to recognize that your project is using this system, but looking for a `CVS` directory in the root directory of your project. You can also force it by setting the following in your project:

```
project Default is
   package IDE is
```

```
      for VCS_Kind use "CVS";
   end IDE;
end Default;
```

This can of course be done via the graphical project editor in GPS.

- *Subversion*

  The Subversion version control system. As for CVS, GPS will automatically recognize that your project is using subversion by looking for a `.svn` directory in the root directory of your project. You can also force it by setting the following in your project:

```
project Default is
   package IDE is
      for VCS_Kind use "Subversion";
   end IDE;
end Default;
```

- *git*

  Distributed fast source code management. Again, GPS will automatically recognize this by looking for a `.git` directory in your project, but you can force this with:

```
project Default is
   package IDE is
      for VCS_Kind use "git";
   end IDE;
end Default;
```

Previous versions of GPS supported a larger range of systems, but these have not been ported to the new code yet. Please let us know whether there is interest in doing so:

- *ClearCase*
- *ClearCase Native*
- *Mercurial*

Most of the version control code in GPS is generic, and customized for each system via one small python plugin. As a result, it should be possible to add support for other systems, by creating such plugins. Take a look at the files in the directory `prefix/share/gps/plug-ins/vcs2` in your GPS install.

As mentioned before, GPS automatically attempts to guess the correct version system you are using. This is similar to having the following declaration in your project:

```
project Default is
   package IDE is
      for VCS_Kind use "auto";
   end IDE;
end Default;
```

Note: you must be sure VCS commands can be launched without needing to enter a password.

In general, you will be have loaded one root project in GPS, but this is turn imports many other projects. Each of these can use its own version control system (so you can mix git and subversion for instance if your sources come from different places), or even the same system but for a different repository (so you could be cloning multiple git repositories).

If you have a setup with multiple systems, GPS will show special buttons in the local toolbars of the views to let you select which is the one to use for the operations (fetching the history, committing,...) These operations only apply to one system at a time, you cannot do a single commit with files that belong to multiple systems (although you can do a single commit for files that belong to multiple projects, provided these projects all use the same system and same repository).

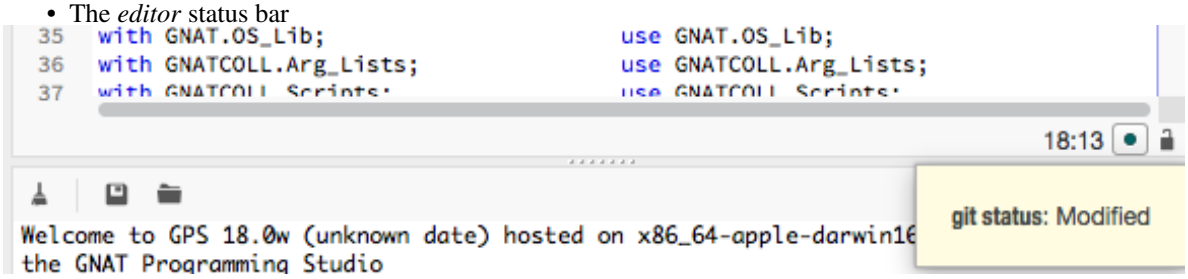
9.2 Finding file status (*Project* view)

Most of the times, you will be using GPS on a project that already exists and for which version control has already been setup.

For such a project, the first task is to find out what is the status of the files, i.e. whether they are locally modified, up-to-date, whether you have created new files but not yet added them to version control, and so on.

To make this convenient, GPS displays this information in a number of places, via a small icon and appropriate tooltips.

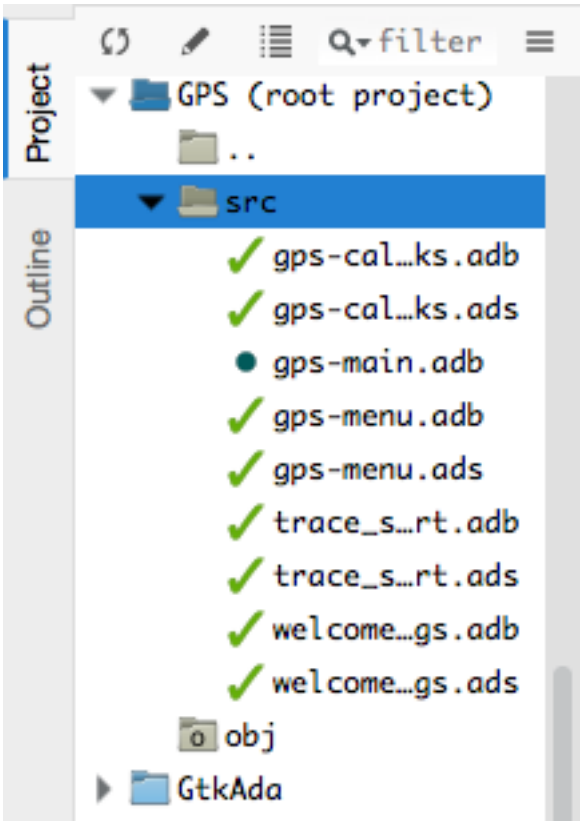
- The *editor* status bar



  Whenever you are editing a file, GPS displays a small icon in the status bar that indicates its current status as seen by GPS. If you hover the mouse, it will show a textual status. In this screenshot, the file has been modified locally, but not committed yet into the version control system (git in this case).

  Clicking on this icon will change to the *The VCS Perspective*.

- The *Project* view

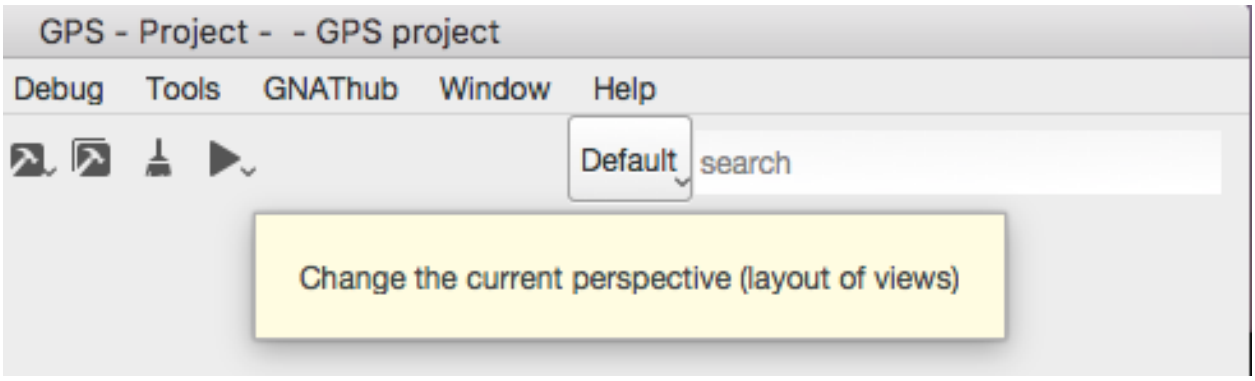


The *Project* view is convenient to see all your source files, grouped by projects and optionally directories. GPS will show the same icon as the editor next to the name of each file, so that you can easily see their status. Again, the tooltip would show the textual status.

- The *Files* view

  This view is similar to the *Project* view, but groups files as they are organized on the disk. GPS will try to guess the best system here, but there might be ambiguities when the same directory is shared among multiple projects which use a different VCS system or repository. We do not recommend this setup.

9.3 The VCS Perspective



To display all pertinent information on your files, GPS uses multiple views, as described below. Although you can open any of them whenever you want, via the *Tools* → *Views* or *VCS* menus, the most convenient is to select the VCS perspective.

This perspective was created to show all VCS related views, and hide unrelated views. As for all GPS perspectives, you can modify the way it looks, which views are displayed,... simply by opening new views or moving them around while this perspective is selected.

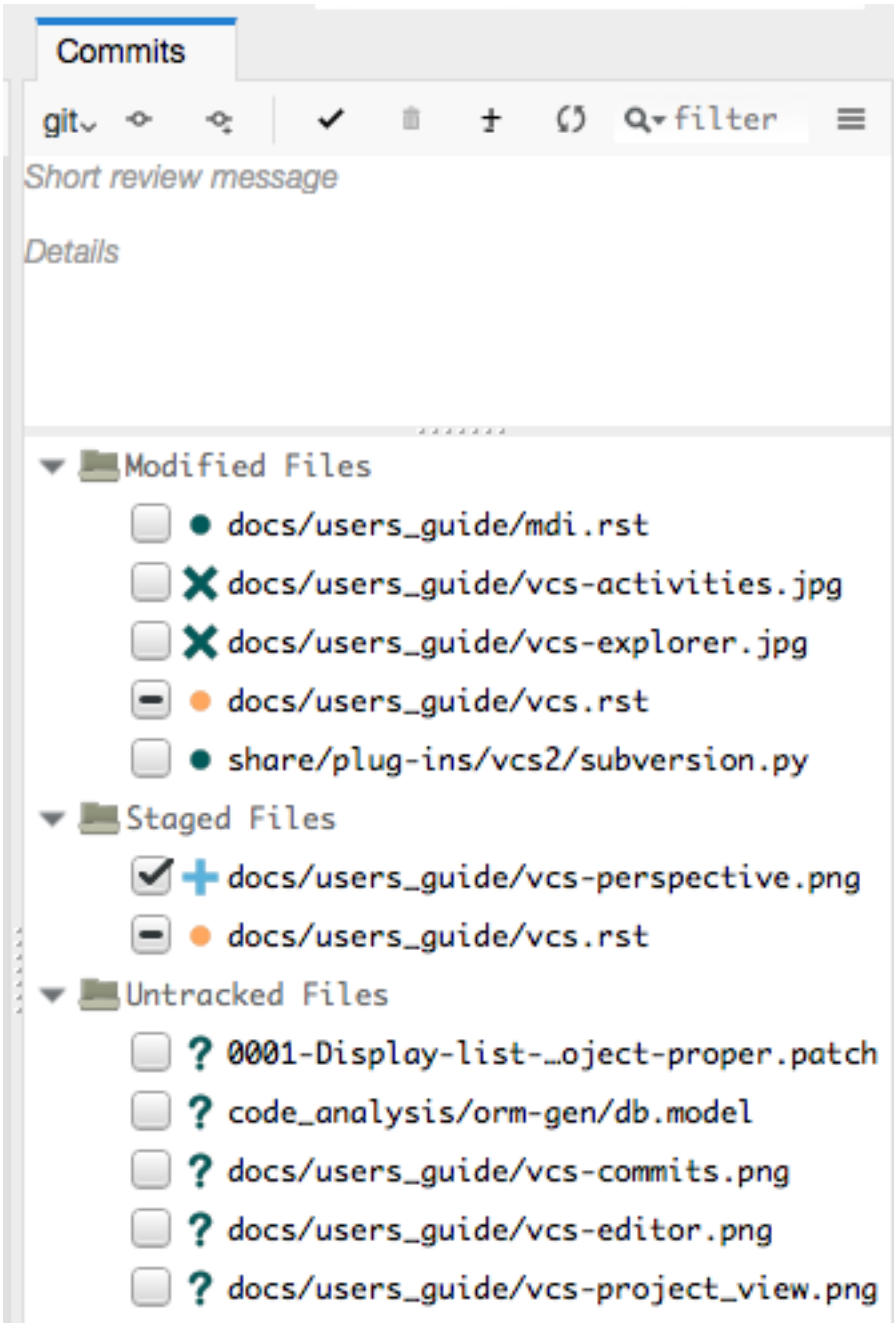
See also *Perspectives* for more information on how to manipulate them.

There are multiple ways to switch to this VCS perspective: as always, you can use the toolbar's perspective selector, or the *Window* → *Perspectives* menu. The most convenient might be to click on the VCS status icon at the bottom of each editor.

In all of these cases, GPS will change which windows are displayed on the screen. It will preserve your editors, but close all other views, and instead show the following:

- The *Project* view, used to check the status of all files
- The *Commits* view (*The Commits view*), used to select which files should be committed, and do the actual commit
- The *History* view (*The History view*), to view all past commits
- The *Branches* view (*The Branches view*), to view various pieces of information about your repository, depending on which system you use.

9.4 The Commits view



The purpose of this view is to let you prepare and then commit your files.

9.4.1 Viewing modified files

The view lists all files in your project, to the exception of up-to-date files (i.e. those files that have been checked out, and never touched locally), and ignored files (i.e. those files for which you have explicitly told the VCS that you will never want to commit them).

By default, they are organized into three sections:

- Staged files

  These files will be part of the next commit (see below)

- Modified but unstaged files

  These are locally modified files, which will not be part of the next commit, unless you stage them.

  It is possible for a file to be in both groups (on the screenshot, this is the case for `vcs.rst`), when it had been modified, then staged, then further modified. If you are using git, the later modification have not been staged for commit, and git will only commit the first set of changes. Other systems like CVS and Subversion will always apply all current change to the file, no matter whether they were done before or after the staging operation.

- Untracked files

  These are files found in your directory, but that are unknown to the VCS. Sometimes these files should simply be ignored and never committed, but sometimes they will be newly created files that you should stage to include them in the next commit.

Various local configurations can be selected to change what is displayed in this view, take a look at the menu and the tooltips.

9.4.2 Committing files

Committing is always a three step process in GPS (this is exactly what git does natively, but also provides more flexibility foro over systems).

- First, you need to select which files will be part of the next commit. It is possible that you have been modifying unrelated source files, which you do not want to commit yet.

  This is called **staging** the files, and can be performed simply by clicking in the checkbox next to the file's name, or by selecting multiple files at once and then clicking on the "stage" toolbar button.

  Staging files can be done at any point in time, not necessarily just before you commit. You can also stage files, exit GPS then restart, and GPS will remember which files had been staged.

- The second step is to provide a commit message. GPS will not let you do a commit without an actual message (most VCS systems don't either). You can enter any message in the editor at the top of the *Commits* view.

  With git, the standard is to have one single short line first then an empty line, then a more extensive message. We recommend similar approaches for other systems. That first line is the one that will be displayed in the *History* (*The History view*).

  Just like for staging files, you can edit this message at any point in time, so it is a useful way to explain your changes as you do them, even if you intend to do further changes before the actual commit.

- Finally, you just press the *Commit* button in the local toolbar. GPS will ask the VCS to do the actual commit, and then will refresh all views. All files that were modified and staged before are shown as no longer modified, for instance.

9.4.3 Actions in the Commits view

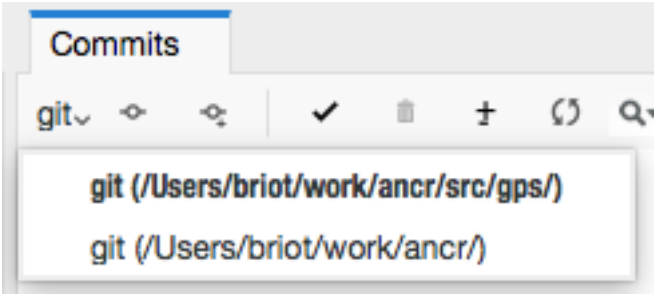
Double-clicking on a file will open an editor for that file.

Clicking and keeping the mouse pressed on a file will open a *Diff* view showing the current changes to the file.

9.4.4 The Commits view local toolbar

The commits view contains a number of buttons in its toolbar. The exact set of buttons will depend on which VCS you are using, but here is some buttons that will be useful in a lot of cases:

- On the left of the toolbar is a button to select the current VCS system, in case your projects uses multiple such systems, or multiple repositories with the same system. The commit and staging will always be done for the current system only.



  This button is hidden if you are using a single VCS system for all your projects.

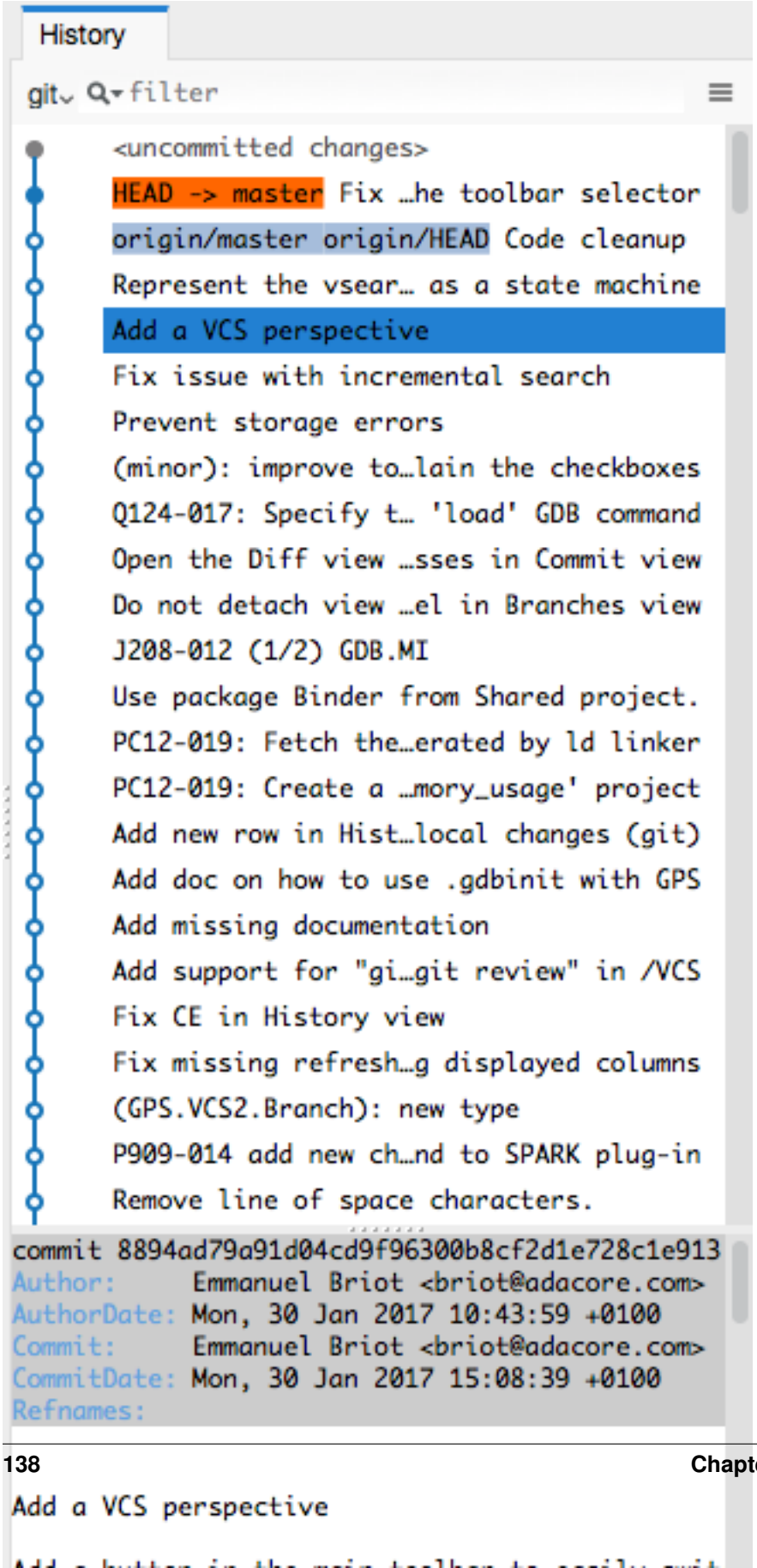
- A button is provided to undo all local changes.

  After confirmation, GPS will revert to the last commit, and cancel all changes you might have done locally. This works for all supported VCS.

- A button to refresh the contents of all VCS views

  This button is not needed if you do all operations from GPS, including editing files. But if you do operations outside of GPS's control, you will need to manually resynchronize the views with what's really in your VCS.

9.5 The History view



The purpose of this view is to show all past changes that you have done with your VCS.

This view is divided into three parts:

9.5.1 List of all past commits

For each commit, GPS displays the first line of the commit message. Optionally, you can configure the view to also show the author, the date, and the unique identifier for these commits.

Depending on the VCS in use, GPS will also show the name of the branches associated with these commits, as well as specific tag names that might have been set.

In particular, git shows the contents of all active branches, so the history is not so linear, and there is a wealth of information to show how the branches were split and joined in the past.

When this is too much information, you can use the local configuration menu to only show the details for the current branch.

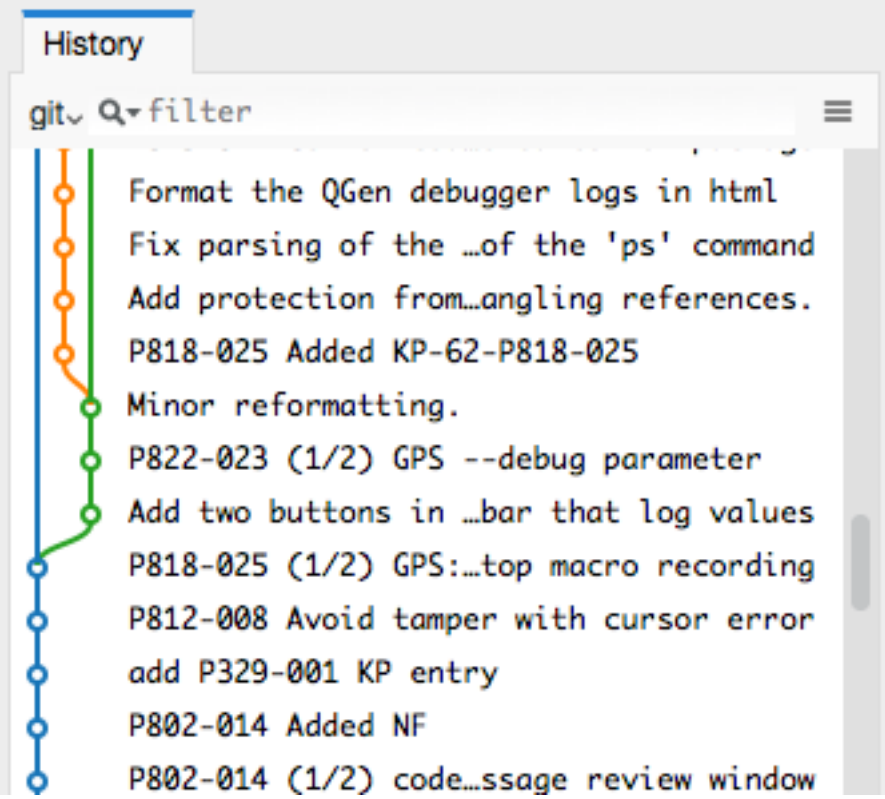
A special line (added at the top in the screenshot above) is displayed in gray when there are local uncommitted changes in your working directory.

By default, GPS only shows the first 2000 commits. If you want to see more, scroll to the bottom and click on the *Show more* buttons to download more entries.

9.5.2 Graph of past history

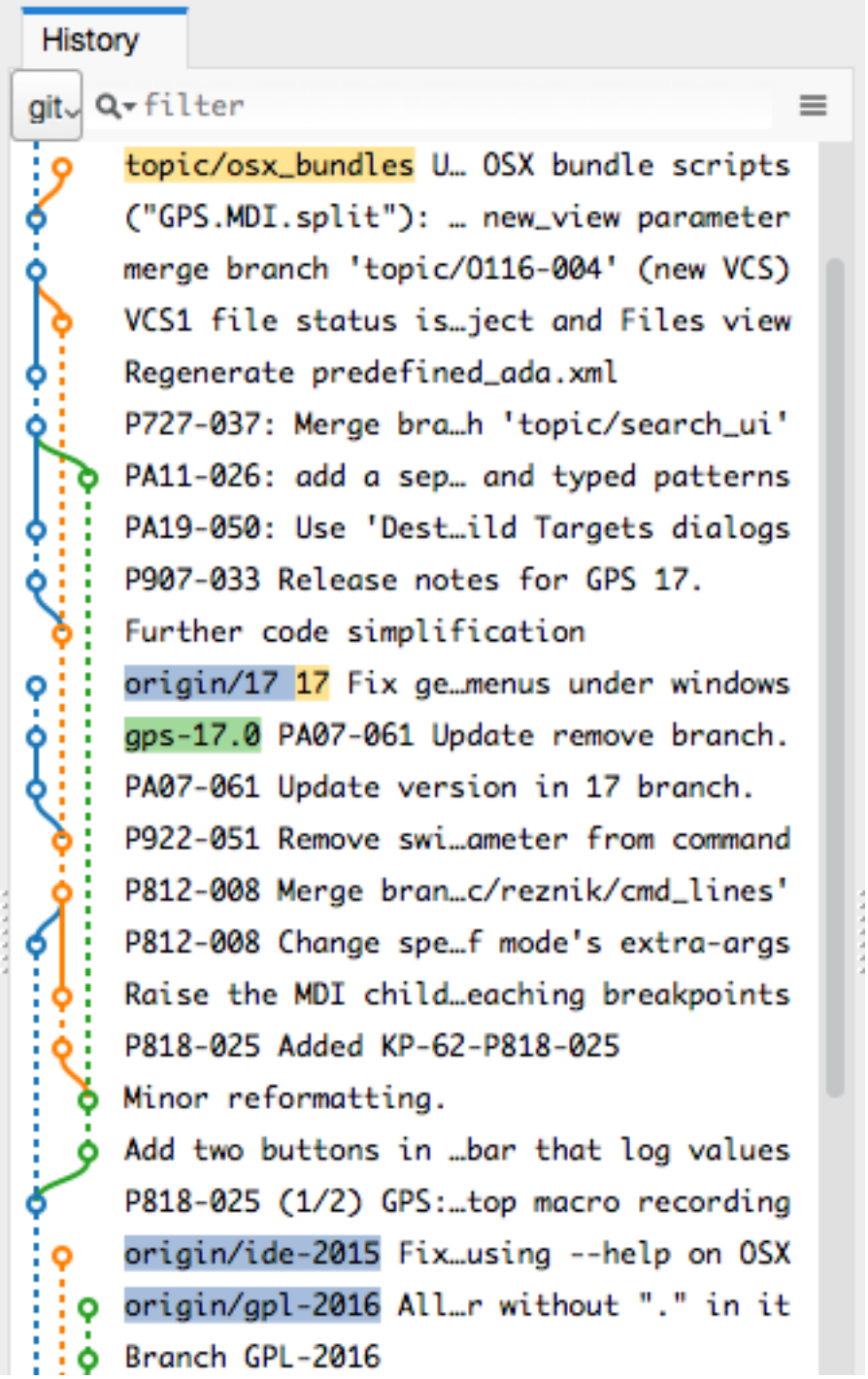
Next to the list of commits is a graph showing their relationships. Most of the times, this history is fairly linear, with each commit having one parent commit, and followed by another commit.

But with some VCS like git, people might chose to use a lot more branches. They create a new branch to work on a specific feature, then merge it into the master branch when the feature is completed. It can become harder to follow the history in such a case.



In this case, the graph becomes more useful, as shown in the screenshot above.

But using the local configuration menu, you can also chose to only show commits that are related to branches (either because they are the beginning of a branch, or because they are a merge of two branches, or because they have a special name (tag or branch name) associated with them. All commits with a single parent and single child are hidden.

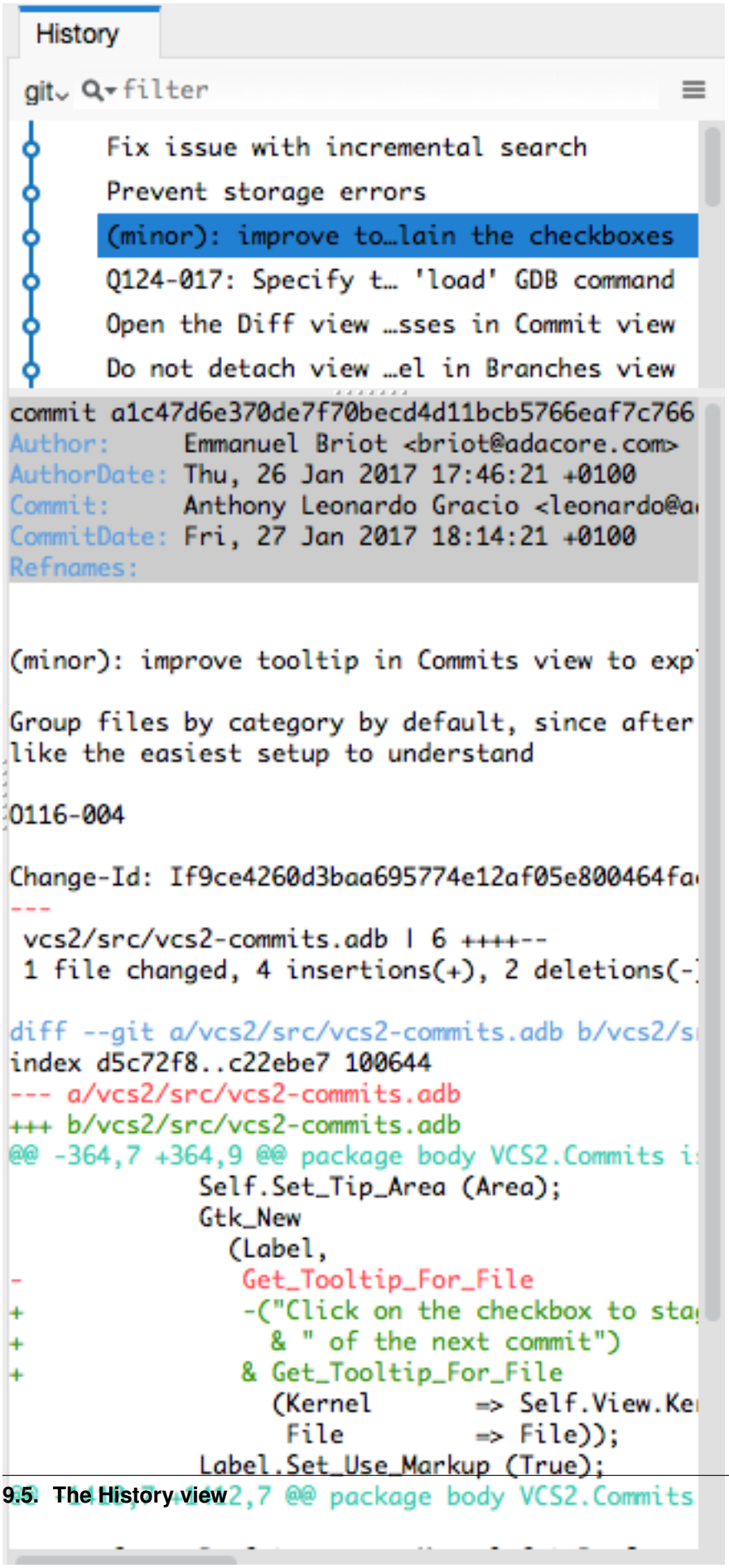


When you are using git, commits that have not yet been pushed to the remote branch will be displayed with a filled circle to help you find out whether you need to push.

9.5.3 Details on selected commits

Whenever you select one or more commits, GPS will download their details and show those at the bottom of the *Commits* view.

These details are those provided by the VCS, and generally include the author and date of the commit, as well as the full commit message and diff of what changes were made.


History
git
filter
Fix issue with incremental search
Prevent storage errors
(minor): improve to…lain the checkboxes
Q124-017: Specify t… 'load' GDB command
Open the Diff view …sses in Commit view
Do not detach view …el in Branches view
commit a1c47d6e370de7f70becd4d11bcb5766eaf7c766
Author: Emmanuel Briot <briot@adacore.com>
AuthorDate: Thu, 26 Jan 2017 17:46:21 +0100
Commit: Anthony Leonardo Gracio <leonardo@a
CommitDate: Fri, 27 Jan 2017 18:14:21 +0100
Refnames:
(minor): improve tooltip in Commits view to exp
Group files by category by default, since after
like the easiest setup to understand
0116-004
Change-Id: If9ce4260d3baa695774e12af05e800464fa
---
vcs2/src/vcs2-commits.adb | 6 ++++--
1 file changed, 4 insertions(+), 2 deletions(-)
diff --git a/vcs2/src/vcs2-commits.adb b/vcs2/s
index d5c72f8..c22ebe7 100644
--- a/vcs2/src/vcs2-commits.adb
+++ b/vcs2/src/vcs2-commits.adb
@@ -364,7 +364,9 @@ package body VCS2.Commits i
Self.Set_Tip_Area (Area);
Gtk_New
(Label,
- Get_Tooltip_For_File
+ -("Click on the checkbox to sta
+ & " of the next commit")
+ & Get_Tooltip_For_File
(Kernel => Self.View.Ke
File => File));
Label.Set_Use_Markup (True);
@@ -444,2,7 @@ package body VCS2.Commits

The diff is syntax highlighted to make it more readable.

9.6 The Branches view

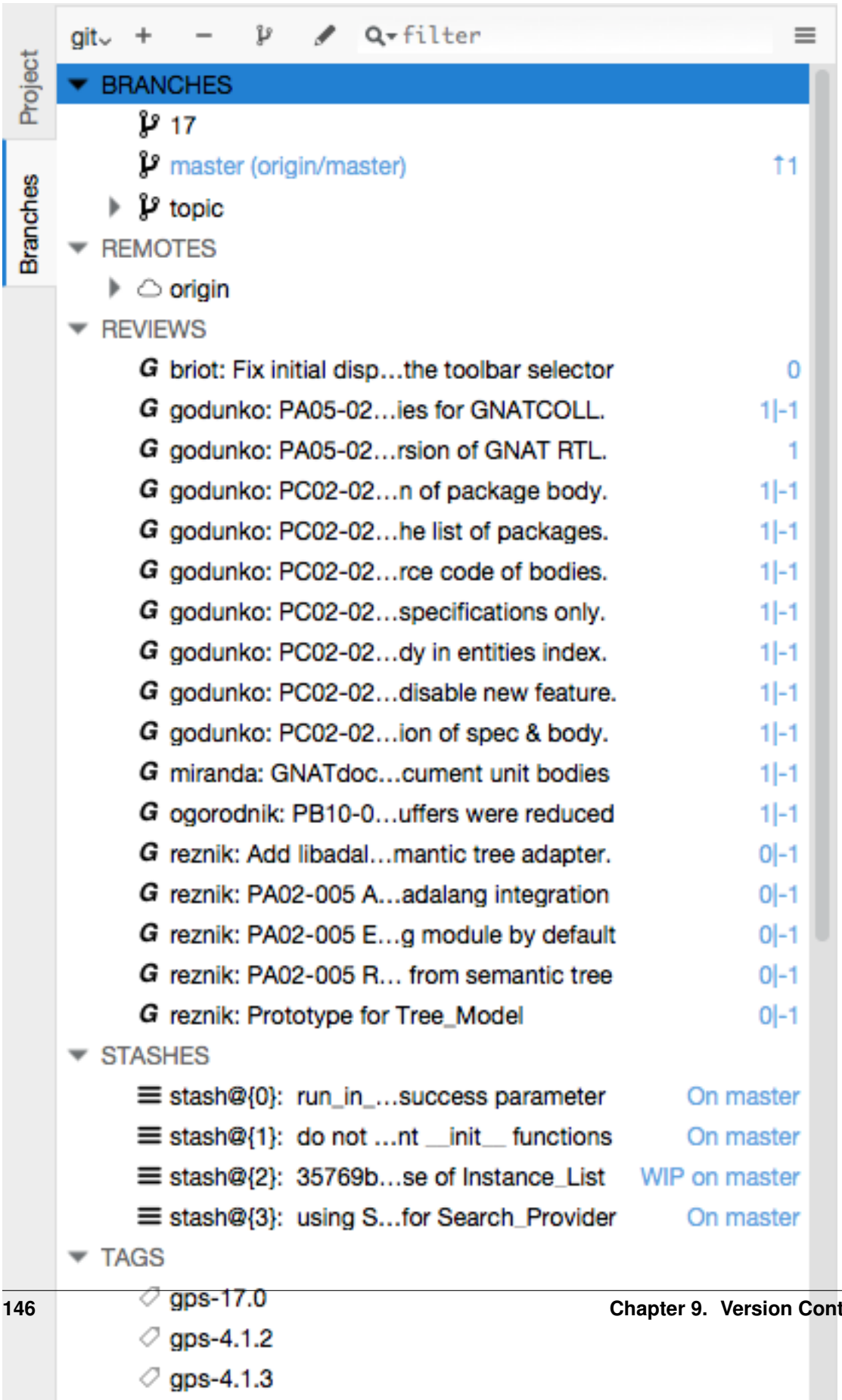
The purpose of this view is to display various pieces of information that are specific to each VCS.

Most notably, it will let you interact with branches.

Various actions are possible in this view, all of which depends on the VCS and which specific section of the view you interact with. Please check the tooltips that are shown when you leave the mouse over a line for a brief while to see what actions are possible. The actions are done via one of the following means:

- double-clicking on a line. This is the same as using the corresponding toolbar button.
- a long click on a line (click and then leave the mouse pressed for a short while). This is the same as using the *rename* toolbar button.
- clicking on the *[+]* button in the toolbar.
- clicking on the *[-]* button in the toolbar.

9.6.1 Git and the Branches view



The screenshot above is for git. In this case, GPS displays the following pieces of information:

- List of local branches

  For each branch, GPS displays the number of commits that have not yet been pushed to the remote branch, and conversely the number of changes that have been made in the remote branch but not yet applied to the local branch.

  Double-clicking on any of them will check it out and make it the current branch. If you have locally modified files at that time, git might refuse to do the checkout, and the error message will be displayed in GPS's *Messages* view.

  A long click on any of the branch names will let you rename the branch.

  A click on *[+]* will create a new branch, starting from the selected one.

  A click on *[-]* will remove the selected branch if it is not the current one.

- List of remote branches

  These are the branches that exist in the git repository, that you can checkout locally by double-clicking on them. The branches are grouped by the name of the remote repository that contains this branch, since git is a distributed system.

  You can also delete a remote branch by clicking on *[-]*.

- List of Gerrit reviews

  If you are doing code reviews via Gerrit, GPS is able to download the list of patches pending review, as well as their current scores.

  Double-clicking on any of the patch will open the Gerrit page in a web browser.

  Clicking on *[+]* will cherry pick the patch and apply it to the local working directory.

  If you are not using Gerrit, this category will not be displayed.

- List of stashes

  In git, stashes are a way to temporary move away local changes to get back to a pristine working directory, without losing your current work.

  GPS displays the list of all stashes, and lets you create new stashes by clicking on *[+]* when the *STASHES* line is selected.

  Clicking on *[-]* will drop the selected stash, and you will lose the corresponding changes.

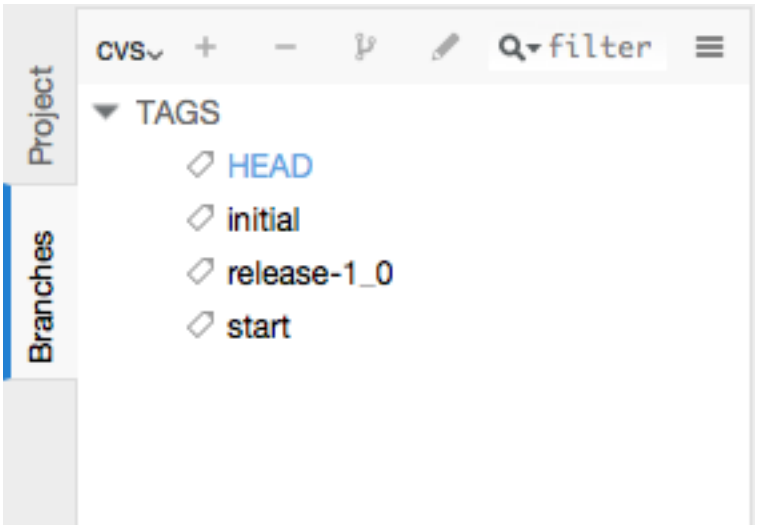
  Double-clicking on a stash will reapply it to the current working directory. It will not drop it though, so that you can also apply it to another branch.

- List of tags

  All tags that have been applied in your repository are also listed. You can create new tags by selecting the *TAGS* line and clicking on *[+]* line.

  You can remove tags by clicking on *[-]*.

9.6.2 CVS and the Branches view



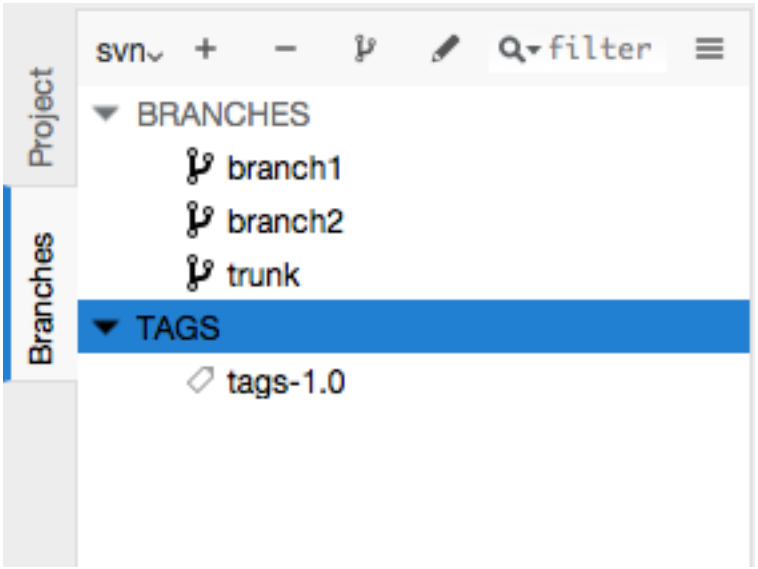
The screenshot above is for CVS. GPS displays far fewer information than for git, and only shows the tags.

Double-clicking on any of the tag will check it out in the working directory.

Clicking on *[-]* deletes the selected tag.

Clicking on *[+]* while the *TAGS* line is selected will create a new tag.

9.6.3 Subversion and the Branches view



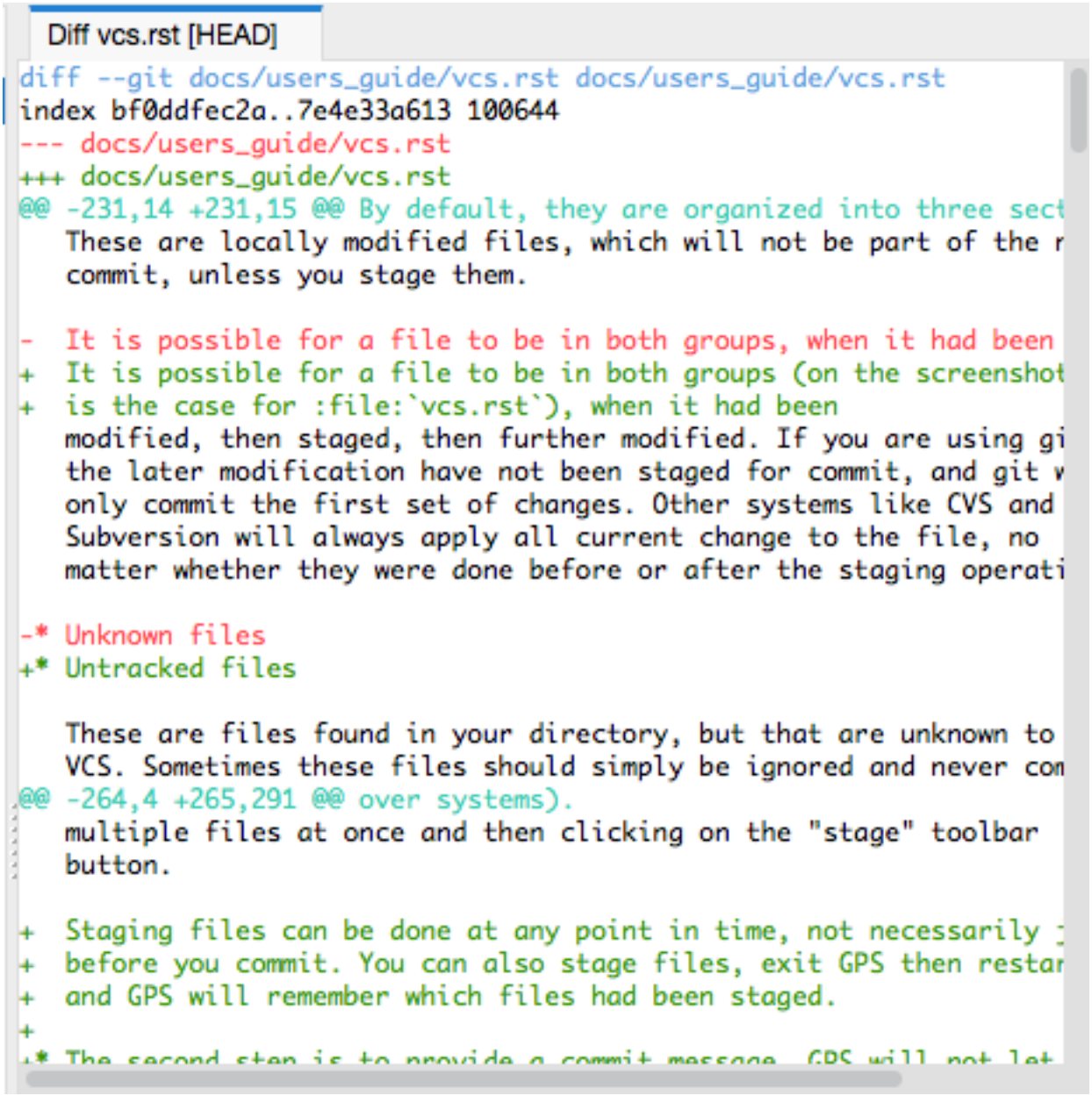
GPS assumes a standard organization of the subversion repository, namely that there are three top-level directories:

```
<repository>/trunk/project/
<repository>/tags/project/
<repository>/branches/project/
```

If this is the organization you are also using, GPS is able to show the list of tags and branches in the *Branches* view.

You can checkout a specific tag or branch by double-clicking on it.

9.7 The Diff View



This view shows a simple color highlighted diff. The screenshot shows the changes currently done to this document...)

This view is opened either by long clicking on a file name in the *Commits* view (*The Commits view*), or by selecting the menu *VCS → Show all local changes*.

CHAPTER

TEN

TOOLS

10.1 The Tools Menu

The *Tools* menu gives access to additional tools. If an option is disabled, it means it is a tool that is not yet available.

The list of active entries includes:

- *Views*
  - *Bookmarks*

    See *The Bookmarks view*.
  - *Call Trees*

    Open a tree view displaying function callers and callees. See *Callgraph browser*.
  - *Clipboard*

    See *The Clipboard view*.
  - *Coverage Report*

    See *Coverage Report*.
  - *Files*

    Open a file system explorer in the left area. See *The Files View*.
  - *File Switches*

    See *File Switches*.
  - *Outline*

    Open a view of the current source editor. See *The Outline view*.
  - *Messages*

    Open the *Messages* view. See *The Messages view*.
  - *Project*

    See *The Project view*.
  - *Remote*

    See *Setup a remote project*.
  - *Scenario*

    See *Scenarios and Configuration Variables*.

  - *Tasks*

    See *The Tasks view*.

  - *Windows*

    Open a view containing all currently opened files. See *The Windows view*.

- *Browsers*
  - *Call Graph*

    See *Callgraph browser*.

  - *Dependency*

    See *The Dependency Browser*.

  - *Elaboration Cycles*

    See *The Elaboration Circularities browser*.

  - *Entity*

    See *The Entity browser*.

- *Coding Standard*

  See *Coding Standard*.

- *Compare*

  See *Visual Comparison*.

- *Consoles*
  - *Python*

    Open a Python console to access the Python interpreter. See *The Python Console*.

  - *OS Shell*

    Open an OS (Windows or Unix) console, using the environment variables **SHELL** and **COMSPEC** to determine which shell to use. See *The Python Console*.

    On Unix, this terminal behaves a lot like a standard Unix terminal, but you need to make sure your shell will output all necessary information. In some cases, the configuration of your shell (`.bashrc` if you are running **bash**, for example) deactivates echoing what you type. Since GPS is not writing anything on its own, but just showing what the shell is sending, you need to ensure your shell always echos what you type. Do this by running the command:

    ```
    stty echo
    ```

    in such cases. Normally, you can safely put this in your `.bashrc`

  - *Auxiliary Builds*

    Open the console containing auxiliary builds output. Only output from automated cross-reference generation is currently sent to this console. See *Working with two compilers*.

- *Coverage*

  See *Code Coverage*.

- *Documentation*

  See *Documentation Generation*.

- *GNATtest*

  See *Working With Unit Tests*.

- *Stack Analysis*

  See *Stack Analysis*.

- *Macro*

  See *Recording and replaying macros*.

- *Metrics*

  See *Metrics*.

- *Interrupt*

  Interrupt the last task launched such as a compilation or VCS operation.

10.2 Coding Standard

Use the *Coding Standard* menu to edit your coding standard file and run it against your code to verifiy its compliance with the coding standard. This file is the input to the **gnatcheck** tool. You can also use the contextual menu to check the conformance of a particular project or source file against a coding standard.

Access the Coding standard editor using the *Tools* → *Coding Standard* → *Edit Rules File* menu. Select either an existing coding standard file or create a new one. The editor adapts itself to the version of **gnatcheck** on your local machine.

GPS summarizes the rules currently in use at the bottom of the editor. Once all rules are defined, check the box *Open rules file after exit* to manually verify the created file. Once you have created the coding standard file, set it as the default coding standard file for a project by going to the project editor, selecting the *Switches* tab, and specifying this file in the *Gnatcheck* section.

10.3 Visual Comparison

The visual comparison, available either from the VCS menus or the *Tools* menu, provides a way to graphically display differences between two or three files or two different versions of the same file.

The 2-file comparison tool uses the standard tool **diff**, available on all Unix systems. Under Windows, a default implementation is provided with GPS, called `gnudiff.exe`, but you may want to provide an alternate implementation, for example by installing a set of Unix tools such as Cygwin (http://www.cygwin.com). The 3-file comparison tool is based on the text tool **diff3**, available on all Unix systems. Under Windows, this tool is not provided with GPS, but is available as part of Cygwin.

GPS displays visual comparisons in either Side-by-Side or Unified mode. In Side-by-Side mode, GPS displays editors for the files involved in the comparison side by side. By default, GPS places the reference file on the left. In Unified mode, GPS does not open a new editor, but shows all the changes in the original editor. Unified mode is used only when comparing two files; when comparing three files, only Side-by-Side mode is available.

Lines in the file editors are highlighted with various colors. In side-by-side mode, only the right editor (for the modified file) has different colors. Each highlight color indicates a different type of line:

gray

All the lines in the reference (left) file.

yellow

Lines modified from the reference file. Small differences within one line are shown in a brighter yellow.

green

Lines not originally in the reference file but added to the modified file.

red

Lines present in the reference file but deleted from the modified file.

You can configure these colors in the preferences dialog.

Like all highlighted lines in GPS, the visual differences highlights are visible in the *Speed Column* at the left of the editors.

GPS adds blank lines in one editor in places corresponding to existing lines in the other editors and synchronizes vertical and horizontal scrolling between the editors involved in a visual comparison. If you close one of those editors, GPS removes the highlighting, blank lines, and scrolling in the other editors.

When you create a visual comparison, GPS populates the *Locations* view with the entries for each chunk of differences; use them to navigate between differences.

Editors involved in a visual comparison have a contextual menu *Visual diff* containing the following entries:

- *Recompute*

  Regenerate the visual comparison. Use this when you have modified one of the files in an editor by hand while it is involved in a visual comparison.

- *Hide*

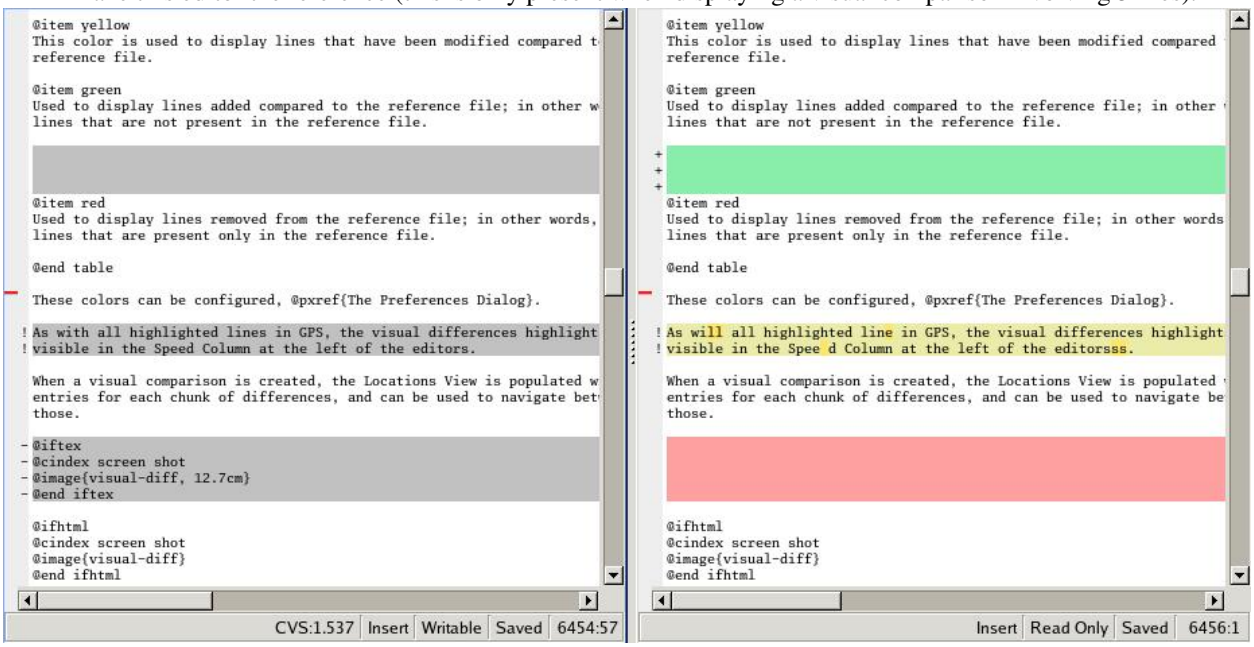
  Remove the highlighting corresponding to the visual comparison from all involved editors.

- *Close editors*

  Closes all editors involved in this visual comparison

- *Use this editor as reference*

  Make this editor the reference (this is only present when displaying a visual comparison involving 3 files).



10.4 Code Fixing

GPS provides an interactive mechanism to correct or improve your source code based on error and warning messages generated by the GNAT compiler. This capability is integrated with the *Locations* view (see *The Locations View*): when GPS can make use of a compiler message, it adds an icon on the left of the line.

If a wrench icon is displayed and you left-click on it, the code is fixed automatically, and you will see the change in the corresponding source editor. This occurs when a simple fix, such as the addition of a missing semicolon, is sufficient to resolve the error.

Right-click on the icon to display a contextual menu with text explaining the action that would be performed on a left-click. Displaying a contextual menu anywhere else on the message line provides an option called *Auto Fix*, giving you access to the same information. For the previous example of a missing semicolon, the menu contains an entry labelled *Add expected string ";"*. You can choose to *Apply to this occurrence* or *Apply to all similar errors*. The latter option applies the same simple fix to all errors that are the same, based on parsing the error message. The wrench icon is removed once the code change has been made.

For more complex errors where more than one change is possible, GPS displays a wrench icon with a blue plus sign. Clicking the icon displays a contextual menu listing the possible fixes. For example, this is displayed when an ambiguity in resolving an entity is reported by the compiler.

Right-clicking on a message with a fix opens a contextual menu with an entry *Auto Fix*. Fixes that can be applied by clicking on the wrench are also available through that menu. In addition, if GPS considers one of the fixes to be safe, it provides additional menu entries to apply fixes at multiple locations:

Fix all simple style errors and warnings

> Offered only when the selected message is a style warning or error. Fixes all other style warnings and errors for which a unique simple fix is available.

Fix all simple errors

> Fixes all errors messages for which a unique simple fix is available

10.5 Documentation Generation

GPS uses the GNATdoc tool to generate documentation from source files. See GNATdoc User's Guide for more information about this tool, including comment formatting and possibilities for customization.

Invoke the documentation generator from the *Tools -> Documentation* menu:

Generate Project

> Generate documentation for all files in the loaded project.

Generate Projects & Subprojects

> Generate documentation for all files in the loaded project and its subprojects.

Generate current file

> Generate documentation for the current file.

10.6 Working With Unit Tests

GPS uses **gnattest**, a tool that creates unit-test stubs as well as a test driver infrastructure (harness). It can generate harnesses for a project hierarchy, a single project or a package. Launch harness generation process from the *Tools* → *GNATtest* menu or a contextual menu.

After a harness project has been generated, GPS switches to it, allowing you to implement tests, compile and run the harness. You can exit the harness project and return to original project at any point.

10.6.1 The GNATtest Menu

The *GNATtest* submenu is found in the *Tools* global menu and contains the following entries:

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 tests that have have never been modified and list them in the *Locations* view. This menu is only active in the harness project.

Exit from harness project

Return from harness to original project.

10.6.2 The Contextual Menu

When relevant to the context, right-clicking displays GNATtest-related contextual menu entries. The contextual menu for a source file containing a library package declaration has a *GNATtest → Generate unit test setup for <file>* menu that generates the harness for that package. The contextual menu for a project, (see *The Project view*), has a *GNATtest → Generate unit test setup for <project>* menu that generates the harness for the entire project. The *GNATtest → Generate unit test setup for <project> recursive* menu generates a harness for whole hierarchy of projects. If a harness project already exists, the *GNATtest → Open harness project* menu opens the harness project.

While a harness project is open, you can simply navigate between the tested routine and its test code. Clicking on the name of a tested routine produces the *GNATtest → Go to test case*, *GNATtest → Go to test setup*, and *GNATtest → Go to test teardown* menus . The contextual menu for source files of test cases or setup and teardown code has a *GNATtest → Go to <routine>* menu to go to the code being tested.

10.6.3 Project Properties

You configure GNATtest's behavior through the GNATtest page in *The Project Properties Editor*.

10.7 Metrics

GPS provides an interface to the GNAT software metrics generation tool **`gnatmetric`**. Metrics can be computed for one source file, the current project, or the current project and all its imported subprojects

Invoke the metrics generator from the *Tools → Metrics* menu or 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 in the current project.

Compute metrics for current project and subprojects

Generate metrics for all files in the current project and subprojects.

10.7.2 The Contextual Menu

When relevant to the context, right-clicking displays metrics-related contextual menu entries. The contextual menu for a source file has an entry *Metrics for file* that generates the metrics for the current file. The contextual menu for a project (see *The Project view*) has an entry *Metrics for project* that generates the metrics for all files in the project.

After computing the requested metrics, GPS displays a new window in the left area showing the computed metrics in a hierarchical tree form, arranged first by files and then by scopes inside the files. Double-clicking any of the files or scopes opens the corresponding source location in the editor. GPS displays any errors encountered during metrics computation in the *Locations* view.

10.8 Code Coverage

GPS is integrated with **gcov**, the GNU code coverage utility. Within GPS, you can compute, load, and visualize code coverage information. You can do this for individual files, for each file of the current project, for individual projects in a hierarchy, or for the entire project hierarchy currently loaded by GPS.

Once computed and loaded, GPS summarizes the coverage information in a graphical report, formatted as a tree-view with percentage bars for each item, and uses it to decorate source code through line highlighting and coverage annotations.

You will find all coverage related operations in the *Tools* → *Coverage* menu. Before GPS can load coverage information, it must be computed, for example by using the *Tools* → *Coverage* → *Gcov* → *Compute coverage files* menu. After each coverage computation, GPS tries to load the needed information and reports errors for missing or corrupted `.gcov` files.

To produce coverage information from **gcov**, your project must be compiled with the **-fprofile-arcs** and **-ftest-coverage** switches, respectively the *Instrument arcs* and *Code coverage* entries in *The Project Properties Editor* and executed.

10.8.1 Coverage Menu

The *Tools* → *Coverage* menu has a number of entries, depending on the context:

- *Gcov* → *Compute coverage files*

  Generate the `.gcov` files for loaded projects that have been compiled and executed.

- *Gcov* → *Remove coverage files*

  Delete all the `.gcov` file for loaded projects.

- *Show report*

  Open a new window summarizing the coverage information currently loaded in GPS.

- *Load data for all projects*

  Load (or reload) coverage information for every project and subproject.

- *Load data for project 'XXX'*

  Load or re-load coverage information for the project *XXX*.

- *Load data for xxxxxxxx.xxx*

  Load (or reload) coverage information for the specified source file.

- *Clear coverage from memory*

  Remove all 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 will see a *Coverage* submenu containing the following options, depending on the type of entity selected. For example, if you click on a file, the options are:

- *Show coverage information*

  Display an annotation column on the left side of the current source editor to indicate which lines are covered and which are not. Lines that are not covered are also listed in the *Locations* view. See *The Locations View*.

- *Hide coverage information*

  Remove the annotation column from the current source editor and clear converage information from the *Locations* view.

- *Load data for xxxxxxxx.xxx*

  Load (or reload) coverage information for the specified source file.

- *Remove data of 'xxxxxxxx.xxx*

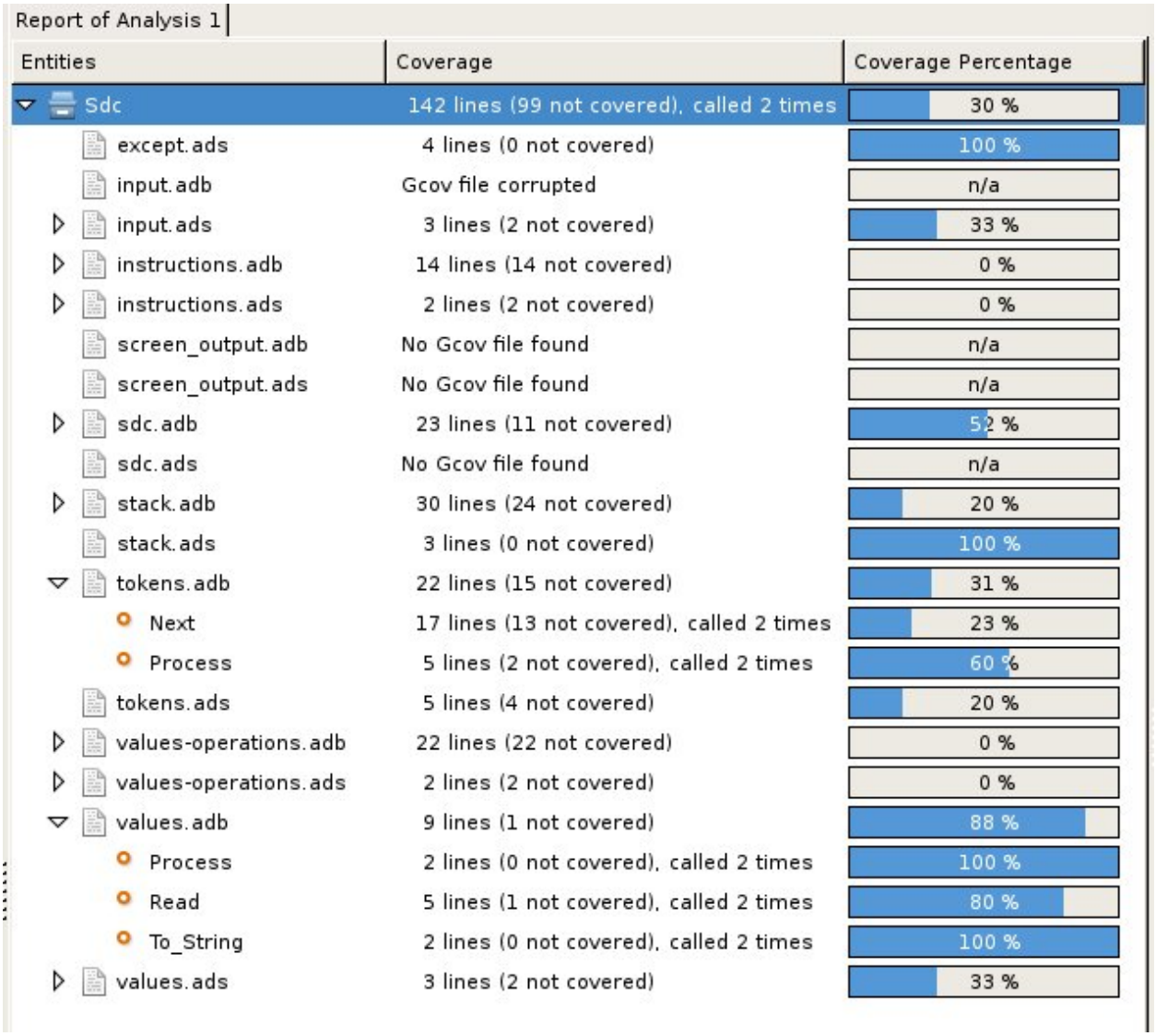
  Delete coverage information from the specified source file.

- *Show Coverage report*

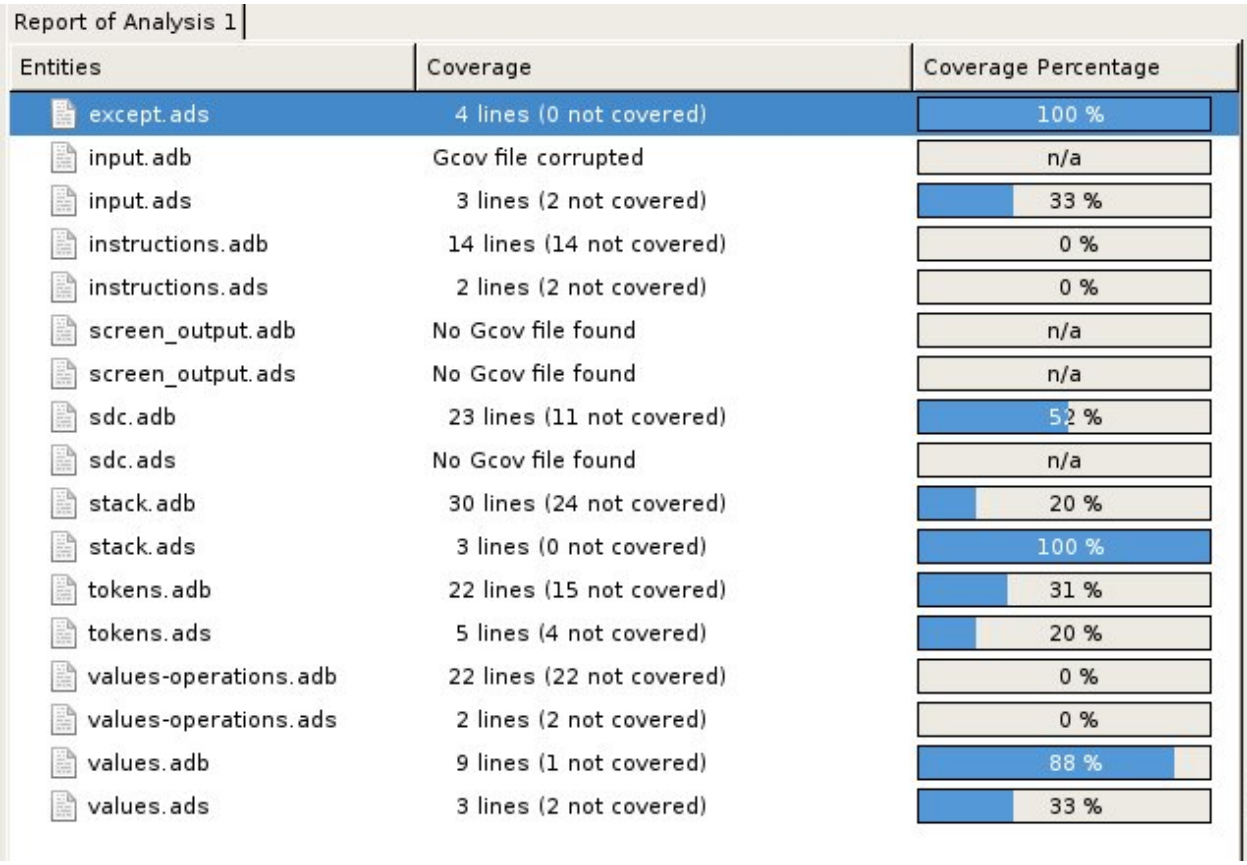
  Open a new window summarizing the coverage information. (This entry appears only if the contextual menu has been created from outside the Coverage Report.)

10.8.3 The Coverage Report

Once GPS loads coverage information, it displays a graphical coverage report containing a tree of Projects, Files and Subprograms with corresponding coverage information for each shown in a column on the side.



The contextual menus generated for this report contain, in addition to the regular entries, some specific Coverage Report options allowing you to expand or fold the tree, or to display flat lists of files or subprograms instead of a tree. A flat list of files looks like:



GPS and **gcov** both support many different programming languages, so code coverage features are available in GPS for many languages. But subprogram coverage details are not available for every supported language. If you change the current main project in GPS, using the *Project* → *Open* menu, for example, GPS deletes all loaded coverage information for the loaded project.

10.9 Stack Analysis

GPS provides an interface to **GNATstack**, the static stack analysis tool. This interface is only available if you have the `gnatstack` executable installed and available on your path. GPS computes, loads, and visually displays stack usage information for the entire project hierarchy. You can enter stack usage information for unknown and unbounded calls within GPS.

Once computed and loaded, GPS summarizes the stack usage information in a report and uses it to annotate source code with stack usage annotations. The largest stack usage path is loaded into the *Locations* view. See *The Locations View*.

Specify stack usage information for undefined subprograms by adding one or more `.ci` files to the set of GNATStack switches in the *Switches* attribute of the `Stack` package of your root project. For example:

```
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*. Use *The Stack Usage Editor* to edit stack usage information for undefined subprograms.

10.9.1 The Stack Analysis Menu

Access all the stack analysis operations via the *Tools* → *Stack Analysis* menu:

Analyze stack usage

Generate stack usage information for the root project.

Open undefined subprograms editor

Open the undefined subprograms editor.

Load last stack usage

Load (or reload) the latest stack usage information for the root project.

Clear stack usage data

Remove stack analysis data loaded in GPS and any associated information such as annotations in source editors.

10.9.2 The Contextual Menu

The contextual menu for a project, file, or subprogram entity (including the entities listed in the coverage report) has a *Stack Analysis* submenu containing the following options, dependong on the type of entity selected:

Show stack usage

Show stack usage information for every subprogram in the currently selected file.

Hide stack usage

Hide stack usage information for every subprogram in the currently selected file.

Call tree for xxx

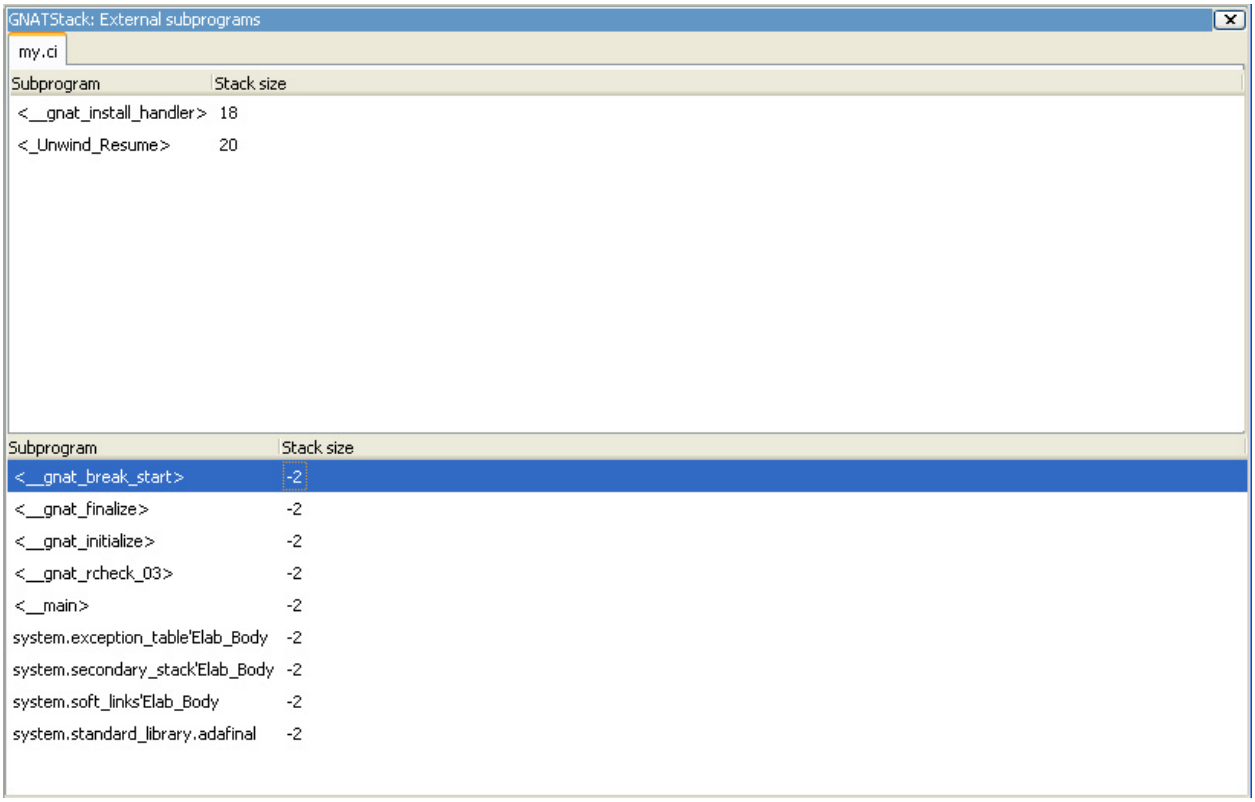
Open the *Call Tree* view for the currently selected subprogram.

10.9.3 The Stack Usage Report

Once GPS has loaded the stack usage information, it displays a report containing a summary of the stack analysis.

10.9.4 The Stack Usage Editor

The *Stack Usage Editor* allows you to specify the stack usage of undefined subprograms so these values can be used to refine results of future analysis.



The *Stack Usage Editor* contains two main areas. The notebook on the top allows you to select the file to edit. It displays the contents of the file and allows you to enter or change the stack usage of subprograms in it. The table in the bottom area displays all subprograms whose stack usage information is not specified and allows you to set them.

Specify the stack usage information for subprograms by clicking in the stack usage column to the right of the subprogram's name. When you specify a value in the bottom 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.

GPS saves all changes when the stack usage editor window is closed.

CHAPTER

ELEVEN

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*.

Two areas of the project editor to modify the project's properties are particularly relevant to cross environments: the *Cross environment* section of the *General* page, and the *Toolchains* section of the *Languages* page.

In the *Toolchains* section, you typically either scan your system to display toolchains found by GPS 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 manually modify some of the tools defined in this toolchain in the *Details* section of the *Languages* page.

For example, assume you have an Ada project using a Powerpc VxWorks configuration. When you press the *Scan* button, you should see the toolchain **`powerpc-wrs-vxworks`** appear in the *Toolchains* section. Selecting this toolchain changes the *Details* section, displaying the relevant tools (e.g., changing *Gnatls* to **`powerpc-wrs-vxworks-gnatls`** and *Debugger* to **`powerpc-wrs-vxworks-gdb`**).

You can modify the list of toolchains that can be selected when using the *Add* button and their default values via a custom XML file. See *Customizing and Extending GPS* and in particular *Customizing Toolchains* 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, 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, you may want to provide a different body file for a specific 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 debugger issues specific to cross environments. You will find more information on debugging at *Debugging*.

To automatically connect to the correct 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, connect to a remote agent by using the *Debug* → *Debug* → *Connect to Board...* menu. This opens 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.

To load a new module on the target, select the *Debug* → *Debug* → *Load File...* menu.

If a module has been loaded on the target and is not known to the current debug session, use the *Debug* → *Debug* → *Add Symbols...* menu to load the symbol tables in the current debugger.

Similarly, if you are running the underlying debugger (gdb) on a remote machine, specify the name of this machine by setting the *Tools host* field of the project properties.

CHAPTER

TWELVE

USING GPS FOR REMOTE DEVELOPMENT

It is common for programmers in a networked environment to use a desktop computer that is not itself suitable for their development tasks. For example, each developer may have a desktop PC running Windows or GNU/Linux as their access to a company network and do all their development work on shared networked servers. These remote servers may be running an operating system different from the one on their desktop machine.

One common way of operating in such an environment is to access the server through a remote windowing system such as X-Windows. GPS can be used in such way, but it is not necessarily the most efficient configuration because running GPS remotely on a shared server increases the workload of the server as well as traffic on the network. When the network is slow, 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 of running natively on the desktop, with compilation, execution, and/or debugging 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, your configuration must meet the following conditions:

- Have a remote connection to the host using **`rsh`**, **`ssh`**, or **`telnet`**. GPS can 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. (**`rsync`** can be found at http://www.samba.org/rsync/ for Unix, and is part of Cygwin under Windows: http://www.cygwin.com.
- Either subprojects must be 'withed' by the main project using relative paths or the absolute paths must be the same on both the desktop and the server.

You perform the full remote development setup in two steps:

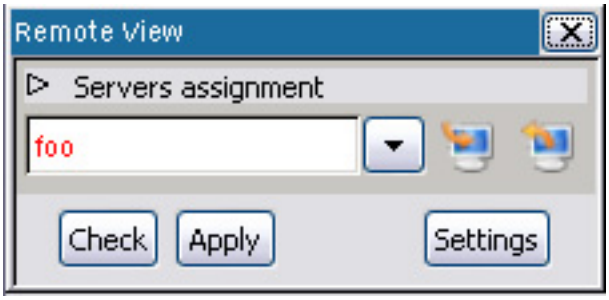
- Setup the remote servers configuration.
- Setup a remote project.

12.2 Setup the remote servers

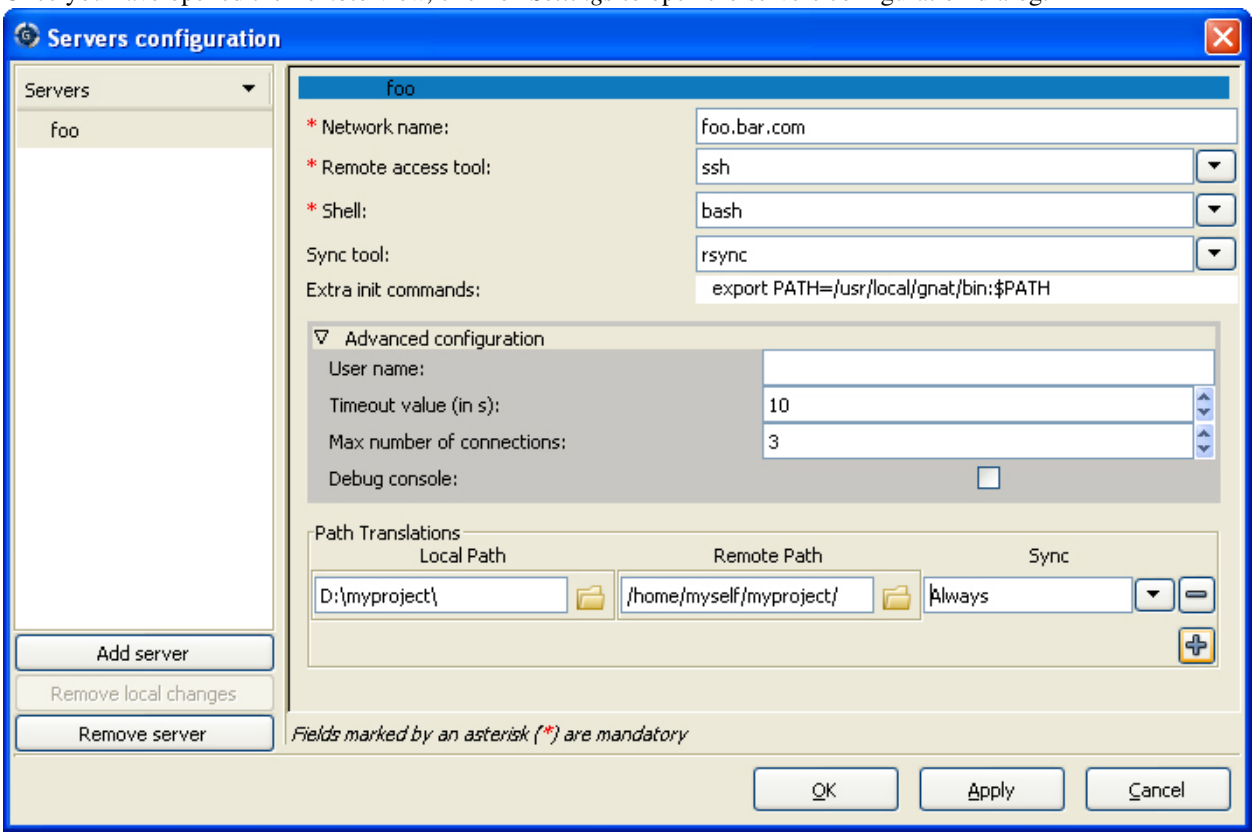
12.2.1 The remote configuration dialog

Open the remote configuration dialog using the *Tools->Views->Remote* menu to configure remote servers. You can also set a predefined configuration when installing GPS by using XML files. (See *Defining a remote server*, and

Defining a remote path translation, for more information.)



Once you have opened the *Remote* view, click on *Settings* to open the servers configuration dialog.



This dialog consists of two parts:

- The left part dialog contains the list of configured servers, each identified by a nickname. Three buttons allow you to create, reinitialize, or delete a server.
- The right part contains the selected server's configuration.

To set up a remote server, first create a new server by clicking on the *Add Server* button on the bottom left of the dialog. Enter a unique nickname identifying the server (not necessarily the network name of the server). This server is automatically selected and the right part of the dialog shows its configuration, which is initially mostly empty.

12.2.2 Connection settings

For each server, you first need to complete the section describing how GPS should connect to that server. All mandatory fields are identified by an asterisk:

- Network Name

  The name used to connect to the server via your network. It can be either an IP address, a host name on your local network, or a fully qualified name with domain.

- Remote Access Tool

  A drop-down list specifying the tool used to connect to the server. GPS contains built in support for the following tools

  - **ssh**
  - **rsh**
  - **telnet**
  - **plink** (Windows tool) in **ssh**, **rsh**, or **telnet** mode

  See *Defining a remote connection tool* if you need to add a different tool. If a tool is not in your path (for example, because it is not installed), it won't appear in the tools list. Some tools incompatible with GPS are not displayed either, such as the Microsoft telnet client.

- Shell

  Which shell runs on the remote server. GPS supports the following Unix shells:

  - sh
  - bash
  - csh
  - tcsh

  GPS also support the Windows shell (`cmd.exe`). See *Limitations*, for Cygwin's shell usage on Windows: it is preferable to use `cmd.exe` as a remote shell on Windows servers.

You may need to specify other fields, but they are not mandatory. Most are accessible through the advanced configuration pane.

- The *Remote Sync Tool* is used to synchronize remote and local filesystems, if these are not shared filesystems. Only **rsync** is supported by GPS.
- The *Extra Init Commands* lists initialization commands that GPS sends to the server when it connects to the remote machine, the chosen shell is launched, and your default initialization files are read (i.e. .bashrc file for the bash shell). GPS sends these extra commands, allowing you to, for example, specify a compilation toolchain.
- The *User Name* specifies the name used to connect to the server. The default is your current login name on your local machine.
- The *Timeout* value determines when a connection to a remote host is considered dead. All elementary operations performed on the remote host (i.e., those operations that normally complete almost immediately) use this timeout value. The default is 10 seconds. If you have a very slow network connection or a very overloaded server, set this to a higher value.
- The *Maximum Number of Connections* is the maximum number of simultaneous connections GPS is allowed to make to this server. If you want to compile, debug, and execute at the same time on the machine, GPS needs more than one connection to do this. The default is 3.
- Depending on the kind of server and the remote access tool used, commands sent to the server may require a specific line terminator, typically either the LF character or CR/LF characters. Usually GPS can automatically detect which is needed (the 'auto' mode), but you can force the choice to CR/LF (cr/lf handling set to 'on') or LF (cr/lf handling set to 'off').

- The *Debug Console* allows you to easily debug a remote connection. If checked, it opens a console displaying all exchanges between GPS and the selected server.

12.2.3 Path settings

The final section of the configuration defines the path translations between your local host and the remote server.

The remote path definitions allow GPS to translate your locally loaded project (that resides in your local filesystem) to paths used on the remote server. This section also tells GPS how to keep those paths synchronized between the local machine and the remote server.

All your project's dependencies must reside in a path defined here. You retrieve those paths by using **`gnat list -v -Pyour_project`**. 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 (see *Connection settings*.) Clicking on *Apply* applies your connection configuration and allows you to browse the remote host to select the remote paths.

You can set one of five types of path synchronization for each path:

- *Never*: no synchronization is required from GPS because the paths are shared using an OS mechanism like NFS.
- *Manually*: synchronization is needed, but is only 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 shared between your host and the remote host (using NFS or SMB filestems, for example), only the roots of those filesystems need to be specified, using each server's native paths (on Windows, the paths are specified using the "X:\my\mounted\directory\" syntax and on Unix, using the "/mnt/path/" syntax).
- If the project's files are synchronized using **`rsync`**, defining a too generic path translation leads to very slow synchronization. In that is the case, define the paths as specifically as possible in order to speed up the synchronization process.

Note that navigation to entities of the run-time is not supported in remote mode.

12.3 Setup a remote project

12.3.1 Remote operations

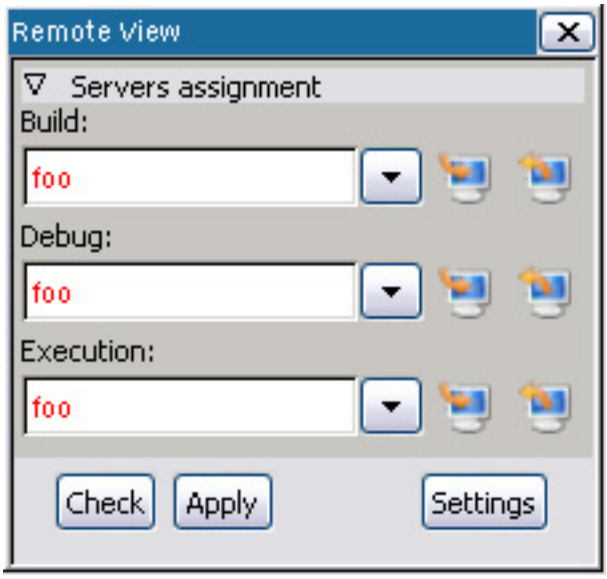
GPS defines four different categories of remote operation and corresponding servers: Build operations, Debug operations, Execution operations and Tools operations. All compiler-related operations are performed on the Build_Server. The Tools_Server is explained below. The debugger runs on the Debug_Server and the project's resulting programs run on the Execution_Server. The GPS_Server (the local machine) is used for all other operations. These "servers" may not (and are often not) different machines.

The Tools_Server handles 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 (which the Tools_Server runs), or whether a specific, older baseline compiler version must be used.

If 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, all Tools_Server operations are always executed on the local machine.

12.3.2 The remote view

Use the *Remote* view (*Tools->Views->Remote*) to assign servers to categories of operations for the currently loaded project. You can assign a different server to each operation category if you fully expand the *Servers Assignment* tab. Alternatively, assign all categories to a single server in one step if the you have left the *Servers Assignment* tab collapsed.



When you select a server for a particular category, the change is not immediately effective, as indicated by the server's name appearing in red. This allows you to check the configuration before applying it, by pressing the *Check* button. This button tests for a correct remote connection and verifies that the project path exists on the build server and has an equivalent on the local machine.

Clicking the *Apply* button performs the following actions:

- Reads the default project paths on the Build_Server and translates them into local paths.
- Synchronizes those paths marked as Sync *Always* or *Once to local* from the build server.
- Loads the translated local project.
- Assigns the Build, Execution and Debug servers.

If one of those operations fails, GPS reports the errors in the *Messages* view and retains the previous project settings. Once a remote server is assigned, the remote configuration is automatically loaded each time the project is loaded.

Use the two buttons on the right of each server to manually perform a synchronization from the server to your local machine (left button) or from your local machine to the server (right button).

12.3.3 Loading a remote project

If the project you want to use is already on a remote server, you can directly load it on your local GPS by using the *Project → Open From Host* menu and selecting the server's nickname. This shows you its file tree. Navigate to your project and select it. The project is loaded as described above with all remote operations categories assigned to the selected server by default.

You can reload your project from local files on your machine. The remote configuration is automatically reapplied.

12.4 Limitations

The GPS remote mode imposes some limitations:

- Execution: you cannot use an external terminal to remotely execute your application. The *Use external terminal* checkbox of the run dialog has 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. To use GPS with a remote Windows server using Cygwin's **bash**, you must use directories that are the same on Windows and Cygwin (absolute paths). For example, a project using "C:\my_project" is accepted if Cygwin's path is `/my_project`, but not if `/cygdrive/c/my_project` is specified.

  Even if you use Cygwin's **sshd** on such a server, you can still access it using `cmd.exe` (*Connection settings*.)

CHAPTER

THIRTEEN

CUSTOMIZING AND EXTENDING GPS

GPS provides several levels of customization, from simple preference dialogs to powerful scripting capability through the Python language. This chapters describes each of these capabilities.

13.1 Color Themes

The *Color Theme window* shows a list of color themes to choose from, presented in the form of a list of screenshots. Clicking on the button underneath a screenshot applies the given color theme to GPS.

Applying a color theme modifies the corresponding GPS preferences. It is therefore possible to customeize the colors after a theme has been applied, through the preferences dialog.

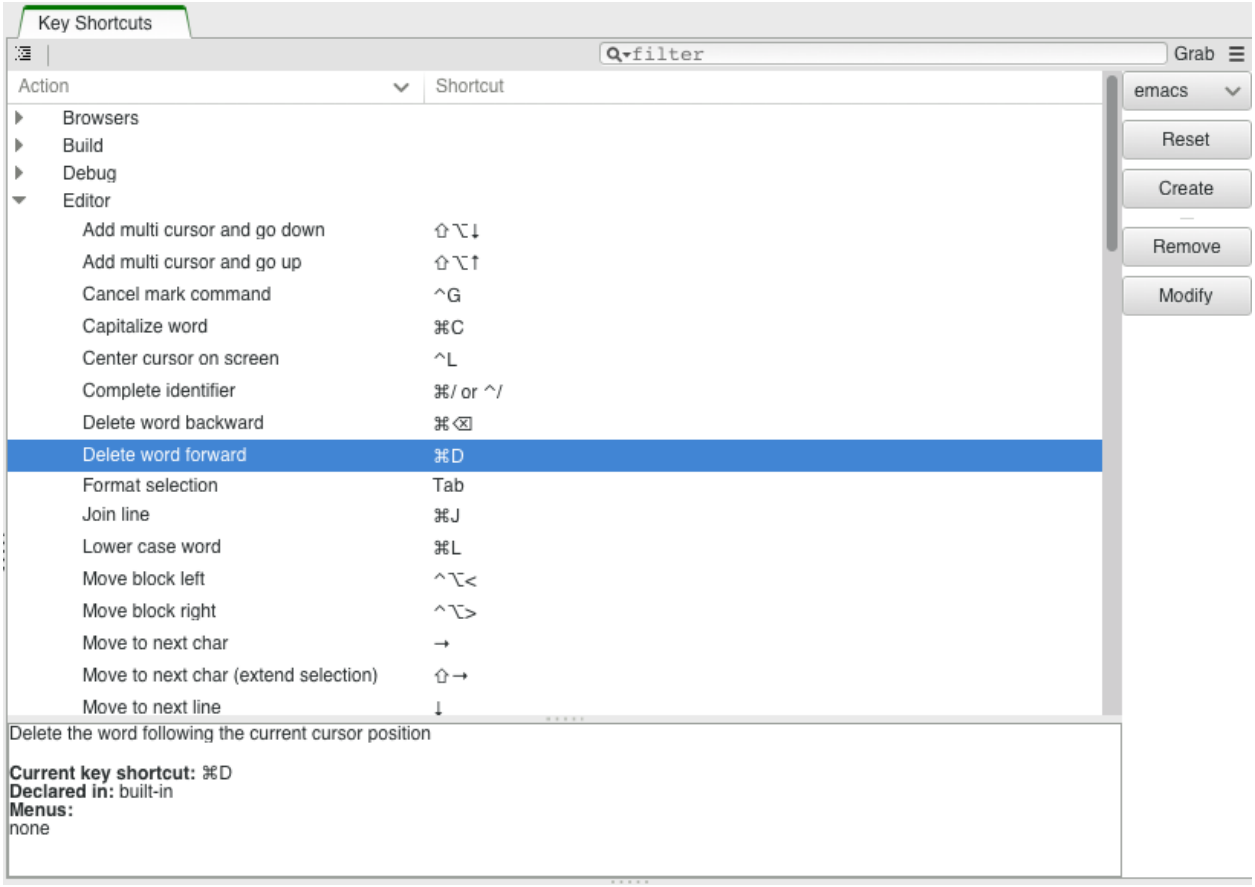
GPS supports importing themes which use the TextMate (`.tmTheme`) format: at startup, GPS will look in the directory file:*INSTALL/share/gps/color_themes/themes/* and will include all the `.tmTheme` found at the first level of each subdirectory.

13.2 Custom Fonts

In addition to the system fonts, GPS will load the fonts located under `share/gps/fonts`) in the GPS installation directory. The supported formats are `.otf`, `.ttf` and `.ttc`.

This mechanism works only on UNIX/Linux systems; under Windows, fonts need to be added at the system level.

13.3 The Key Shortcuts Editor



This editor is started through the *Edit* → *Key Shortcuts* menu. It provides a convenient way to edit the keyboard shortcuts that are available throughout GPS.

This editor is displayed as a separate dialog (floating window) by default, but you can embed it in the main GPS window by selecting the menu *Window* → *Floating*.

All keyboard shortcuts are associated with actions, which are either predefined in GPS, or defined in your customization python files, as documented in *Customizing through XML and Python files*. The main part of the editor is a list showing all actions that are defined in GPS, grouped into categories.

There are literally hundreds of such actions, and finding the one you should use might be difficult. To simplify the search, you can use the filter field at the top-right corner of the editor. Typing any text in this field will restrict the list of actions to those that contain the text either in their name, their description, their keyboard shortcut, or the menus the action is bound to. Entering keyboard shortcut is in fact easier done via the *Grab* button next to the filter field. Click on it, then type the shortcut you are looking for.

By using the local configuration menu (click on the top-right button of the editor), you can further restrict what is displayed in the editor:

- *Shortcuts only* will only display the actions that have an actual shortcut, and hide all the others.
- *Show categories* can be unset if you just want to display a flat list of all the actions.
- All menus in GPS are themselves bound to actions. In general, it is better to associate a key shortcut to the action itself, as opposed to the menu. For this reason, GPS by default does not list all the menus in the keyboard shortcuts editor. However, historically, GPS used to show all menus there and you can get this behavior back by enabling the *Show all menus* configuration.

When you select an action, GPS will display its documentation in the bottom part of the ditor. This documentation also includes a pointer to the python file that defines the action (or whether it is built-in in GPS), as well as the list of menus that will execute this action when selected.

Finally, the editor includes a set of buttons on its right side, which are grouped into two logical sets:

- The top three buttons allow you to control *Key themes*. These are sets of keyboard shortcuts that are either provided as part of GPS (for instance GPS provides an Emacs key theme which attempts to emulate some of the Emacs key bindings) or created by the user.

  The first box lists all known themes, and lets you alternate between them simply by selecting their name. This will unset all existing key bindings except the ones you have set manually, and replace them with the shortcuts loaded from the key theme. This also updates all the menus to reflect the new shortcuts.

  The *Reset* button will discard all the keybindings you have manually overridden, and revert to the theme's default key bindings.

  The *Create* lets you create a new key theme by copying all the current shortcuts (those from the theme and the ones you have set manually) into a new theme. In effect, this creates a new XML file in the directory `$HOME/.gps/key_themes`. Removing a custom key theme is done by deleting the file from that directory, no GUI is provided for this at the moment.

- The second group of buttons lets you edit the shortcut for the currently selected action either by removing the shortcut, or by overriding the ones that is currently set.

  When you click on the *Modify* button, GPS waits for you to press any keyboard shortcut you wish to associate with the action. This shortcut can include multiple keys, so for instance to get an Emacs-like binding you could for instance press `Ctrl-x` and then press `Ctrl-k`. After pressing the last key in the sequence, wait for a short delay and GPS will associate the resulting shortcut to the action and update the menus, when relevant, to show the new binding. Note that multi-key shortcuts cannot be displayed in menus due to technical limitations of the GUI toolkit.

  Assigning a new shortcut to an action causes the following:

  - All actions and menus currently associated with key are no longer executed when the key is pressed.
  - All key shortcuts defined for this action are replaced by the new one; the action is only executable through this new shortcut.

Any change to the shortcuts is immediately and automatically saved, so that they become instantly usable in GPS, and will be restored properly when GPS is restarted.

13.4 Editing Plugins

You can extensively customize GPS through external plugins, either ones you write (see *Customization files and plugins*) or using one of the plugins in GPS's own collection.

Some plugins are loaded by default when GPS starts (such as support for the CVS version management system and support for highlighting in various programming languages) and others are available but not loaded automatically, such as Emacs emulation mode.

Some plugins provided with GPS are:

- `Makefile` support

  A plugin that parses a `Makefile` and creates menus for each of its targets so you can easily start a **`make`** command.

- Cross-references enhancements

Some plugins take advantage of GPS's cross-references information to create additional menus for navigation such as jumping to the primitive operations of Ada tagged types and to the body of Ada separate entities.

- Text manipulation

  Several plugins provide support for advanced text manipulation in the editors, for example to align a set of lines based on various criteria or to manipulate a rectangular selection of text.

You can graphically choose which plugins are loaded on startup by opening the preferences editor dialog (*Edit* → *Preferences* menu), under the *Plugins* section. This section lists all the known plugins on the left. By selecting one particular plugin, the corresponding preferences page is opened on the right. Each plugin page comes with the same layout:

- A *General* group

  This group indicates the exact location of the plugin file. Moreover, this group contains a toggle button (*Loaded at startup*) which allows you to decide if this plugin should be loaded or not in the next GPS session.

  As described in *Customization files and plugins*, GPS searches for plugins in various directories and, based on these directories, decides whether to automatically load the plugin on startup.

- An optional *Preferences* group

  This group lists all the preferences related to the selected plugin, allowing you to customize the plugin behavior. Note that this group is displayed only if preferences have been registered for this plugin.

- A *Documentation* frame

  This frame displays the plugin file documentation. By convention, each plugin starts with a comment indicating the purpose of this plugin and more detailed documentation on its usage.

If you have modified the list of plugins that should be loaded at startup, you will need to restart GPS, since it cannot unload a module due to such an action having too many possible effects on GPS: then, a dialog is displayed asking you whether you would like to exit GPS when closing the preferences editor dialog.

All the changes explictly set by the user in the list of plugins to load at startup are saved in `HOME/.gps/startup.xml`.

13.5 Customizing through XML and Python files

13.5.1 Customization files and plugins

You can customize many capabilities in GPS using files it loads at startup. For example, you can add items to the menu and tool bars as well as defining new key bindings, languages, and tools. Using Python as a programming language, you can also add new facilities and integrate your own tools into the GPS platform.

GPS searches for these customization files at startup in several different directories. Depending on where they are found, they are either automatically loaded by GPS (and thus can immediately modify things in GPS) or may only be made visible in the *Plugins* section of the preferences editor dialog (see *Editing Plugins*).

GPS searches these directories in the order given below. Any script loaded later can override operations performed by previously loaded scripts. For example, they can override a key shortcut, remove a menu, or redefine a GPS action.

In each directory name below, `INSTALL` is the name of the directory in which you have installed GPS. `HOME` is your home directory, either by default or as overridden by the `GPS_HOME` environment variable. In each directory, only files with `.xml` or `.py` extensions are used. Other files are ignored, although for compatibility with future versions of GPS you should not have keep other files in these directories.

- Automatically-loaded, global modules

  The `INSTALL/share/gps/plug-ins` directory contains the files GPS automatically loads by default (unless overridden by the user via the guilabel:*Plugins* section of the preferences editor dialog). These plugins are visible to any user on the system using the same GPS installation. Reserve this directory for critical plugins that almost everyone will use.

- Not automatically-loaded, global modules

  The `INSTALL/share/gps/library` directory contain files GPS displays in the *Plugins* section of the preferences editor dialog but does not load automatically. Typically, these files add optional capabilities to GPS that many of users generally will not use.

- `GPS_CUSTOM_PATH`

  Set this environment variable before launching GPS to be a list of directories, separated by semicolons (';') on Windows systems and colons (':') on Unix systems. All files in these directories with the appropriate extensions are automatically loaded by default by GPS, unless overridden by the user through the *Plugins* section of the preferences editor dialog.

  This is a convenient way to have project-specific customization files. You can, for example, create scripts 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.

  These directories are also used to search for icons referenced in your plug-ins.

- Automatically loaded user directory

  The directory `HOME/.gps/plug-ins` is searched last. Any script in it is loaded automatically unless overridden via the *Plugins* section of the preferences editor dialog.

  This is a convenient way for you to create your own plugins or test them before you make them available to all GPS users by copying them to one of the other directories.

Any script loaded by GPS can contain customization for various aspects of GPS, such as aliases, new languages or menus, in a single file.

Python files

You can format the Python plugin in any way you want (as long as it can be executed by Python, of course), the following formatting is suggested. These plugins are visible in the *Plugins* section of the preferences editor dialog, so having a common format makes it easier for users to understand each plugin:

- Comment

  Your script should start with a comment on its goal and usage. This comment should use Python's triple-quote convention, rather than the start-of-line hash ('#') signs. The first line of the comment should be a one line explanation of the goal of the script, separated by a blank line from the rest of the comment.

- Implementation

  Separate the implementation from the initial comment by a form-feed (control-L); the startup scripts editor only displays the first page of the script in the first page of the editor.

  If possible, scripts should avoid executing code when they are loaded. This gives the user a chance to change the value of global variables or override functions before the script is actually launched. Instead, you should 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 command:<*?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 arbitrary. The format is therefore:

```
<?xml version="1.0" ?>
<root_node>
   ...
</root_node>
```

The first line after the <*?xml?*> tag should contain a comment describing the purpose and usage of the script. This comment is made visible in the the preferences page associated with this plugin, under *Plugins* section of the preferences editor dialog. The list of valid XML nodes that you can specify under `<root>` is described in later sections. It includes:

- <*action*>
- <*key*>
- <*submenu*>
- <*pref*>
- <*preference*>
- <*alias*>
- <*language*>
- <*button*>
- <*entry*>
- <*vsearch-pattern*>
- <*tool*>
- <*filter*>
- <*contextual*>
- <*case_exceptions*>

- *<documentation_file>*
- *<doc_path>*
- *<project_attribute>*
- *<remote_machine_descriptor>*
- *<remote_path_config>*
- *<remote_connection_config>*
- *<rsync_configuration>*

13.5.2 Defining Actions

This mechanism links actions to their associated menus or key bindings. Actions can take several forms: external commands, shell commands and predefined commands, each explained in more detail below.

Define new actions using the `<action>` tag. This tag accepts the following attributes:

- `name` (required)

  The name by which the action is referenced elsewhere in the customization files, for example when it is associated with a menu or toolbar button. It can contain any character, although you should avoid XML special characters and it cannot start with a '/'.

- `output` (optional)

  Where the output of the commands are sent by default. You can override this for each command using the same attribute for `<shell>` and `<external>` tags. See *Redirecting the command output*.

- `show-command` (optional, default **`true`**)

  Whether the text of the command itself should be displayed in the same place as its output. Neither are displayed if the output is hidden. The default shows the command along with its output. You can override this attribute for each command.

- `show-task-manager` (optional, default **`false`**)

  Whether an entry is in the tasks view to show this command. The progress bar indicaton is associated with this entry so if you hide the entry, no progress bar is shown. Alternatively, several progress bars may displayed for your action if this is enabled, which might be an issue depending on the context. You can override this attribute for each external command.

- `category` (optional, default **`General`**)

  The category in the keybindings editor (*Edit* → *Key bindings* menu) in which the action is displayed. If you specify an empty string, the action is considered part of the implementation and not displayed in the editor and the user will not be able to assign it a keybinding through the graphical user interface (although this can still be done via XML commands).

If you define the same action multiple times, the last definition is used. However, items such as menus and buttons that reference the action keep their existing semantics: the new definition is only used for items created after it is defined.

The `<action>` tag can have one or several children, all of which specify a command to execute. All commands are executed sequentially unless one fails, in which case the following commands are ignored.

The valid children of `<action>` are the following XML tags:

- `<external>`

  Defines a system command (i.e. a standard Unix or Windows command).

- `server` (optional)

  Execute the external command on a remote server. The values are **`gps_server`** (default), **`build_server`**, **`execution_server`**, **`debug_server`**, and **`tools_server`**. See *Remote operations* for information on what each of these servers are.

- `check-password` (optional)

  Tell GPS to check for and handle password prompts from the external command. The values are false (default) and true.

- `show-command` (optional)

- `output` (optional)

  Override the value of the attribute of the same name specified in the `<action>` tag.

- `progress-regexp` (optional)

- `progress-current` (optional, default **`1`**)

- `progress-final` (optional, default **`2`**)

  `progress-regexp` is a regular expression that GPS matches the output of the command against. When the regular expression matches, it must provide two subexpressions whose numeric values represent the current and total number of steps to perform, which are used to display the progress indicators at the bottom-right corner of the GPS window. `progress-current` is the ordinal of the subexpression containing the current step, and `progress-final` is the ordinal of the subexpression containing the total number of steps, which grows as needed. For example, **`gnatmake`** outputs the number of the file it is currently compiling and the total number of files to be compiled. However, that last number may increase, since compiling a new file may cause additional files to be compiled.

  The name of the action is printed in the progress bar while the action is executing. Here is an example:

```
<?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-hide` (optional, default **`true`**)

  If true, all lines matching `progress-regexp` and are used to compute the progress are not displayed in the output console. Otherwise, those lines are displayed with the rest of the output.

- `show-task-manager` (optional, default inherited from `<action>`)

  Whether an entry is created in the tasks view to show this command. The progress bar indicator is associated with this entry, so if you hide the entry, no progress is shown. Alternatively, several progress bars may be displayed for your action if this is enabled, which might be an issue depending on the context.

  If set a value for *progress-regexp*, this attribute is automatically set to true so the progress bar is displayed in the tasks view.

Note for Windows users: like Unix, scripts can be called from a custom menu. To allow that, you need to write your script in a `.bat` or `.cmd` file and call this file. So the `external` tag would look like:

```
<?xml version="1.0" ?>
<external_example>
  <action name="my_command">
    <external>c:\\.gps\\my_scripts\\my_cmd.cmd</external>
  </action>
</external_example>
```

- `on-failure`

  Specifies a command or group of commands to be executed if the previous external command fails. Typically, this is used to parse the output of the command and fill the *Locations* view appropriately (see *Processing the tool output*).

  For example, the following action spawns an external tool and parses its output to the *Locations* view. It calls the automatic fixing tool if the external tool fails.

  You can use the %... and $... macros in this group of commands (see *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`

  You can use custom menu items to invoke GPS commands using the `shell` tag. These are 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 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 written in the GPS shell language. However, you can specify the language through the `lang` attribute, whose default value is "shell". You can also specify "python".

  When programming with the GPS shell, execute multiple commands by separating them with semicolons. Therefore, the following example adds a menu that 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 %1</shell>
  </action>
</current_file_uses>
```

- `<description>`

  A description of the command, which is used in the graphical editor for the key manager. See *The Key Shortcuts Editor*.

- `<filter>`, `<filter_and>`, `<filter_or>`

  The context in which the action can be executed. See *Filtering actions*.

You can mix both shell commands and external commands. For example, the following command opens an :program'xterm' (on Unix systems only) in the current directory, which depends on the context:

```
<?xml version="1.0" ?>
<xterm_directory>
  <action name="xterm in current directory">
    <shell lang="shell">cd %d</shell>
    <external>xterm</external>
  </action>
</xterm_directory>
```

As you can see in some of the examples above, some special strings are expanded by GPS just prior to executing the command, for example "%f" and "%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 includes, for example, aliases expansion and manipulating MDI windows. You can display all known actions (both predefined and the ones you defined in your own customization files) by opening the key shortcut editor using the *Edit* → *Key shortcuts* menu.

13.5.3 Macro arguments

You use macro arguments to pass parameters to shell or external commands in any actions you define. Macro arguments are special parameters that are transformed every time the command is executed. The macro arguments below are provided by GPS. The equivalent Python code is given for some arguments. This code is useful when you are writing a full python script.

- `%a`

  If the user clicked inside the *Locations* view, name of the current line's category.

- `%builder`

  Replaced by the default builder configured in GPS. This can be **`gnatmake`** if your project contains only Ada code, or **`gprbuild`** for non-Ada or multi-language projects. This macro is only available in commands defined in the *Build Manager* and *Build Launcher* dialogs.

- `%c`

  The column number on which the user clicked. Python equivalent:

  ```
  GPS.current_context().column()
  ```

- `%d`

  Current directory. Python equivalent:

```
GPS.current_context().directory()
```

- `%dk`

  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, which may, for example, be due to cross-reference information not being up-to-date.

- `%E`

  Full path to the executable name corresponding to the target.

- `%ek`

  Krunched name of the entity the user clicked on. Like `%e`, except long names are shorted as in `%fk`.

- `%eL`

  Either an empty string or `-eL`, depending on whether the *Fast Project Loading* preference if set. `-eL` is used by GNAT tools to specify whether symbolink links should be followed when parsing projects. This macro is only available in commands defined in the *Build Manager* and the *Build Launcher* dialogs.

- `%external`

  Command line specified in the *External Commands* → *Execute command* preference.

- `%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()
```

- `%fd`

  Absolute path for the directory that contains the current file.

- `%fk`

  Krunched base name of the currently selected file. This is the same as `%f` except that long names are shortened with some letters replaced by "[...]". Use this 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 in the *Messages* view. This macro is only available in commands defined in the *Build Manager* and *Build Launcher* dialogs.

- `%gnatmake`

  The **gnatmake** executable configured in your project file.

- `%gprbuild`

  The **gprbuild** command line configured in your project file.

- `%gprclean`

  Default cleaner configured in GPS. This can be, for example, **gnat clean** or **gprclean**. This macro is only available in commands defined in the *Build Manager* and *Build Launcher* dialogs.

- `%GPS`

  GPS's home directory (i.e., the `.gps` directory in which GPS stores its configuration files).

- `%i`

  If the user clicked inside the *Project* view, name of the parent project, i.e., the one that is importing the one clicked on. With this definition of parent project, a given project may have multiple parents, but the one here is the one from the *Project* view..

- `%l`

  Number of the line in which the user clicked. Python equivalent:

```
GPS.current_context().line()
```

- `%o`

  Object directory of the current project.

- `%O`

  Object directory of the root project.

- `%system_bin_dir`

  The directory containing the GPS executable.

- `%p`

  Name of the current project (not the project file). The `.gpr` extension is not included and the casing is the one in the project file not that of the file name itself. If the current context is an editor, the name of the project to which the source file belongs. Python equivalent:

```
GPS.current_context().project().name()
```

- `%P`

  Name of root project. Python equivalent:

```
GPS.Project.root().name()
```

- `%Pb`

  Basename of the root project file.

- `%Pl`

  Name of the root project converted to lower case.

- `%pp`

  Current project file pathname. If a file is selected, the project file to which the source file belongs. Python equivalent:

```
GPS.current_context().project().file().name()
```

- `%PP`

  Root project pathname. Python equivalent:

```
GPS.Project.root().file().name()
```

- `%pps`

  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 does not belong to any project. This is intended mostly for use with the GNAT command line tools. GPS quotes the project name if it contains spaces. Python equivalent:

```
if GPS.current_context().project():
   return "-P" & GPS.current_context().project().file().name()
```

- `%PPs`

  Similar to :;file:*%PP*, except it returns the project name prepended with **`-P`**, or an empty string if the root project is the default project. This is intended mostly for use with the GNAT command line tools.

- `%(p|P)[r](d|s)[f]`

  Replaced by the list of sources or directories of a project. This list is space-separated with all names surrounded by double quotes for proper handling of spaces in directories or file names. The first letter specifies the project and successive letters which files are in the list and related options:

  - `P`

    root project.

  - `p`

    The selected project or the root project if project is selected.

  - `r`

    Recurse through the projects, including all subprojects.

  - `d`

    List source directories. Python equivalent:

```
GPS.current_context().project().source_dirs()
```

  - `s`

    List source files. Python equivalent:

```
GPS.current_context().project().sources()
```

  - `f`

    Write the list into a file and replace the parameter with the name of the file. This file is never deleted by GPS; you must do so manually in the plugin when you no longer need it.

  Examples:

  - `%Ps`

    List of source files in the root project.

  - `%prs`

    List of files in the current project and all imported sub projects, recursively.

  - `%prdf`

    Name of a file containing a list of source directories in the current project and all imported sub projects, recursively.

- `%s`

  Text selected by the user, if a single line was selected. If multiple lines are selected, returns the empty string

- `%S`

  Text selected by the user or the current entity if no selection. If the entity is part of an expression ("A.B.C"), the whole expression is returned instead of the entity name.

- `%switches(tool)`

  Value of **`IDE'Default_Switches (tool)`**. 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 you can retrieve them using this macro. The result is a list of switches, or an empty list if none.

  This macro is only available in the commands defined in the *Build Manager* and *Build Launcher dialogs*.

- `%T`

  Subtarget being considered for building. Depending on the context, this can correspond to such things as the base filename of a main source or `makefile` targets. This macro is only available in the commands defined in the *Build Manager* and *Build Launcher* dialogs.

- `%TT`

  Like *%TT*, but the full path to main sources rather than the base filename.

- `%TP`

  Similar to *%TT%*, but returns the name of the project to which the main belongs.

- `%python(cmd)`

  Executes the python command *cmd*. It should return either a string (which is inserted as is in the command line), a list of strings (which are each passed as a separate argument), or a boolean. If it returns *False*, then the target is not executed at all.

  The cmd itself can include other macros, which will be expanded. Not all macros are expanded though. For instance, a *%python()* cannot include another *%(python)*, nor any other function-like macros, like *%vars()* for instance.

  The python function should have no side effect if possible, since it might be called more than once (for instance as part of showing what the command line will be when GPS display the dialog to let you edit that command line prior to actual execution).

  Due to the way command-line parsing works, it is recommended to put triple quotes around the whole argument, as in:

```
-foo """%python(func("%TT", 1))""" -bar
-foo """%python("-one" if Choice else "-two")"""
```

to make sure the python argument is not split on spaces for instance. The closing parenthesis must be the last character before the closing triple quotes.

- `%attr(Package'Name[,default])`

  Project attribute **`Package'Name`L: the attribute :file:`Name`** from the package `Package`. You can omit `Package'` 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 none is specified.

  This macro is only available in the commands defined in the *Build Manager* and *Build Launcher* dialogs and only supports attributes that return a single string, not those returning lists.

- `%dirattr(Package'Name[,default])`

  Like `%attr`, but the directory part of an attribute value.

- `%baseattr(Package'Name[,default])`

  Like `%attr`, but the base name an attribute value.

- `%vars`

  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 scenario variables defined in the current project tree are listed. You can also use `%vars(-D)` to generate a list of switches of the form `-D`*`variable=value`*. This macro is only available in the commands defined in the *Build Manager* and *Build Launcher* dialogs.

- `%X`

  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 are listed. This macro is only available in the commands defined in the *Build Manager* and *Build Launcher* dialogs.

- `%target`

  The string `--target=`*`t`* where **`t`** is the build target, as determined by the current toolchain.

- `%%`

  The literal `%` character.

- `%ts`

  The short name for the current window ('Search', 'Project', 'Outline', or the base name for the current file).

- `%tl`

  The long name for the current window ('Search', 'Project', 'Outline' or the absolute path name for the current file).

- `%rbl`

  The name of the Remote Build server (defaults to 'localhost' when no such server is configured, hence the final 'l' in the name of the macro).

Another type of macros are expanded before commands are executed: they start with the `$` character and represent parameters passed to the action by its caller. Depending on the context, GPS passes zero, one or many arguments to an action. You will commonly use these macros when you define your own VCS system. Also see the shell function `execute_action`, which executes an action and passes it arguments.

These macros are the following

- `$1, $2, ... $n`

  Where **n** is a number. These are the argument with the corresponding number that was passed to the action.

- `$1-, $2-, ... $n-*`

  Likewise, but a string concatenating the specified argument and all subsequent arguments.

- `$*`

  String concatenating all arguments passed to the action.

- `$repeat`

  Number of times the action has been consequtively executed. This is 1 (the first execution of the action) unless the user invoked the *Repeat Next* action.

  By default, when *Repeat Next* is invoked by the user, it repeats the following action the number of times the user specified. However, in some cases, either for efficiency reasons or for other technical reasons, you may want to handle the repeat yourself. Do this 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"""
   ...
```

  The idea here is to do something only the first time the action is called (the **if** statement), but pass your shell function the number of times it should repeat (the `$remaining` parameter).

- `$remaining`

  Like $repeat, but indicates the number of times the action remains to be executed. This is 0 unless the user invoked the *Repeat Next* action.

13.5.4 Filtering actions

By default, an action can execute in any context in GPS. When the user selects the menu or key, GPS executes the action. You can restrict when an action is permitted. If the current context does not permit the action, GPS displays an error message.

You can use one of several types of restrictions:

- Using macro arguments (see *Macro arguments*).

  If an action uses one of the macro arguments defined in the previous section, GPS checks that the information is available. If not, it will not run any of the shell commands or external commands for that action.

  For example, if you specified `%F` as a parameter to a command, GPS checks there is a current file such as a currently selected file editor or a file node selected inside the *Project* view. This filtering is automatic: you do not have to do anything else.

  However, the current context may contain more information than you expect. For example, if a user clicks on a file name in the *Project* view, the current context contains a file (and hence satisfies `%F`) and also a project (and hence satisfies `%p` and similar macros).

- Defining explicit filters

  You can also specify explicit restrictions in the customization files by using the <filter>, <filter_and> and <filter_or> tags. Use these tags to further restrict when the command is valid. For example, 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

You can define filters in one of two places in the customization files:

- At the toplevel.

  You can define named filters at the same level as other tags such as <action>, <menu> or <button> tags. These are global filters that can be referenced elsewhere.

- As a child of the <action> tag.

  These filters are anonymous, although they provide exactly the same capabilities as the ones above. These are intended for simple filters or filters that you use only once.

There are three different kinds of tags representing filters:

- <filter>

  A simple filter. This tag has no child tag.

- <filter_and>

  All the children of this tag are merged to form a compound filter. they are each evaluated in turn and if one of them fails, the whole filter fails. Children of this tag can be of type <filter>, <filter_and> or :file: *<filter_or>*.

- <filter_or>

  Like <filter_and>, but as soon as one child filter succeeds, the whole filter succeeds.

If several filter tags are found under an <action> tag, they act as if they were all under a single <filter_or> tag.

The <filter>, <filter_and>, and <filter_or> tags all accept the following common attributes:

- name (optional)

  Used to create named filters that can be reused, via the *id* attribute, elsewhere in actions or compound filters. The name can have any form.

- error (optional)

  Error message GPS will display if the filter does not match and hence the action cannot be executed. If you are using the <filter_and> or <filter_or> tag, GPS will only display the error message of that filter.

In addition, the <filter> tag has the following specific attributes:

- id (optional)

  If this attribute is specified, all other attributes are ignored. Use this to reference a named filter previously defined. Here is 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>
```

GPS contains a number of predefined filters:

- `Source editor`

  Match if the currently selected window in GPS is an editor.

- `Explorer_Project_Node`

  Match if clicking on a project node in the *Project* view.

- `Explorer_Directory_Node`

  Match if clicking on a directory node in the *Project* view.

- `Explorer_File_Node`

  Match if clicking on a file node in the *Project* view.

- `Explorer_Entity_Node`

  Match if clicking on an entity node in the *Project* view.

- `File`

  Match if the current context contains a file (for example 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)

  Name of the language that must be associated with the current file in order for the filter to match. For example, if you specify **`ada`**, the user must have an Ada file selected for the action to execute. GPS determines the language for a file by using several methods such as looking at file extensions in conjunction with the naming scheme defined in the project files.

- `shell_cmd` (optional)

  Shell command to execute. The output of this command is used to find if the filter matches: if it returns "1" or "true", the filter matches. In any other case, the filter fails.

  Macro arguments (such as `%f` and *%p*) may be used in the text of the command to execute.

- `shell_lang` (optional)

  Which language the command in `shell_cmd` is written. The default if that the command is written for the GPS shell.

- `module` (optional)

  The filter only matches if the current window was created by this specific GPS module. For example, if you specify `Source_Editor`, the filter only matches if the active window is a source editor.

  You can obtain the list of module names by typing **`lsmod`** in the shell console at the bottom of the GPS window.

  This attribute is useful mostly when creating new contextual menus.

When several attributes are specified for a `<filter>` node (which cannot be combined with `id`), they must all match for the action to be executed:

```
<?xml version="1.0" ?>
<!-- The following filter only matches 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 is only 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 is 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

Actions can be associated with menus, tool bar buttons, and keys, all using similar syntax.

Each menu item has an associated path, which it behaves like a UNIX path, except it references menus, starting from the menu bar itself. The first character of this path must be `/`. The last part is the name of the menu item. For example, specifying **`/Parent1/Parent2/Item`** as a menu path is a reference to a *Parent1* → *Parent2* -> *Item* menu. If you are creating a new menu item, GPS creates any parent menus that do not already exist.

You bind a menu item to an action through the `<menu>` and `<submenu>` tags. The `<menu>` tag can have the following attributes:

- `action` (required)

  Action to execute when the item is selected by the user. If no action by this name is defined, GPS does not add a new menu. If the action name starts with a '/', it represents the absolute path to an action.

  Omit this attribute only when no title is specified for the menu. Doing that makes it a separator (see below).

  If you associate a filter with the action via the `<filter>` tag, the menu is greyed out when the filter does not match.

- `before` (optional)

  Name of another menu item before which the new menu should be inserted. If that item has not been previously created, the new menu is inserted at the end. Use this attribute to control precisely where the item menu is displayed.

- `after` (optional)

  Like `before`, but with a lower priority. If specified and there is no `before` attribute, it specifies an item after which the new item should be inserted.

The `<menu>` tag should have one XML child called `<title>`, which specifies the label of the menu. This label is actually a path to a menu, so you can define submenus.

You can define the accelerator keys for your menus using underscores in the title to designate the accelerator key. For example, if you want an accelerator on the first letter in a menu named `File`, set its title to `_File`.

The `<submenu>` tag accepts several children, such as `<title>` (which can present at most once), `<submenu>` (for nested menus), and `<menu>`.

`<submenu>` does not accept the `action` attribute. Use `<menu>` for clickable items that result in an action and `<submenu>` to define several menus with the same path.

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 more typing, but allows you to specify the exact location, at each level, of the parent 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 method is shorter but less flexible than the following, where we also create the `My_Tools` menu, if it does not already exist, to appear after the *File* menu. This cannot be done by using only `<menu>` tags. We also insert several items in that new menu:

```
<?xml version="1.0" ?>
<test>
  <submenu>
    <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>
```

If you add an item with an empty title or no title at all, GPS inserts a menu separator. For example, 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

You can also add actions as new items in contextual menus anywhere in GPS. Contextual menus are displayed when the user right clicks and only show actions relevant to the current context.

Add an item using the `<contextual>` tag, which takes the following attributes:

- `action` (required)

  Name of action to execute, which must be defined elsewhere in one of the customization files.

  If set to an empty string, a separator is inserted in the contextual menu. If you specify an item using the `before` or `after` attribute, the separator is displayed only when the specified item is.

- `before` (optional)

  Name of another contextual menu item before which the new item should appear. You can find the list of names of predefined contextual menus by looking at the output of **`Contextual.list`** in the GPS shell console. The name of your contextual menu item is the value of the `<title>` child.

  There is no guarantee the new menu item will appear just before the specified item. For example, it will not if the new item is created before the specified menu item or if a later contextual menu item also specified it must be displayed before the same item.

- `after` (optional)

  Like `before`, except it indicates the new menu item should appear after the specified item.

  If you specify both `after` and `before`, only the latter is honored.

- `group` (optional, default **0**)

  Allows you to create groups of contextual menus that are put next to each other. Items with the same group number appear before all items with a larger group number.

The `<contextual>` tag accepts one child tag, `<Title>`, which specifies the name of the menu item. If not specified, the menu item uses the name of the action. The title is the full path to the new menu item, like in the `<menu>` tag. You can create submenus by using a title of the form **`Parent1/Parent2/Menu`**. You can use macro arguments in the title, which are expended based on the current context. See *Macro arguments*.

GPS only displays the new contextual menu item if the filters associated with the action match the current context.

For example, the following example inserts a new contextual menu item that displays 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, displayed when the menu item is:

```
<?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, by using the `<button>` tag. Like 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 the action action does not exist.

This tag accepts one optional attribute, `iconname` which you can use to override the default image registered for the action or set one if the action no image. See *Adding custom icons* for more information on icons.

The following example defines a new button:

```
<?xml version="1.0" ?>
<stats>
  <button action="undo" />   <!--  use default icon -->
  <button action="execute my stats" iconname='my-image' />
</stats>
```

Use the `<button>` tag 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. Create a combo box with the `<entry>` tag, which accepts the following attributes:

- `id` (required)

  Unique id for this combo box, used later on to refer it, specifically from the scripting languages. It can be any string.

- `label` (default)

  Text of a label to display on the left of the combo box. If not specified, no text is displayed

- `on-changed` (default)

  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.

It also accepts any number of *<choice>* tags, each of which defines one value 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 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>
```

GPS provides a more convenient interface for Python, the `GPS.Toolbar` class, which provides 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 actions can be bound to specific key shortcuts through the `<key>` tag. This tag has two different forms:

- *<key load='file.xml'/>* tells GPS to load the given key theme (either from the GPS predefined directory or from the user's own directory.
- *<key action='name' exclusive='true'>shortcut</key>* 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 (although it is preferred to bind to an actual action). If the action is specified as an empty string, the key is no longer bound to any action.

  This tag does not contain any child tags. Instead, its text contents specifies the keyboard shortcut. The name of the key can be prefixed by any combination of the following:

  - **`control-`** is the control key on the keyboard;
  - **`alt-`** is the alt key on the keyboard (left or right) or the option key on OSX;
  - **`shift-`** is the shift key. It should not be necessary if you want to point to symbols for which shift would be necessary, so for instance on an US keyboard, **`shift-%`** and **`%`** are the same);
  - **`cmd-`** is the command key on OSX, or the alt key on other keyboards;
  - **`primary-`** is the command key on OSX, or the control key on other keyboards.

  You can also define multi-key bindings similar to Emacs' by separating them by a space. For example, **`control-x control-k`** means the user should press `Ctrl-x`, followed by a `Ctrl-k` to activate the corresponding action. This only works if the first key is not already bound to an action. If it is, you must first unbind it by passing an empty action to `<key>`.

  This XML node has one optional attribute **`exclusive`**. When this is set to *true*, the shortcut will no longer be used for any action that might be already using it. If you set it to *false*, multiple actions will be bound to the same shortcut. The first one for which the filter applies (i.e. the current context is right for the action) will be executed.

Use an empty string as 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>
```

If you bind multiple actions to the same key binding, they are executed sequentually, 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 if *exclusive* is true. This ensures that any action previously associated with it, either by default in GPS or in some other XML file, is no longer executed. This removal is not done when loading key themes (i.e. XML files from `$HOME/.gps/key_themes` directory), so it is possible to bind an action to multiple key bindings as part of a key theme.
- Adds the new key to the list of shortcuts that can execute the action. Any existing shortcut for the action is preserved, allowing multiple shortcuts for the action.

13.5.9 Configuring preferences

Creating new preferences

GPS contains a number of predefined preferences to configure its behavior and appearance, which are all customizable through the *Edit* → *Preferences* menu.

You can add preferences for your extension modules through the usual GPS customization files. Preferences are different from project attributes (see *Defining project attributes*); the latter varies depending on which project is loaded by the user, while preferences are always set to the same value independent of what project is loaded.

You create your own preferences with the `<preference>` tag, which accepts the following attributes:

- `name` (required)

  Name of the preference, used both 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. These names cannot contain spaces or underscore characters: use minus signs instead of the latter.

- `page` (optional, default **`General`**)

  Name of the page in the preferences editor where the preference are edited. If the page does not already exist, GPS automatically creates it. If this is the empty string, the preference is not editable interactively. Use this to save a value from one session of GPS to the next without allowing the user to change it. Subpages are reference by separating pages name with slashes (`/`).

- `default` (optional, default depends on type of the preference)

  Default value of the preference. If not specified, this is 0 for integer preferences, the empty string for string preferences, True for boolean preferences, and the first possible choice for choice preferences.

- `tip` (optional)

  Text of the tooltip that appears in the preferences editor dialog.

- `label` (required)

  Name of the preference as it appears in the preferences editor dialog

- `minimum` (optional, default **`0`**), `maximum` (default **`10`**)

  Minimum and maximum values for integer preferences.

- `type` (required)

  Type of the preference. Must be one of:

  - **boolean**
  - **integer**
  - **string**
  - **font**
  - **color**

    A color name, in the format of a named color such as "yellow", or a string like "#RRGGBB", where RR is the red component, GG is the green component, and BB is the blue component.

  - **choices**

    The preference is a string whose value is chosen among a static list of possible values, each of which is defined in by 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 &amp;amp; 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 used by GPS itself when it saves the preferences edited via the preferences dialog.

This tag requires one attribute, `name`, the name of the preference for which you are setting a default value. These names are defined when the preference is registered in GPS. You can find them 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 tags, but the value of the `<pref>` tag defines the default value of the preference, which is used unless the user has overridden it in his own preferences file.

Any setting you defined in the customization files is overridden by a specification of that preference in the user's preferences file.

The `<pref>` tag is mostly intended for use in themes (see *Creating themes*).

13.5.10 Creating themes

You can create your own themes and share them between users and then selectively chose which themes each user want to activate through the preferences dialog.

You create new themes in the customization files using the `<theme>` tag.

This tag accepts the following attributes:

- `name` (required)

  Name of the theme as it appears in the preferences dialog

- `description` (optional)

  This text should explain what the theme does. It appears in the preferences dialog when the user selects that theme.

- `category` (optional, default **`General`**)

  Name of the category in which the theme should be presented in the preferences dialog. Categories are currently only used to organize themes graphically. GPS creates a category automatically if you choose one that has not previously been created.

This tag accepts any other customization tags including setting preferences (`<pref>`), defining key bindings (`<key`), and defining menus (`<menu>`).

If you define the same theme in multiple locations (either multiple times in the same customization file or in different files), the customizations in each are merged. The first definition of the theme seen by GPS sets the description and category for the theme.

All child tags of the theme are executed when the user activates the theme in the preferences dialog. There is no strict ordering of the child tags. The default 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.

Here is an example of a theme:

```
<?xml version="1.0" ?>
<my-plugin>
   <theme name="my theme" description="Create a new menu">
       <menu action="my action"><title>/Edit/My Theme Menu</title></menu>
   </theme>
</my-plugin>
```

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. For example, *Ada assignment*.

Define your own search patterns in the customization files using the `<vsearch-pattern>` tag. This tag can have the following child tags:

- `<name>`

  String displayed in the search dialog to represent the new pattern. This is the text the user sees (instead of the often hard-to-understand regular expression)

- `<regexp>`

  Regular expression to use when the pattern is selected by the user. Be careful to protect reserved XML characters such as < and replace them by their equivalent expansion (**`&lt;`** in that case).

  This tag accepts one optional attribute, `case-sensitive` a boolean that specifies whether the search should distinguish lower case and upper case letters. The default is **`false`**.

- `<string>`

  A constant string that should be searched. Provide either `<regexp>` or `<string>`, but not both. If both are provided, the first `<regexp>` child found is used. The tag accepts the same optional attribute `case-sensitive` as above.

Here is a short example, showing how the *Ada assignment* pattern is 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 Defining custom highlighters

The mechanism here described allows any user to add syntax highlighting to GPS for any language in a declarative domain specific language.

Tutorial: Add support for python highlighting in GPS

In this short tutorial, we will walk through the steps needed to create a small plugin for GPS that will allow it to highlight python code.

The idea of the whole API is for the user to declare matchings in a declarative way, specifying the matcher via a classic regular expression syntax, and taking the appropriate action depending on the kind of the matcher. There are basically two types of matches:

- Simple matchers will just apply a tag to the matched text region. This will be useful to highlight keywords or number expressions in source, for example.
- Region matchers will change the set of matchers to the one specified in the region definition. That way, you can do more complex highlighters in which some simple matchers will work only in some context.

In addition to that, you have a set of helpers that will simplify common patterns based on those two primitives, or make some additional things possible. See the full API doc below for more details.

IMPORTANT NOTE: As you will see, the way you register an highlighter is by specifying the language it applies to in the call to register_highlighter. If you want to highlight a language that is not yet known to GPS, you have to register a new language. The way to do that is detailled in the *Adding support for new languages* section.

First step, creating a dumb highlighter

As a first step, we will just create an highlighter that highlights the self symbol in python, as a simple hello world.:

```
from highlighter.common import *

register_highlighter(
    language="python",
    spec=(
        # Match self
        simple("self", tag=tag_keyword),
    )
)
```

As we can see, the first step to register a new highlighter is to call the `register_highlighter()` function, giving the name of the language and the spec of the language as parameters.

The spec parameter is a tuple of matchers. In this case we have only one, a simple matcher, as described above, which will match the "self" regexp, and apply the "keyword" tag everytime it matches.

The tag parameter is the name of the tag that will be used to highlight matches . GPS has a number of built-in tags for highlighting, that are all defined in the `highlighter.common` module. They may not be sufficient, so the user has the possibility of creating new styles, a capability that we will talk about later on.

Second step, discovering our first helper

Highlighting just self is a good first step, but we would like to be a little more pervasive in our highlighting of keywords. Fortunately for us, python has a way to dynamically get all the language's keywords, by doing:

```
from keywords import kwlist
```

By combining that with the `words()` helper, we can easily create a matcher for every python keyword:

```
register_highlighter(
    language="python",
    spec=(
        # Match keywords
        words(kwlist, tag="keyword"),
    )
)
```

The `words()` helper just creates a simple matcher for a list of words. `words(["a", "b", "c"], tag="foo")` is equivalent to `simple( "a|b|c", tag="foo")`.

Third step, highlighting strings literals in a clever way

Next, we're gonna want to highlight some literals. Let's start by strings, because they are hard and interresting. A string is a literal that starts with a " or a ' character, and ends with the same character, but one needs to be careful because there are several corner cases:

- If an escaped string delimiter occurs in the string (" in a " string for example), it should *not* end the string !
- In python, strings delimited by " or ' are single line strings. It means that the match needs to be terminated at the end of the line
- BUT, if the last character of the line is a backslash, the string actually continues on the next line !

Additionally, some editors are nice enough to highlight escaped chars in a specific colors inside string literals. Since we want our highlighter to be cutting edge, we will add this requirement to the list. Here is the region declaration for this problem, for the case of single quoted strings:

```
string_region = region(
    r"'", r"'|[^\]$", tag="string",
    highlighter=(
        simple(r"\.", tag=tag_string_escapes),
    )
)
```

Here are the important points:

- The first parameter is the regular expression delimiting the beginning of the region, in this case a simple quote.
- The second parameter is the regular expression delimiting the end of the region, in this case, either a simple quote, either an end of line anchor ($). This way, a string will be terminated after a new line.
- The way both line continuations and escaped quotes are handled is actually very simple: The simple matcher declared inside the region's highlighter will match any character preceded by a backslash, including newlines. An important point to understand is that, when inside a region, the matcher for ending the region has the lowest priority of all. In this case, it means the simple matcher will consume both quotes and new lines if they are preceded by a backslash, and so they won't be available for the ending matcher anymore.

Creating custom style tags

If the style tags predefined in the `highlighter.common` module are not enough, you can define new ones with the `new_style()` function.

When you define a new style via this function, a corresponding preference will be created in the GPS preferences, so that the user can change the color later.

The tag_string_escapes common tag is defined with this function this way:

```
tag_string_escapes = new_style(lang="General", name="string_escapes",
                               foreground_colors=('#875162', '#DA7495'))
```

The first parameter is the name of the language for which this applies, or "General" if this can potentially apply to several languages. This will be used by GPS to choose which preference category will be used for the corresponding preference.

The second parameter is the name of the style.

The third parameter is the colors that will be used by default for this style. The first color is the one used for light themes, the second color is the one used for dark themes.

Going further

All the details of the engine are not yet documented, but if while creating your highlighter you find yourself stuck, don't hesitate to look at the C or Python highlighters, in the c_highlighter and python_highlighter modules that are shipped with your version of GPS. Those are complete real world examples that are used by GPS to highlight files in those languages.

API Documentation

highlighter.interface.**existing_style**(*pref_name*, *name=''*, *prio=-1*)

Creates a new style to apply when a matcher succeeds, using an existing style as a basis. This probably should not be used directly, but one should use one of the existing styles declared in `Highlighter.common()`

Parameters

- **pref_name** (*string*) – The name of the preference to bind to the style
- **name** (*string*) – The name of the style, used for the underlying gtk tag
- **prio** (*int*) – The priority of the style compared to others. Higher priority styles will take precedence over lower priority ones. -1 means default priority: tags added last have precedence.

Return type highlighter.engine.Style

highlighter.interface.**new_style**(*lang*, *name*, *label*, *doc*, *foreground_colors*, *background_colors=('white', 'white')*, *font_style='default'*, *prio=-1*)

Creates a new style to apply when a matcher successfully matches a portion of text. A style is the conflation of

•An editor tag with corresponding text style

•A user preference that will be added to the corresponding language page

Parameters

- **lang** (*string*) – The language for which this style will be applicable . This is used to automatically store the preference associated with this style in the right preferences subcategory.
- **name** (*string*) – The name of the style, used to identify it.
- **label** (*string*) – The label that will be shown in the preferences dialog for this style.
- **doc** (*string*) – The documentation that will be shown in the preferences dialog for this style.
- **foreground_colors** (*string, string*) – The foreground colors of the style, expressed as a tuple of two CSS-like strings, for example ("#224488", "#FF6677"). The first color is used for light themes, the second is used for dark themes
- **background_colors** (*string, string*) – The background colors of the style.
- **font_style** (*string*) – : The style of the font, one of "default", "normal", "bold", "italic" or "bold_italic"
- **prio** – The priority of the style. This determines which style will prevail if two styles are applied to the same portion of text. See `Highlighter.region()` -1 means default priority: tags added last have precedence.

Return type highlighter.engine.Style

highlighter.interface.**region**(*start_re*, *end_re*, *tag=None*, *name=''*, *highlighter=()*, *matchall=True*, *igncase=False*)

Return a matcher for a region, which can contain a whole specific highlighter

Parameters

- **start_re** (*string*) – The regexp used to match the start of the region
- **end_re** (*string*) – The regexp used to match the end of the region
- **tag** (*highlighter.engine.Style*) – The Tag which will be used to highlight the whole region. Beware, if you plan to apply other tags to elements inside the region, they must have an higher priority than this one !

Return type RegionMatcher

highlighter.interface.**region_ref**(*name*)

Used to reference a region that already exists. The main and only use for this is to define recursive regions, eg. region that can occur inside themselves or inside their own sub regions. See the tutorial for a concrete use case.

The returned region reference will behave exactly the same as the original region inside the highlighter.

Parameters **name** – The name of the region.

Return type RegionRef

highlighter.interface.**region_template**(**args*, ***kwargs*)

Used to partially construct a region, if you want to define for example, several regions having the same sub highlighter and tag, but not the same start and end regular expressions.

Parameters

- **args** – Positional params to pass to region
- **kwargs** – Keyword params to pass to region

Returns A partially constructed region

highlighter.interface.**register_highlighter**(*language*, *spec*, *igncase=False*)

Used to register the declaration of an highlighter. See the tutorial for more information

Parameters

- **language** (*string*) – The language to be used as a filter for the highlighter.
- **spec** (*tuple*) – The spec of the highlighter.

highlighter.interface.**search_for_capturing_groups**(*regexp_string*)

Return a list of matches for capturing groups in a regular expression.

Parameters **regexp_string** (*str*) – The regular expression we want to analyze.

highlighter.interface.**simple**(*regexp_string*, *tag*)

Return a simple matcher for a regexp string. Raises an exception if capturing groups are present in the regular expression (not supported by the engine).

Parameters **regexp_string** (*str*) – The regular expression for this matcher

Return type SimpleMatcher

highlighter.interface.**words**(*words_list*, ***kwargs*)

Return a matcher for a list of words

Parameters **words_list** (*str|list[str]*) – The list of words, either as a string of "|" separated words, or as a list of strings.

Return type SimpleMatcher

highlighter.common.**tag_block = <Style : blocks_hl>**
Style for blocks

Type Style

highlighter.common.**tag_comment = <Style : comments_hl>**
Style for comments

Type Style

highlighter.common.**tag_comment_notes = None**
Style for notes in comments. Used for highlighting TODO and NOTE strings in comments.

Type Style

highlighter.common.**tag_default = <Style : default_hl>**
Default style

Type Style

highlighter.common.**tag_keyword = <Style : keywords_hl>**
Style for keywords

Type Style

highlighter.common.**tag_number = <Style : numbers_hl>**
Style for numbers

Type Style

highlighter.common.**tag_string = <Style : strings_hl>**
Style for strings

Type Style

highlighter.common.**tag_string_escapes = None**
Style for escapes in strings, such as n or t

Type Style

highlighter.common.**tag_type = <Style : types_hl>**
Style for types

Type Style

13.5.13 Adding support for new languages

You have two ways of defining a new language in GPS:

- Basic support from registering languages in Python is provided.
- If the support provided in Python is not enough, more extensive support is provided via the XML interface. With time all capabilities will be provided in the Python interface, and the XML facility will be deprecated.

Adding support for a new language via the Python interface

You can register a new language in Python via the class `GPS.Language`. The first step is to define a new subclass of GPS.Language, the second is to register it via a call to `GPS.Language.register`. Here is an example

```
class JavaLang(GPS.Language):
    def __init__(self):
        pass

GPS.Language.register(JavaLang(), "java", ".java", "", "", INDENTATION_SIMPLE)
```

The class is provided to provide the possibility of future further user customization for a specific language.

For the moment, the support is rudimentary. This is mostly useful if you want to then register an highlighter for the language in question via the new highlighters API, see *Defining custom highlighters*.

Adding support for a new language via the XML interface

Define new languages in a custom file by using the `<Language>` tag. Defining languages gives GPS the ability to perform language-specific operations such as highlighting the syntax of a file, exploring a file using the *Project* view, and finding files associated with that language.

NOTE: The highlighting of syntax via the mechanisms described here are deprecated. See *Defining custom highlighters* for the current way to highlight custom languages.

The following child tags are available:

- `<Name>`

  Short string giving the name of the language.

- `<Parent>`

  Optional name of language that provides default values for all properties not explicitly set.

- `<Spec_Suffix>`

  String identifying the filetype (including the '.' character) of spec (definition) files for this language. If the language does not have the notion of spec or definition file, you should use the `<Extension>` tag instead. Only one such tag is permitted for each language.

- `<Body_Suffix>`

  String identifying the filetype of body (implementation) files for this language. Only one such tag is permitted for each language.

- `<Obj_Suffix>`

  String identifying the extension for object files for this language. For example, it is `.o` for C or Ada and `.pyc` for Python. The default is **–**, which indicates there are no object files.

- `<Extension>`

  String identifying one of the valid filetypes for this language. You can specify several such children.

- `<Keywords>`

  Regular expression for recognizing and highlighting keywords. You can specify multiple such tags, which will all 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>` tag. If you specified a parent language, you can append to the parent `<Keywords>` by providing a `mode` attribute set to **`append`** (the default for `mode` is **`override`**, where the `<Keywords>` definition replaces the one from the parent).

  You can find the full grammar for regular expression 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. If you want to also include other special characters (for example < and > in XML), use this tag. The value of this tag is a string consisting of all the special characters that may be present in keywords. You need not include letters, digits or underscores.

- <Context>

  Information that GPS uses to determine the syntax of a file for highlighting purposes. The following child tags are defined:

  - <Comment_Start>, <Comment End>

    Strings that determine the start and end of a multiple-line comment.

  - <New_Line_Comment_Start>

    A regular expression defining the beginning of a single line comment that ends at the next end of line. This regular expression may contain multiple possibilities, such as **;|#** for comments starting after a semicolon or after the pound sign. If you specified a parent language, you can append to the parent's <New_Line_Comment_Start> by including a mode attribute with a value of **append** (the default is **override**, meaning the <New_Line_Comment_Start> definition replaces the one in the parent).

  - <String_Delimiter>

    Character defining the string delimiter.

  - <Quote_Character>

    Character defining the quote (also called escape) character, used to include the string delimited inside a string (\ in C).

  - <Constant_Character>

    Character defining the beginning of a character literal, in languages that support such literals (e.g., C).

  - <Can_Indent>

    Boolean indicating whether indentation is enabled. The indentation mechanism is 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>

    Boolean indicating whether the language syntax should be highlighted and colorized.

  - <Case_Sensitive>

    Boolean indicating whether the language (in particular the identifiers and keywords) is case sensitive.

  - <Accurate_Xref>

    Boolean indicating whether cross reference information for this language is fully accurate or whether it is either an approximation or not present). Default is **False**.

  - <Use_Semicolon>

    Boolean indicating whether semicolons are expected in sources and can be used as a delimiter for syntax highlighting purposes. Default is **False**.

- <Categories>

  Optional tag to describe the categories supported by the *Project* view. This tag contains a list of <Category> tags, each describing the characteristics of a single category, with the following child tags:

  - <Name>

    Name of the category, either one of the predefined categories or a new name, in which case GPS will create a new category.

    The predefined categories are **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**, and **exception_handler**.

  - <Pattern>

    Regular expression to select a language category. Like <Keywords> tags, if you specify multiple <Pattern> tags, GPS will concatenate them into a single regular expression.

  - <Index>

    Index of the subexpression in the pattern that extracts the name of the entity in this category.

  - <End_Index>

    Optional tag providing the index of the subexpression used to start the next search. The default is the end of the pattern.

- <Project_Field>

  Information about the tools used to support this language. The name of these tools is stored in the project files so you can specify only a limited number of tools. This tag is currently only used by the project properties and wizard and not by other components of GPS.

  This tag two attributes:

  - Name

    Name of the attribute in the project file. Currently, you can only specify **compiler_command**.

  - Index

    If present, specifies the index to use for the attribute in the project file. The line defining this attribute looks 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 attribubte is the string to use in the project properties editor when editing this project field.

here is an example of a 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>
```

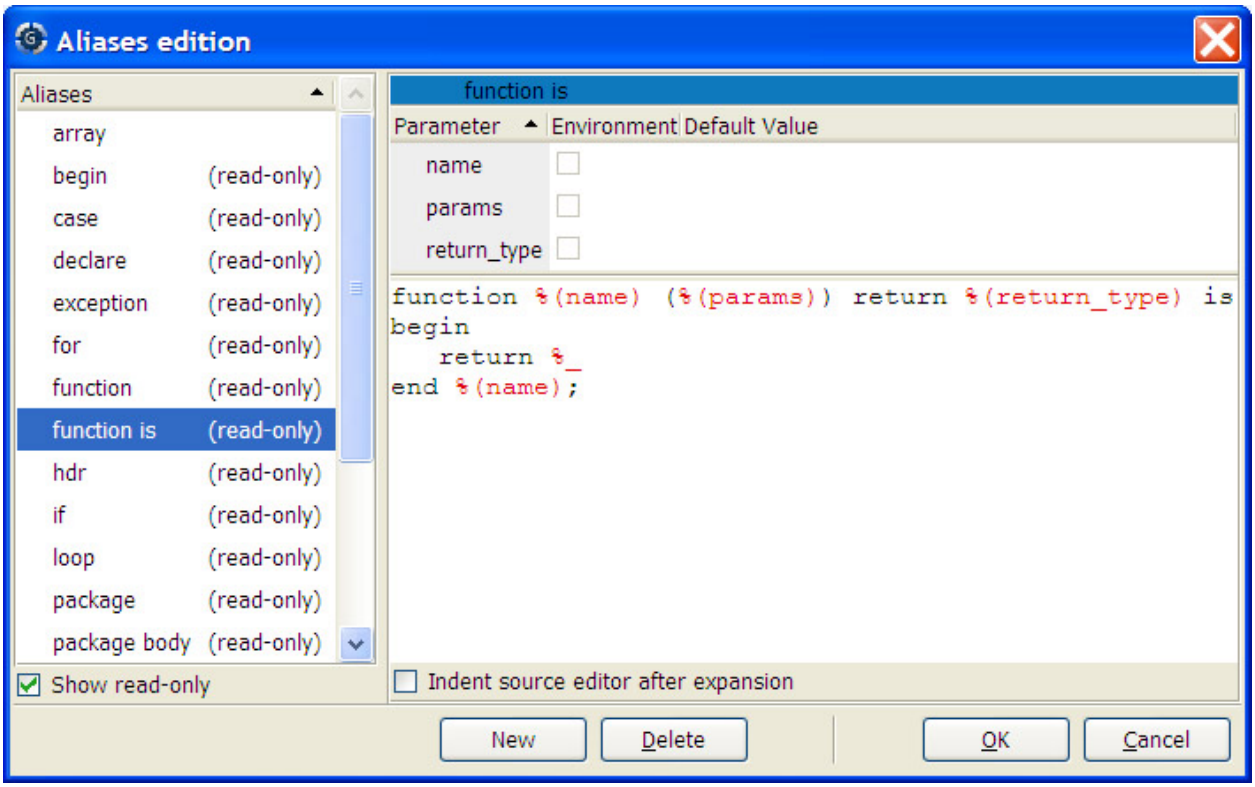
```
    <Context>
      <New_Line_Comment_Start>--</New_Line_Comment_Start>
      <String_Delimiter>&quot;</String_Delimiter>
      <Constant_Character>&apos;</Constant_Character>
      <Can_Indent>True</Can_Indent>
      <Syntax_Highlighting>True</Syntax_Highlighting>
      <Case_Sensitive>False</Case_Sensitive>
    </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>
  </Language>
</Custom>
```

13.5.14 Defining text aliases

GPS provides a mechanism known as "aliases". The user can define these using the *Edit->Aliases* menu.

Each alias has a name, generally a short string of characters. When you type that string in any textual entry in GPS (usually a source editor, but also entry fields anywhere, such as in a file selector) and press the special activation key (by default `Ctrl-o`, but controlled by a preference), GPS replaces the string by the text you have associated with it.

Alias names can contain any character except newline but must start with a letter. GPS looks backward to the start of the word before the current cursor position, and if the characters between there and the cursor position is an alias name (using a case insensitive comparison), it expands the alias.



The alias editor is divided into three main parts. The left shows the list of currently defined aliases. Clicking on any of them displays its replacement text. If you click a second time, GPS displays a text entry allowing you to rename that alias. A checkbox at the bottom determines whether the editor displays read-only aliases (i.e., system-wide aliases).

The second part displays the expansion text for the alias, at the bottom right corner. This text can span multiple lines and contain macros, which are displayed in a different color. You can insert these macros either by typing their symbols (as shown below) or by right-clicking in the editor and selecting the entity in the contextual menu.

The alias editor supports the following macros:

- `%_`

  Position in the replacement text where the cursor is placed.

- *`%name`*

  Name of a parameter. **`name`** can contain any characters except closing parenthesis. See below for more information on parameters.

- `%D`

  Current date, in ISO format. The year is displayed first, then the month and day.

- `%H`

  Current time (hour, minutes, and seconds).

- `%O`

  For recursive aliases expansion. This macro expands the text before it in the current replacement of the parameters and possibly other recursive expansions. This is similar to pressing `Ctrl-o` in the expansion of the alias.

  You cannot expand an alias recursively when already expanding that alias. If the alias expansion for, e.g., **`procedure`** contains **`procedure%O`**, the inner procedure is not expanded.

- `%%`

  A percent sign.

The remaining macros are only expanded if the alias is being expanded in a source editor:

- `%l`

  Line on which the cursor is when pressing `Ctrl-o`.

- `%c`

  Like `%l`, except the current column.

- `%f`

  Name of current file (its base name only, not including directory).

- `%d`

  Directory containing current file.

- `%p`

  Base name of the project file referencing the current file.

- `%P`

  Like `%p`, but the full name of the project file (directory and base name).

GPS preserves the indentation of the alias when it is expanded. All lines are indented the same as the alias name. You can override this default behavior by selecting the checkbox *Indent source editor after expansion*. In that case, GPS replaces the name of the alias by its expansion and then recomputes the position of each line with its internal indentation engine as if the text had been inserted manually.

The third part of the alias editor, the top right corner, lists the parameters for the currently selected alias. Whenever you insert a `%name` string in the expansion text, GPS detects new, changed, or deleted parameter references and updates the list of parameters to show the current list.

Each parameter has three attributes:

- `name`

  The name you use in the expansion text of the alias in the `%name` macro.

- `Environment`

  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 value. 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 first displays a dialog to ask for the value of the parameters. You can interactively enter this value, which replaces all corresponding `%name` entities in the expanded text.

13.5.15 Alias files

Customization files can also contain alias definitions that can be used to create project or system- wide aliases. All customization files are parsed to look for aliases definitions. All customization files are treated as read-only by GPS and therefore cannot be edited through its graphical interface. You can override some of the aliases in your own custom files. The system files are loaded first and aliases defined there can be overridden by the user-defined file. There is

one specific files which must contain only aliases definitions: `$HOME/.gps/aliases`. Whenever you edit aliases graphically or create new ones, they are stored in this file, which is the only one GPS ever modifies automatically.

These files are standard XML customization files. The XML tag to use is `<alias>`, one per new alias. The following example contains a standalone customization file, though you may wish to merge the `<alias>` tag into any other customization file.

The following child tags are supported:

- `<alias>`

  Indicates the start of a new alias. It has one mandatory attribute, `name`, the text to type before pressing `Ctrl-o`, and one optional attribute, `indent`, which, if set to **`true`** tells GPS to recompute the indentation of the newly inserted paragraph after the expansion.

- `<param>`

  One per alias parameter. It has one mandatory attribute, `name`, the name in `%{name)` in the alias expansion text, and two optional attributes: `environment`, indicating whether or not the default value must be read from the environment variables and `description`, a string that is displayed when asking the parameter's value while expanding the alias.

- `<text>`

  Replacement text, possibly multiple lines.

Here is an example of an alias definition in a configuration 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>
```

13.5.16 Defining project attributes

Project files are required by GPS and store various pieces of information related to the current set of source files, including how to find the source files and how the files should be compile or manipulated through various tools.

The default set of attributes used by GPS in a project file is limited to those attributes used by tools packaged with GPS or GNAT. If you are delivering your own tools, you may want to store similar information in the project files, since they are a very convenient location to associate specific settings with a given set of source files.

GPS lets you manipulate the contents of projects through XML customization files and script commands. You can add your own typed attributes into the projects and have them saved automatically when the user saves the project and reloaded automatically when GPS reloads the project.

Declaring the new attributes

You can declare new project attributes in two ways: either using the advanced XML tags below or the `<tool>` tag (see *Defining tool switches*).

The customization files support the file:*<project_attribute>* tag, used to declare attributes GPS should support in a project file. Attributes that are not supported by GPS are not accessible through the GPS scripting languagues and generate warnings in the *Messages* window.

Each project attributes has a type typed and can either have a single value or have a set of values (a list). Each value can be a free-form string, a file name, a directory name, or a value extracted from a list of preset values.

Attributes declared in these customization files are also graphically editable through the project properties dialog or the project wizard. When you define an attribute, you need to specify how it is presented to the GPS user.

The `<project_attribute>` tag accepts the following attributes:

- `package` (string)

  Package in the project file containing the attribute. Good 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` (string, required)

  Name of the attribute. A string with no space that represents a valid Ada identifier (typically starting with a letter and be followed by a set of letters, digits or underscores). This is an internal name used when saving the attribute in a project file.

- `editor_page` (string, default **`General`**)

  Name of the page in the *Project Properties* editor dialog in which the attribute is presented. If no such page exists, GPS creates one. If the page already exists, the attribute is appended to the bottom of those already on the page.

- `editor_section` (string)

  Name of the section, if any, inside the editor page where the attribute is displayed. These sections are surrounded by frames, the title of which is given by the this attribute. If not present, the attribute is put in an untitled section.

- `label` (string, default: name of the attribute)

  Textual label displayed to the left of the attribute in the graphical editor used to identify the attribute. However, it can be set to the empty string if the attribute is in a named section of its own, since the title of the section may be good enough.

- `description` (string)

  Help message describing the role of the attribute, displayed as a tooltip if the user hovers over the attribute.

- `list` (boolean, default **`false`**)

  If **`true`**, the project attribute contains a list of values, as opposed to a single value. An example is the list of source directories in standard projects.

- `ordered` (boolean, default **`false`**)

  Only relevant if the project attribute contains a list of values, when it indicates whether the order of the values is relevant. In most cases, it is not. However, the order of source directories, for example, matters since it also indicates where GPS searches for the source files, and it stops at the first match.

- `omit_if_default` (boolean, default **`true`**)

  Whether the project attribute should be set explicitly in the project if the user left it with its default value. Enable this to keep the project files as simple as possible if all the tools using this attribute know about the default value. Otherwise, set it **`false`** to always emit the definition of the project attribute.

- `base_name_only` (boolean, default **`false`**)

If the case of attributes that are a file or directory name, whether the base name (**true**) or the full path (**false**) is stored. In most cases, the full path is best. However, since GPS looks for source files in the list of directories the list of source files, for example, should only contain base names. This also increases the portability of project files.

- `case_sensitive_index` (**true**, **false** (default), or **file**)

  Only relevant for project attributes that are indexed on another attribute (see below for more information on indexed attributes). It indicates whether two indexes that differ only by their casing are considered the same. For example, 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".

  The value **file** indicates that the case sensitivity is the same as the filesystem of the local host. Use that value when the index is a filename.

- `hide_in` (string)

  Context in which GPS will not allow graphical editing of this attribute. 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 contexs are specified, GPS will not display the widget to edit this attribute. Use this to keep the graphical interface simple.

- `disable_if_not_set` (boolean, default **false**)

  If **true**, the field to edit this attribute is greyed out if the attribute is not explicitly set in the project. In most cases, you will not specify this, since the default value of the attribute can populate that field. However, when the value of the attribute is automatically computed depending on other attributes, you cannot specify the default value in the XML file, and it might be simpler to grey out the field. A checkbox is displayed next to the attribute so the user can choose to enable the field and add the attribute to the project.

- `disable` (space-separated list of attribute names)

  List of attribute whose fields should be greyed out if this attribute is specified. This only works if both the current attribute and the referenced attributes all have their `disable_if_not_set` attribute set **true**. Use this to create mutually exclusive attributes.

Declaring the type of the new attributes

The type of the project attribute is specified by child tags of `<project_attribute>`. The following tags are recognized:

- `<string>`

  Attribute is composed of a single string or a list of strings. This tag accepts the following XML attributes:

  - `default` (string)

    Default value of the attribute. If the attribute's type is a file or directory, the default value is normalized: an absolute path is generated based on the project's location, with **"."** representing the project's directory. As a special case, if `default` is surrounded by parenthesis, no normalization is done so you can on test whether the user is still using the default value.

    Another special case is when you specify **project source files**, which is replaced by the known list of source files for the project. However, this does not work from the project wizard, since the list of source files has not been computed yet.

  - `type` (empty string (default), **file**, **directory**, or **unit**)

    What the string represents. In the default case, any value is valid. For **file**, it should be a file name, although no check is done to ensure the file actually exists. Similarly, **directory** tells GPS to expect a directory. For **units**, GPS should expect the name of one of the project's units.

  - filter (**none**, **project**, **extending_project**, **all_projects**)

    Ignored for all types except **file**, where it further specifies what type of files should be specified by this attribute. If **none**, any file is valid. If **all_projects**, files from all projects in the project tree are valid. If **project**, only files from the selected project are valid. If **extended_project**, only the files from the project extended by the current project can be specified. This attribute is not shown if the current project is not an extension project.

  - allow_empty (boolean, default **True**)

    Whether the value for this attribute can be an empty string. If not and the user does not specify a value, GPS will display an error message in the project properties editor and project wizard.

- <choice>

  One of the valid values for the attribute. Use multiple occurrences of this tag to provide a static list of such values. If combined with a <string> tag, indicates that the attribute can be any string, although a set of possible values is provided to the user. This tag accepts one optional XML attribute, default, a boolean which indicates whether this value is the default. If several details attributes are present the default value of the attribute is a list, as opposed to a single value.

- <shell>

  GPS scripting command to execute to get a list of valid values for the attribute. Like the <choice> tag, this 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 any valid is valid.

  The <shell> tag accepts two attributes:

  - lang (string, default **shell**)

    Scripting language in which the command is written. The only other possible value is **python**.

  - default (string)

    Default value of the attribute if the user has not specied one.

Sometimes either the type of the project attribute or its default value depends on what the attribute applies to. The project file supports this in the form of indexed project attributes. This, for example, is used to specify the name of the executable generated when compiling each of the main files in the project (e.g., the executable for gps.adb is gps.exe and the one for main.c is myapp.exe).

You can also declare such attributes in 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 accepts one of the already mentioned <string>, <choice>, or <shell> tags as children.

The <index> tag specifies 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 containing the list of main files for the project.

It accepts the following XML attributes:

- attribute (string, required)

  Name of the other attribute, which must be declared elsewhere in the customization files and whose type must be a list of values.

- package (string)

  Package in which the index project attribute is defined. This is used to uniquely identify attributes with the same name.

Use the <specialized_index> tag to override the default type of the attribute for specific values of the index. For example, project files contain an attribute specifying the name of the compiler for each language, which is indexed on the project attribute specifying the language used for each source file. Its default value depends on the language

(**gnatmake** for Ada, **gcc** for C, etc.). This attribute requires one XML attribute, `value`, which is the value of the attribute for which the type is overridden.

Almost all the standard project attributes are defined through an XML file, `projects.xml`, which is part of the GPS installation. Examine this file for advanced examples on declaring project attributes.

Examples

The following declares three attributes, each with a single string as their value. This string represents a file in the first case and a directory in the last two:

```
<?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 declares an attribute whose value is a string. However, it provides list of predefined possible values as an help for the user. If the `<string>` tag was not specified, the attribute's value could only 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 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, the example uses a script command 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 declares an attribute whose value is a set of file names. The order of files in this list matters to the tools using this 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 declares an attribute whose value is a set of predefined values. By default, two such values are selected, unless the user overrides the default:

```
<?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 declares an attribute whose value is a string. However, the value is specific to each language (it could, for example, be the name of a compiler to use for that language). This is an indexed attribute, with two default values, one for Ada and 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 project attributes

Attributes you define are accessible from the GPS scripting languages like all the standard attributes, see *Querying project switches*.

For example, you can 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 example, by calling:

```
GPS.Project.root().get_attribute_as_list ("main")
```

13.5.17 Adding casing exceptions

You can use the customization files to declare a set of case exceptions by using the `<case_exceptions>` tag. Put each exception in child tag of `<word>` or `<substring>`. GPS uses these exceptions to determine the case of identifiers and keywords when editing case insensitive languages (except if corresponding case is set to **Unchanged**). Here is an example:

```
<?xml version="1.0" ?>
<exceptions>
```

```
  <case_exceptions>
     <word>GNAT</word>
     <word>OS_Lib</word>
     <substring>IO</substring>
  </case_exceptions>
</exceptions>
```

13.5.18 Adding documentation

You can add new documentation to GPS in various ways. You can create a new menu, through a `<menu>` tag in a configuration file, associated with an action that either spawns an external web browser or calls the function `GPS.Help.browse()`. However, this will not show the documentation in the *Help* → *Contents* menu, which is where people expect to find it. To do both, use the `<documentation_file>` tag. These tags are usually found in a `gps_index.xml` file, but are permitted in any customization file.

Your documentation files can contain the usual HTML links. In addition, GPS treats links starting with '%' specially and considers them as script commands to execute instead of files to display. The following examples show how to insert a link that, which clicked by the user, opens a file in GPS:

```
<a href="%shell:Editor.editor g-os_lib.ads">Open runtime file</a>
```

The first token after '%' is the name of the language; the command to execute is after the ':' character.

The `<documentation_file>` tag accepts the following child tags:

- `<name>`

  Name of the file, either an absolute filename or a filename relative to one of the directories in `GPS_DOC_PATH`. If this child is omitted, you must specify a `<shell>` child. The name can contain a reference to a specific anchor in the HTML file, using the standard HTML syntax:

  ```
  <name>file#anchor</name>
  ```

- `<shell>`

  Name of a shell command to execute to get the name of the HTML file. This command could create the HTML file dynamically or download it locally using some special mechanism. This child accepts one attribute, `lang`, the name of the language in which the command is written

- `<descr>`

  Description for this help file, which appears as a tool tip for the menu item.

- `>category>`

  Used in the *Help* → *Contents* menu to organize all documentation files.

- `<menu>`

  Full path to the menu. See *Adding new menus*. If not set, GPS does not display a menu item for this file, although it still appears in the *Help->Contents* menu

  This tag accepts two attributes, `before` and `after`, that control the position of the menu item. If the new menu is inserted in a submenu, the attribute controls the deeper nesting. Parent menus are created as needed, but if you wish to control their display order, create them first with a `<submenu>` tag.

The following example creates a new entry *item* in the *Help* menu, that displays `file.html` (searched for in the `GPS_DOC_PATH` path):

```
<?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>
```

The directories given by the GPS_DOC_PATH environment variable are searched for the HTML documentation files. However, you can also use the <doc_path> XML node to define additional directories to search. Such a directory is relative to the installation directory of GPS. For example:

```
<?xml version="1.0"?>
<GPS>
   <doc_path>doc/application/</doc_path>
</GPS>
```

adds the directory <prefix>/doc/application to the search path for documentation.

Youc an also add such a directory via Python, as in:

```
GPS.HTML.add_doc_directory ('doc/application')
```

13.5.19 Adding custom icons

You can also provide custom icons to be used throughout GPS in places such as buttons, menus and toolbars.

Images must be either in the PNG or SVG format. The latter (scalable vector graphic) is preferred, since the image will always display sharply whatever size is used on the screen. GPS itself always uses SVG icons.

The images are searched in multiple base directories:

- Any directory mentioned in the environment variable GPS_CUSTOM_PATH.
- HOME/.gps/icons
- <gps_install>/share/gps/icons

In all these cases, icons can be in either the directory itself, or in subdirectories named hicolor/48x48/apps, with the following conventions:

- hicolor is the name of the icon theme. The default is 'hicolor', and that cannot be changed from GPS itself.
- 48x48 is the size of the icon. This is only relevant to PNG images, in case you want to provide multiple sizes for the image. The directory name should match the size of the icon, and GPS will automatically select the most appropriate format when it needs to display the image. For SVG images, can you instead choose a subdirectory named hicolor/scalable/16x16, where the final size does not matter since these images can always be resized to any size.

Icons are referenced with the basename of the file (no directory info) with no extension. gtk+ will automatically try a number of variants like name.svg, name.png, name-rtl.svg, name-symbolic.svg, ...

If you name your icon name-symbolic.svg, GPS will automatically change the foreground and background colors to match the selected color theme by the user (dark or light). But these icons are only displayed in grayscale.

As shown in the example above, you should prefix the icon with a unique name, here `my-vcs-`, to make sure predefined icons do not get overridden by your icons.

So for instance, if you have put a file `mylogo.png` in `/dir/plug-ins/`, then you should do the following:

- set *GPS_CUSTOM_PATH* to include '/dir/plug-ins/'
- use *iconname="mylogo"* in your plug-in

Further information about icons could be found in a separate document - Icon Theme Specification.

13.5.20 Customizing Remote Programming

There are two parts to specifying the configuration of remote programming functionality: the configuration of the tools (remote connection tools, shells, and rsync parameters) and the servers.

The first part (see *Defining a remote connection tool*, *Defining a shell*, and *Configuring rsync usage*) is performed by a pre-installed file in the plugins directory called `protocols.xml`.

The second part (see *Defining a remote server* and *Defining a remote path translation*) creates a `remote.xml` file in the user's `gps` directory when the user has configured them (see *Setup the remote servers*). System-wide servers can be also installed.

Defining a remote connection tool

A remote connection tool is responsible for making a connection to a remote machine. GPS already defines several remote access tools: **`ssh`**, **`rsh`**, **`telnet`**, and **`plink`**. You can add support other tools using the tag `<remote_connection_config>`, which requires a `name` attribute giving the name of the tool. This name need not necessarilly correspond to the command used to launch the tool.

The following child tags are defined:

- `<start_command>` (required)

  The command used to launch the tool. This tag supports the `use_pipes` attribute, which selects on Windows the manner in which GPS launchs the remote tools and accepts the following values:

  - **`true`**

    Use pipes to launch the tool.

  - **`false`** (default)

    Use a tty emulation, a bit slower but allows password prompt retrieval with some tools.

- `<start_command_common_arg>`

  Arguments provided to the tool. This string can contain the following macros, which are replaced by the following strings:

  - **`%C`**: Command executed on the remote host (normally the shell command).
  - **`%h`**: Remote host name.
  - **`%U`**: Value of `<fstart_command_user_args>`, if specified.
  - **`%u`**: User name.

  If you have not included either **`%u`** or **`%U`** in the string and the user specifies a username in the remote connection configuration, GPS places the value of `<start_command_user_args>` at the beginning of the arguments.

- <start_command_user_args>

  Arguments used to define a specific user during connection. **%u** is replaced by the user name.

- <send_interrupt>

  Character sequence to send to the connection tool to interrupt the remote application. If not specified, an Interrupt signal is sent directly to the tool.

- <user_prompt_ptrn>, <password_prompt_ptrn>, <passphrase_prompt_ptrn>

  Regular expressions to detect username, password, and passphrase prompts, respectively, sent by the connection tool. If not specified, appropriate defaults are used.

- <extra_ptrn>

  Used to handle prompts from the connection tool other than for username, password or passphrase. The auto_answer attribute selects whether GPS provides an answer to this prompt or asks the user. If **true**, a <answer> child is required. Its value is the answer to be supplied by GPS. If **false**, a <question> child is required. Its value is used by GPS to ask the user a question. Provide this child once for every prompt that must be handled.

Defining a shell

GPS already defines several shells: **sh**, **bash**, **csh**, **tcsh**, and, on Windows, cmd.exe). You can add other shells by using the <remote_shell_config> tag which has one required attribute, name, denoting the name of the shell. This name need not be same as the command used to launch the shell.

The following child tags are defined:

- <start_command> (require)

  Command used to launch the shell. Put any required arguments here, separated by spaces.

- <generic_prompt> (optional)

  Regular expression used to identify a prompt after the initial connection. If not set, a default value is used.

- <gps_prompt> (required)

  Regular expression used to identify a prompt after the initial setup is performed. If not set, a default value is used.

- <filesystem> (required)

  Either **unix** or **windows**, representing the filesystem used by the shell.

- <init_commands> (optional)

  Contains <cmd> children, each containing a command to initialize a new session.

- <exit_commands> (optional)

  Like <init_commands>, but each <cmd> child contains a command to exit a session.

- <no_echo_command> (optional)

  Command used to tell the remote shell to suppress echo.

- <cd_command> (require)

  Command to change directories. **%d** is replaced by the directory's full name.

- <get_status_command> (required)

  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>. You must include a single pair of parentheses; that subexpression identifies the status.

Configuring rsync usage

GPS includes native support for the **rsync** tool to synchronize paths during remote programming operations.

By default, GPS uses the **--rsh=ssh** option if **ssh** is the connection tool used for the server. It also uses the **-L** switch when transfering files to a Windows local host.

You can define additional arguments to rsync by using the <rsync_configuration> tag, which accepts <arguments> tags as children, each containing additional arguments to pass to **rsync**.

Defining a remote server

Users can define remote servers, as described in *Setup the remote servers*. Doing this creates a remote.xml file in the user's gps directory, which can be installed in any plugins directory to set the values system-wide. The tag used in this file is <remote_machine_descriptor> for each remote server. You can also write this tag manually. Its attributes are:

- nickname (required)

  Uniquely identifies the server.

- network_name (required)

  Server's network name or IP address.

- remote_access (required)

  Name of the remote access tool used to access the server. These tools are defined in *Defining a remote connection tool*.

- remote_shell (required)

  Name of the shell used to access the server. See *Defining a shell*.

- remote_sync (required)

  Remote file synchronisation tool used to synchronize files between the local host and the server. Must be **rsync**.

- debug_console (optional)

  Boolean that indicates whether GPS displays a debug console during the connection with a remote host. Default is **false**.

The optionally child tags for this tag are:

- <extra_init_commands>

  Contains <cmd> children whose values are used to set server-specific initialization commands.

- max_nb_connections

  Positive number representing the maximum number of simultaneous connections GPS is permitted to launch.

- timeout

  Positive number representing a timeout value (in ms) for every action performed on the remote host.

Defining a remote path translation

The user can also define a remote path translation, as described in *Setup the remote servers*. Each remote paths translations corresponds to one `<remote_path_config>` tag, which has one required attribute, `server_name`, the server name that uses this path translation, and contains child `<remote_path_config>` tags, that have the following required attributes:

- `local_path`

  Absolute local path, written using local filesystem syntax.

- `remote_path`

  Absolute remote path, written using remote filesystem syntax.

- `sync`

  Synchronization mechanism used for the paths (see *Path settings*). Must be one of **`NEVER`**, **`ONCE_TO_LOCAL`**, **`ONCE_TO_REMOTE`**, or **`ALWAYS`**.

13.5.21 Customizing Build Targets and Models

You can customize the information displayed in *The Target Configuration Dialog* and in the *Mode selection* via the XML configuration files.

Defining new Target Models

Define a model with a `target-model` tag, which has one attribute, `name`, containing the name of the model, and which supports the following child tags:

- `<iconname>` (required)

  Name of the icon associated by default with targets of this model. See *Adding custom icons*.

- `<description>` (required)

  One-line description of what the model supports.

- `<server>` (default **`Build Server`**)

  Server used for launching targets of this model. See *Remote operations*.

- `<is-run>` (default **`False`**)

  Whether targets of this model correspond to the launching of an executable instead of performing a build. GPS launches such targets using an interactive console and does not parse their output for errors.

- `<uses-shell>` (default **`False`**)

  Whether GPS should launch targets of this model with the shell pointed to by the `SHELL` environment variable.

- `<uses-python>` (default **`False`**)

  When this is set to :command‘True‘, launch a Python command rather than an external process. In this case, the arguments in the command line are first process using the macro replacement mechanism, and then concatenated to form the string which is interpreted. For instance the following:

```
<command-line>
   <arg>GPS.Console("Messages").write("</arg>
   <arg>%PP</arg>
```

```
   <arg>")</arg>
</command-line>
```

Is interpreted as:

```
GPS.Console("Messages").write("<full path to the project>")
```

- <command-line> (required)

  Contains <arg> child tags, each containing an argument of the default command line for this model, beginning with the executable name.

- <persistent-history> (default **True**)

  Whether GPS should keep command line history over GPS sessions. If set to False, GPS provide history of command lines during current session only and will reset command line to default value after restart.

- <switches>

  Description of the switches. (See *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>
```

Aditionally, switches defined for target models accept a `filter` attribute, allowing you to define when a switch is relevant or not (e.g: switch only defined for newer versions of the executable to launch).

The `filter` attribute accepts any named filter: predefined ones or custom filters defined in XML via the *<filter>* tag.

Here is a simple example showing how to define filters for target model switches:

```
<GPS>
   <!-- filter checking that the tool's version supports the switch
        by calling a python function that actually verifies it -->
   <filter name="Is_My_Tool_Version_Supported" shell_lang="python"
       shell_cmd="check_my_tool_version(GPS.current_context())"/>

   <my_model>
      <target-model name="my_tool" category="">
         <description>Model for targets based on 'my_tool'</description>
         <command-line>
            <arg>my_tool</arg>
         </command-line>
```

```
         <switches command="%(tool_name)s" columns="1">
            <check label="vesrion specific switch"
                   switch="--version-specific-switch"
                   tip="This switch is only supported by newer versions"
                   filter="Is_My_Tool_Version_Supported"/>
         </switches>
      </target-model>
   </my_model>
</GPS>
```

Defining new Targets

Define targets with a `<target>` tag, which has three attributes:

- `name`

  Name of the target. It must be a unique name. Underscores are interpreted as menu mnemonics. If you want an actual underscore, use a double underscore.

- `category`

  Category containing the target for purposes of ordering the tree in the *Target Configuration Dialog* and *Build* menus. Underscores are interpreted as menu mnemonics. If you want an actual underscore, use a double underscore. If the string begins and ends with an underscore, GPS places the menu for the target in the toplevel *Build* menu.

- `messages_category`

  Name of category to organize messages in the *Locations* view.

- `model`

  Name of the initial model that this target inherits.

This tag accepts the following child tags:

- `<iconname>`

  Name of the icon associated by default with targets of this model. See *Adding custom icons*.

- `<in-toolbar>` (default **`False`**)

  Whether the target has an associated icon in the toolbar.

- file:*<in-menu>* (default **`True`**)

  Whether the target has an associated entry in the *Build* menu.

- `<in-contextual-menus-for-projects>` (default **`False`**)

  Whether the target has an associated entry in the contextual menu for projects.

- `<in-contextual-menus-for-files>`

  Likewise, but for files.

- `<visible>` (default **`True`**)

  Whether the target is initially visible in GPS.

- `<read-only>` (default **`False`**)

  Whether the user can remove the target.

- `<target-type>`

  If present, a string indicating whether the target represents a simple target (empty) or a family of targets. The name is a parameter passed to the **`compute_build_targets`** hook. If set to **`main`**, a new subtarget is created for each main source defined in the project.

- `<launch-mode>` (default **`MANUALLY`**)

  How GPS should launch the target. Possible values are **`MANUALLY`**, **`MANUALLY_WITH_DIALOG`**, **`MANUALLY_WITH_NO_DIALOG`**, and **`ON_FILE_SAVE`**.

- `<server>` (default **`Build_Server`**)

  Server used for launching Target. See *Remote operations*.

- `<command-line>`

  Contains a number of *<arg>* nodes, each with an argument of the default command line for this target, beginnng with the name of the executable:

```
<?xml version="1.0" ?>
<my_target>
   <target model="gprclean" category="C_lean" 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>
```

- `<output-parsers>`

  Optional list of output filters. See *Processing Target's Output* for details.

Processing Target's Output

You can filter output produced by a target's run by using custom code. The list of filters already provided by GPS is shown below. By default, each is executed during each run of a target.

- `output_chopper`

  Breaks output stream to pieces. Each of the piece contains one or more line of output and an end of line.

- `utf_converter`

  Converts the stream to UTF-8 encoding if output is not in UTF-8.

- `progress_parser`

  Drives GPS's progress bar by looking for progress messages in the output stream. It excludes such messages from the stream.

- `console_writer`

  Populates the GPS console with output from the stream.

- location_parser

  Looks for special patterns in output to extract messages associated with processed files and locations and sends such messages to *Location* view (see *The Locations View*).

- text_splitter

  Splits output into separate lines to simplify further processing.

- output_collector

  Aggregates output and associates it with the build target. As result, the output is available for scripting (see GPS.get_build_output()) after the build completes:

```
text = GPS.get_build_output(<name of your target>)
```

- elaboration_cycles

  Detects the **gnatbind** report about circles in elaboration dependencies and draws them in the *Elaboration Circularities* browser (see *The Elaboration Circularities browser*).

- end_of_build

  Cleans up internal data after a build run.

See GPS.OutputParserWrapper for examples of writing custom filter.

Defining new Modes

Define modes with a <builder-mode> tag which has one attribute, name, containing the name of the model. It supports the following child tags:

- <description>

  One-line description of what the mode does

- <subdir>

  Optional base name of the subdirectory to create for this mode. GPS will subsitute the macro arguments %subdir in the <extra-args> tags with this value.

- <supported-model>

  Name of a model supported by this mode. You can provide multiple tags, each corresponding to a supported model and optionally specify a filter attribute corresponding to the switches used for this mode. By default, all switches are considered. GPS passes the <extra-args> of the mode matching filter to commands of the supported models.

- <extra-args>

- sections

  Optional attribute sections contains spaceseparated list of switches delimiting a section of a command line (such as **-bargs -cargs -largs**). See more details in *Defining tool switches*.

  List of <arg> tags, each containing one extra argument to append to the command line when launching targets while this mode is active. Optional attribute section sets section of given argument. Macros are supported in the **<*arg*>** nodes:

```
<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 sections="-cargs">
     <arg>--subdirs=%subdir</arg>
     <arg section="-cargs">-O2</arg>
  </extra-args>
 </builder-mode>
</my_mode>
```

13.5.22 Customizing Toolchains

You can customize the list of toolchains and their values presented in the project editor (see *The Project Wizard*) with the XML configuration files. GPS's default list is contained in `toolchains.xml`. You can add your own toolchain by providing an XML description with the following tags:

- `<toolchain_default>`

  Default names for the different tools used by all toolchains. The final name used is `toolchain_name-default_name`.

- `<toolchain>`

  Defines a toolchain, with an attribute, `name`, giving the name of the toolchain, which overrides the default values defined in `<toolchain_default>`.

Each of the above tags can have the following child tags:

- `<gnat_driver>`

  GNAT driver to use.

- `<gnat_list>`

  GNAT list tool to use.

- `<debugger>`

  Debugger to use.

- `<cpp_filt>`

  Reserved.

- `<compiler>`

  Requires a `lang` attribute naming an language and defines the compiler to use to compile that language.

You can override (including by setting the value to null) any value in the `<toolchain_default>` tag by providing the same tag withing a `toolchain` tag.

13.6 Adding support for new tools

GPS has built-in support for many external tools. This list of tools is frequently enhanced, so if you are planning to use the external tool support in GPS, check the latest GPS version available.

You can use this feature to support additional tools (in particular, different compilers). You need to do following to successfully use a tool:

- Specify its command line switches.
- Pass it the appropriate arguments depending on the current context and on user input.
- Spawn the tool.
- Optionally parse its result and act accordingly.

Each of these is discussed below. In all 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*). Tool descriptions are found in `<tool>` tags, which accept the following attributes:

- `name` (required)

  Name of the tool. This is purely descriptive and appears throughout the GPS interface whenever this tool is referenced, for example the tabs of the switch editor.

- `package` (default **`ide`**)

  Which package is used in the project to store information about this tool, including its switches. You should use the default value unless you are using one of the predefined packages.

  See also *Defining project attributes* for more information on defining your own project attributes. Using the XML `package`, `attribute`, or `index` attributes of `<tool>` implicitly creates new project attributes as needed.

  If **`ide`** is specified, switches cannot be set for a specific file, but only at the project level. Support for file-specific switches currently requires modification of the GPS sources themselves.

- `attribute` (default **`default_switches`**)

  Name of the attribute in the project used to store the switches for the tool.

- `index` (default is the tool name)

  What index is used in the project. This is mostly for internal use by GPS and indicates which index of the project attribute GPS uses to store the switches for the tool.

- `override` (default **`False`**)*

  Whether the tool definition can be redefined. If the tool is defined several times GPS will display a warning.

This tag supports the following child tags, each described in a separate section:

- *<switches>*
- *<language>*
- *<initial-cmd-line>*

13.6.1 Defining supported languages

A tool supports one or more languages. If you do not specify any language, the tool applies to all languages and the switches editor page is displayed for all languages. If at least one language is specified, the switches editor page will only be displayed if that language is supported by the project.

Specify the languages that the tool supports using the `<tool>` tag:

```
<my_tool>
  <tool name="My Tool" >
    <language>Ada</language>
    <language>C</language>
```

```
  </tool>
</my_tool>
```

13.6.2 Defining the default command line

You can define the command line to be used for a tool when the user is using the default project and has not overridden the command line in the project. Do this with the `<initial-cmd-line>` tag, as a child of the `<tool>` tag. Its value is the command line to be passed to the tool. This command line is parsed in the usual manner and quotes are used 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 must be able to specify which switches are passed to the tool. If the tool is only called through custom menus, you can hardcode some or all of the switches. However, it is usually better to use the project properties editor so the user can specify project-specific switches.

This is what GPS does by default for Ada, C, and C++. Look at the GPS installation directory to see 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, allowing the user to edit the command line both as text and through GUI widgets.

In customization files, the switches are declared with the `<switches>` tag, which must be a child of a `<tool>` tag as described above. Use this tag to produce the needed GUI widgets to allow a user to specify the desired switch value.

This tag accepts the following attributes:

- `separator`

  Default character placed between a switch and its value, for example, **=** produces **-a=1**. Can can override this separately for each switch. If you want the separator to be a space, you must use the value ** ** instead of a blank since XML parser will normalize the latter to the empty string when reading the XML file.

- `use_scrolled_window` (default **False**)

  Whether boxes of the project editor are placed into scrolled window. This is particularily useful if the number of displayed switches is large.

- `show_command_line` (default **True**)

  If **False**, the command line is not displayed in the project properties editor. Use this, for example, if you only want users to edit the command line through the buttons and other widgets but not directly as text.

- `switch_char` (Default **-**)

  Leading character of command line arguments that are considered to be switches. Arguments not starting with this character remain unmodified and do not have graphical widgets associated with them.

- `sections`

Spaceseparated list of switches delimiting a section (such as **-bargs -cargs -largs**). A section of switches is a set of switches that are grouped together and preceded by a particular switch. Sections are always placed at the end of the command line, after regular switches.

The <switches> tag can have any number of child tags, listed below. Repeat them multiple times if you need several check boxes. For consistency, most of these child tags accept the following attributes:

- line (default **1**), column (default **1**)

  This indicates the row or column (respectively) of the frame to contain the switch. See the attributes of the same name above.

- label (required)

  Label displayed in the graphical interface.

- switch (required)

  Text put in the command line if the switch is selected. This text might be modified, see the description of <combo> and <spin> below. The value must not contain any spaces.

- switch-off

  Defined in **<*check*>** tags, where it specified the switch used for deactivating the relevant feature. Use this for features that are enabled by default but can be disabled.

- section

  Switch section delimiter (such as **-cargs**). See the sections attribute of the <switches> tag for more information.

- tip

  Tooltip describing the switch more extensively. Tags accepting this attribute also accept a single child <tip> whose value contains the text to be displayed. The advantage of the latter is that text formatting is retained.

- before (default **false**

  Whether the switch must always be inserted at the begining of the command line.

- min (default **1**), max (default **1**)

  Only supported for <spin> tags. Specifies the minimum or maximum (respectively) value allowed for the switch.

- default (default **1**)

  Used for <check> and <spin> tags. See the description below.

- noswitch, nodigit

  Only valid for <combo> tags and documented there.

- value (required)

  Only valid for **<*combo-entry*>** tags and documented there.

- separator

  Overrides 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>

  Accepts the line and column attributes and used to give a name to a specific frame. The value of the tag is the title. You need not specify a name.

Use the line-span or column-span attribute to specify how many rows or columns (respectively) the frame should span. The default for both is **1**. If is set to **0**, the frame is hidden from the user. See, for example, the usage in the Ada or C switches editor.

- <check>

  Creates a toggle button. When active, the text defined in the switch attribute is added to the command line. The switch can also be activated by default (the default attribute is **on** or **true**), in which case, deactivating the switch adds the value of switch-off to the command line.

  Accepts the line, column, label, switch, switch-off, section, default, before, and tip attributes, and you can specify an optional <tip> child.

- <spin>

  Adds the contents of the switch attribute followed by the current numeric value of the widget to the command line. One usage is to indicate indentation length. If the current value of the widget is equal to the default attribute, nothing is added to the command line.

  This tag accepts the line, column, label, switch, section, tip, min, max, separator, and default attributes and you can specify an optional <tip> child.

- <radio>

  Groups together any number of children, each of which is associated with its own switch, allowing only one of the children can be selected at any given time.

  This tag accepts the line, column , label, switch, section, before, and tip attributes. Specify an empty value for the switch attribute to indicate the default switch to use in this group of radio buttons. Each child must have the tag radio-entry or <tip>.

- <field>

  Creates a text field, which can contain any text the user types and be editable by the user. This text is prefixed by the value of the switch attribute and the separator character. If the user does not enter any text in the field, nothing is added to the command line.

  You can specify an optional <tip> child tag. This tag accepts the line, column, label, switch, section, separator, before, and tip attributes, and the following additional attributes:

  - as-directory

    If **true**, an extra *Browse* button is displayed, allowing the user to easily select a directory.

  - as-file

    Like as-directory, but opens a dialog to select a file instead of a directory. If both attributes are **true**, GPS displays a file selector.

- <combo>

  GPS inserts the text from the switch attribute, concatenated with the text of the value attribute for the currently selected entry, into the command line. If the value of the current entry is the same as that of the nodigit attribute, only the text of the switch attribute is inserted into the command line. (This is used, for example, to interpret the gcc switch **-O** as **-O1**.) If the value of the current entry is the same as that of the noswitch attribute, nothing is added to the command line.

  This tag accepts the line, column, label, switch, section, before, tip, noswitch, separator, and nodigit attributes and any number of combo-entry child tags, each of which accepts the label and value attribute. You can also include an optional <tip> child.

- <popup>

Displays a button that, when clicked, displays a dialog with some additional switches. This dialog, like 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 those attributes as well as the `label` attribute and any number of child `<switch>` tags.

- `<dependency>`

  Describes a relationship between two switches. For example, when the *Debug Information* switch is selected for *Make*, we need to force it for the compiler as well.

  This tag supports the following additional attributes:

  - `master-page`, `master-switch`, `master-section`

    Define the switch that can force a specific setting for a slave switch. In our example, they 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, you must also specify the `master-section` attribute. If the user selects the check button of the this switch, GPS forces the selection of the check button for the slave switch. Likewise, if user sets the field to any value, GPS sets the slave switch to that same value.

  - `slave-page`, `slave-switch`, :file:‘ slave-section‘

    Likewise, but designates the slave switch. In our example, they have the values **`Ada`** and **`-g`**. The switch referenced by these attributes must be of type `<check>` or `<field>`.

  - `master-status`, `slave-status`

    Which state of the master switch forces which state of the slave switch. In our example, they both have the value **`on`**: when the user enables debug information for **`make`**, GPS also enables compiler debug information. However, if the user does not enable debug information for **`make`**, nothing is changed for the compiler debug information. If you specify **`off`** for `master-status` and the master switch is a field, GPS changes the status of the slave when the user does not specify any value in the master switch's field.

- `<default-value-dependency>`

  Describes a relationship between two switches, which is slightly different from the `<dependency>` tag. This relationsip only affects the default values. For example, when the **`-gnatwa`** switch is selected for the Ada compiler, other switches, such as **`-gnatwc`** and **`-gnatwd`**, are enabled by default. But the user can disable them by specifying, e,g., **`-gnatwC`** and **`-gnatwD`**.

  It supports the following additional attributes:

  - `master-switch`

    Switch that triggers the dependency. If that switch is present in the command line, GPS changes the default status of `slave-switch`.

  - `slave-switch`

    Switch whose default value depends on `master-switch`. This must be a switch already defined in a `<switch>` tag. The switch can match the `switch` or `switch-off` attributes. In the latter case, the slave-switch default value is disabled if the user specifies the `master-switch`.

- `<expansion>`

  Describe how switches are grouped together on the command line to keep it shorter. It also defines aliases between switches.

  It is easier to explain the functioning of this tag with an example. Specifying the GNAT switch **`-gnatyy`** is equivalent to specifying **`-gnaty3abcefhiklmnprst`**. This is a style check switch with a number of default values. But it can also be decomposed it into several switches, such as **`-gnatya`** and **`-gnatyb`**. Knowing this, GPS can keep the command line length as short as possible, making it more readable.

Specify the above details in the <expansion> tag, which supports two attributes: switch is mandatory and alias is optional. In our example, alias contains the text **-gnatyabcefhiklmnprst**.

This tag works in two ways:

- If you do not specify the alias attribute, the switch attribute requests GPS to group all switches starting with that prefix. For example, if you specify **-gnatw** as the value of the switch attribute, if the user selects both the **-gnatwa** and **-gnatw.b** switches, GPS merges them on the command line as **-gnatwa.b**.
- If you specify the alias, GPS views the switch attribute as a shorter way of writting the switch. For example, if switch is **-gnatyy** and alias is "**-gnaty3abcefhiklmnprst**, then if the user types **-gnatyy**, it means the whole set of options.

You can specify the same switch attribute can be used in multiple <expansion> tags nodes if you want to combine their behavior.

For historical reasons, this tag supports <entry> child tags, but these are no longer used.

13.6.4 Executing external tools

Once the user specified the switches to use for the external tool, it can be spawned from a menu item or by pressing a key. Both cases are described in an XML customization file, as described previously, and both execute what GPS calls an action, a set of commands defined by an <action> tag.

Chaining commands

The <action> tag (see *Defining Actions*) executes one or more commands, either internal GPS commands (written in any of the scripting language supported by GPS) or external commands provided by executables found on the PATH.

You can hard-code the command line for each of these commands in the customization file or it can be the result of previous commands executed as part of the same action. As GPS executes each command from the action, it saves its output on a stack. If a command line contains the construct **%1**, **%2**, etc., these constructs are replaced respectively by the result the last command executed, the previous command, and so on. The replacement is done with the value returned by the command, not by any output it might have made to some of the consoles in GPS. Each time GPS executes a new command, it pushes the previous result on the stack, so that, for example, the value of **%1** becomes the value of **%2**.

The result of the previous commands is substituted exactly as is. However, if the output is surrounded by quotes, GPS ignores them when a substitution is done, so you must put them back if needed. This is done because many scripting languages systematically protect their output with quotes (simple or double) and these quotes are often undesired when calling further 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>
```

Saving open windows

Before launching the external tool, you may want to force GPS to save all open files. Do this using the same command GPS uses before starting a compilation, **`MDI.save_all`**, which takes one optional boolean argument specifiying whether GPS displays an interactive dialog for the user.

This command fails when the user presses cancel, so you can 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

You can use GPS shell commands 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, 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 in 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 shown as *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 outputs 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. This function queries the Ada compiler switches for the current project and prints them in the *Messages* view:

```
<?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

You can also query the arguments for the tool by asking the user interactively. The scripting languages provides a number of solutions for these, which generally have their own native way to read input, possibly by creating a dialog. The simplest solution is to often use the predefined GPS commands:

- `yes_no_dialog`

  This takes a single argument, a question to display, and presents two buttons to the user, *Yes* and *No*. The result of this function is the button the user selected, as a boolean value.

- `input_dialog`

  This function is more general. It takes a minimum of two arguments. 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, each querying 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 does not have support for lists and would return a concatenation of the values. However, this function is especially useful in 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* view:

```
<?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

By default, GPS sends the output of external commands to the *Messages* view. However, you can exercise finer control using the `output` attribute of the `<external>` and `<shell>` tags. You can also specify this attribute in the `<action>` tag, where it defines the default value for all `<shell>` and `<external>` tags.

This attribute is a string. Specifying an empty string (to override a specification in the `<action>` tag)' produces the default behavior. A value of **none** tells GPS to hide the output of the command as well as the text of the command itself and not show it to the user. If you specify any other value, GPS creates a new window with the title given by the attribute. If such a window already exists, it is cleared before any 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:

```
<?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

Once the output of the tool has either been hidden or made visible to the user in one or more windows, you can do several additional things with this output, for further integration of the tool in GPS.

- Parsing error messages

  External tools usually display error messages for the user that are associated with specific locations in specific files. For example, the GPS builder itself analyzes the output of **`make`** using this information.

  You can do this done for your own tools using the shell command `Locations.parse`, which 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 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 typing *help Locations.parse* in the GPS Shell).

  Here is a short example showing how to run a make command and send the errors to the *Locations* view.

  For languages that support it, it is recommended that you quote the argument with triple quotes (see *The GPS Shell*), so that any special character such as newlines and quotes in the output of the tool are not specially interpreted by GPS. You should also 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

  GPS supports automatically correcting errors for some of languages. You can get access to this auto-fixing feature through the `Codefix.parse()` shell command, which takes the same arguments as `Locations.parse()`. This automatically adds pixmaps to the relevant entries in the *Locations* view, so you should call `Locations.parse()` before 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 those errors that can be fixed for a given session. To do so, first get a handle for that section, as shown in the example below. Then get the list of fixable errors through the `errors()` command.

  Each error is of the class `CodefixError`, which has one important method, `fix()`, allowing you to perform an automatic correction of 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*, that contains a menu item named *item 1*, associated with the external command `external-command 1` and a sub menu named *other menu*:

```
<?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 of how you can define 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 the 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 asks 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 currently includes support for two scripting languages, although additional languages may be added in the future. These two languages are described in the following section.

Support for scripting is currently a "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 those who extend GPS directly in Ada, but with major advantages: they do not require recompilation of the GPS core and can be tested and executed interactively. The goal of such scripts is to help automate processes such as builds and 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 displays a prompt, at which you can type interactive commands. These consoles provide completion of the command names through the `tab` key.

For example, in the GPS shell console you can start typing:

```
GPS> File
```

then press the `tab` key, which lists all functions whose name starts with `File`.

A similar feature is available in the Python console, also providing completion for all the standard Python commands and modules.

All the scripting languages share the same set of commands exported by GPS, because of 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.

You can execute scripts immediately upon startup of GPS by using the command line switch **`--load`**. Specifying the following command line:

```
gps --load=shell:mytest.gps
```

forces the GPS script `mytest.gps` to be executed immediately, before GPS starts responding to user's requests. Do this if you want to preform some initializations of the environment. It can also be used as a command line interface to GPS, if your script's last command is to exit GPS.

The name of the language is optional, and defaults to python:

```
gps --load=script.py
```

You can also specify in-line commands directly on the command line through **`--eval`** command line switch.

For example, 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'
```

The language defaults to python, as for **`--load`**.

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. You can bind actions to menus and keys through the customization files or the *Edit* → *Key shortcuts* dialog. These actions can execute any script command (see *Defining Actions*) using the <*shell*> XML tag.

But the opposite is also true. From a script, you can execute any action registered in GPS. For example, you can split windows or highlight lines in the editor when no equivalent shell function exists. You can use this to execute external commands if the scripting language does not support this easily. 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 shell commands take subprograms as parameters. If you are using the GPS shell, you to pass the name of a GPS action. If you are using Python, you pass a subprogram. See *Subprogram parameters*.

13.8.3 The GPS Shell

Warning: The GPS Shell is deprecated, and only accessible through XML commands now, for backward compatibility. Don't try to use it for any new development, and use the Python Shell instead.

The GPS shell is a very simple-minded, line-oriented language. It is not interactively accessible in GPS anymore.

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/ for documentation on 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 *Tools* → *Consoles* → *Python* menu. The full documentation on what GPS makes visible through Python is available from the *Help* → *GPS* → *Python extensions* menu.

The same example as shown for the GPS shell follows, now using Python. As you notice, the name of the commands is similar, although they are not run exactly in the same way. Specifically, GPS uses 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 *Locations* view:

```
>>> 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 most GPS functions 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>
```

You can also 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']
```

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 you can use this Python feature to make your scripts more readable. (A notable exception are functions that allow a variable number of parameters.) Using named parameters, you can 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

GPS automatically imports (with Python's **import** command) all files with the extension `.py` found in the directory `$HOME/.gps/plug-ins`, the directory `$prefix/share/gps/plugins` or in the directories pointed to by `GPS_CUSTOM_PATH` on startup. 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, you 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.

Because GPS uses the `import()` command, functions defined in this modules are only accessible by prefixing their name by the name of the file in which they are defined. For example, if a file `mystartup.py` is copied to the startup directory and defines the function `func()`, the latter is accessible in GPS as `mystartup.func()`.

Python's own mechanism for loading files at startup (using 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 for other people to use, you need to provide the Python files themselves. This is a set of `.py` files, which the user should install in the `plugins` directory.

To make the Python functions accessible through GPS, you can:

- Export the APIs directly through Python, under the form of Actions (the `Action` class), Menus (the `Contextual` and `Menu` classes) or toolbar buttons (the `ToolButton` and `Toolbar` classes).
- Write an XML that creates a set of actions using the `<action>` tag (see *Defining Actions* and which is 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 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 `Ctrl-c n` and can be distributed as a single Python file:

```
# This code can be stored in a file test.py in $HOME/.gps/plug-ins
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 you might want to use, but more importantly that show how GPS can be extended from Python.

If your script does not do what you expect it to do, there are several ways to debug it. The easiest is probably to add some **print** statements. Since some output of the scripts is sometimes hidden by GPS (for example, for interactive commands), you might not see this output. In that 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 include the Python traces as part of the general traces available in the file `$HOME/.gps/log`. Note that this may slow down GPS if there is a lot of output to process.

13.8.6 Subprogram parameters

Some 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

  You cannot define new functions in the GPS shell. However, this concept is similar to the GPS actions (see *Defining Actions*), which allow you to execute a set of commands and launch external processes. A subprogram parameter in the GPS shell is a string, the name of the action to execute.

  For examle, the following code defines the action `on_editing`, which is called each 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_editing">
               <shell>echo "File edited"</shell></action>"""
  Hook "file_edited"
  Hook.add %1 "on_editing"
  ```

- 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_editing(self, *arg):
      print "File edited"
  GPS.Hook("file_edited").add(on_editing)
  ```

  The situation is slightly more complex if you want to pass methods as arguments. Python has three notions of callable subprograms, detailed below. The following examples all create a combo box in the toolbar that 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 also no extra parameter.

However, 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 illustrates, there is no direct access to `MyClass` when executing `on_changed()`. An easy workaround is the following, in which the global data is stored in the instance of `MyClass` and therefore 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. You can make it more convenient by using 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.

Note 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()
```

You may find it convenient to use the object-oriented approach when writing Python scripts. If, for example, 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 looks 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 with 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 possible 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()`, a bound method, as a callback. That tells Python it should automatically and transparently add an extra first parameter, `self()`, when calling `MyClass.on_match()`. This is why the first example has four parameters for `on_match()`.

However, the second example only has three parameters, because GPS detected that `self()` (the instance of `MyClass`) and the instance of `GPS.Process()` are the same in this case. So it need not ada an extra parameter (`self()` and `process()` would have been the same).

13.8.7 Python FAQ

This section lists some 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, diplays a dialog saying the famous 'Hello World!'.

Here is the code that you need to put in `hello_world.py`:

```
import GPS
import gps_utils

@gps_utils.interactive(menu='/Help/Hello World!')
def hello_world():
   GPS.MDI.dialog("Hello World!")
```

To use this plugin, launch GPS with the following command line:

```
$ gps --load=python:hello_world.py
```

If want the plugin to be loaded every time you launch GPS without having to specify it on the command line, copy `hello_world.py` to your `$HOME/.gps/plug-ins/` directory (`%USERPROFILE%\.gps\` on Windows). Alternatively, you can add the directory containing your plugin 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 are various mechanisms to spawn external processes from a script:

- Use the functionalities provided by the `GPS.Process` class.
- Execute a GPS action through `GPS.execute_action()`.

  The 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 does not provide a full interaction with the process.

- Use a standard **expect** library in 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, since GPS's consoles do not fully emulate all the primitive functions of that class (there is no file descriptor, for example).

  When possible, we recommend using one of the methods above instead.

Redirecting the output of spawned processes

In general, you can redirect the output of any Python script to any GPS window (either an already existing one or one GPS creates automatically) using 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 instead of the usual Python output that GPS has overridden.

There are two solutions for this:

- Execute the external process in 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 example 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 that way, like:

```
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 you can using 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 is not 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 is 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 contains information about the project the file or directory belongs to. Thus this implicit filter is not sufficient to hide your contextual menu.

As a result, you need to do a slightly more complex test, where you check that the current context does not 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 will 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 the above for such a simple filter. GPS provides a predefined filter for just that purpose, so 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, GPS displays the output of all Python commands in the Python console. However, you might, in some cases, want to create other windows in GPS for this output. This can be done in one of two ways:

- Define a new action

  If the entire output of your script should be redirected to the same window or if the script is used interactively through a menu or a key binding, the easiest way is to create a new XML action and redirect its 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 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 within your script where the output should be sent (for example if you cannot 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 redirects all input and output to a new window, which is created if it does not yet exist. Note however that the output of `stderr()` is not redirected: you need to explicitely 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 console.

  You can alternatively create separate objects for the output and use them explictly:

```
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 **%N** parameters if the command is executed in a GPS action. It should normally be 1, except for `stderr()` when it should be 0.

Reloading a Python file in GPS

After you have made modification to a Python file, you may want to reload it. This requires careful use of Python commands. Let's assume you have a Python file (`"mymod.py"`) containing the following:

```
GPS.parse_xml ("""
   <action name="my_action">
      <shell lang="python">mymod.myfunc()</shell>
   </action>""")

def myfunc():
   print "In myfunc\\n"
```

This file defines an action `my_action`, that you can, for example, associate with a keybinding through the *Edit* → *Key shortcuts* menu.

If you copy this file into one of the `plugins` directories, GPS automatically loads it at startup. The function `myfunc()` is in a separate namespace, with the name `mymod`, like the file. If you decide, during your GPS session, to edit this file, for exampl to have the function print "In myfunc2" instead, you then to reload the file by typing the following command in the Python console:

```
> execfile ("HOME/.gps/plug-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 the file should be reloaded in the namespace `mymod`.

If you omit the optional second parameter, Python loads the file, but the function `myfunc()` is defined in the global namespace, so the new definition is accessible through:

```
> myfunc()
```

Therefore, the key shortcut you previously set, which still execute `mymod.myfunc()`, will keep executing the old definition.

GPS provides a contextual menu, *Python* → *Reload module* when you are editing a Python file to deal with all the above details.

Printing the GPS Python documentation

The Python extension provided by GPS is fully documented in this manual and in a separate manual accessible through the *Help* menu in GPS. However, this documentation is provided in HTML, and might not be the best format suitable for printing. To generate your own documentation for any Python module, including GPS, and print the result:

```
import pydoc
pydoc.writedoc (GPS)
```

In the last comamnd, "GPS" is the name of the module whose documentation you want to print.

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 you can can use to convert the text file into a Postscript document.

Automatically loading python files at startup

At startup, GPS automatically loads all Python files found in the `share/gps/plugins` and `$HOME/.gps/plug-ins` directories. 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 access to most of its functionality through contextual menus, accessed by right clicking in various parts of GPS. Due to the number of tools provided by GPS, these contextual menus can be large and you might want to control what is displayed in them. There are several ways to do that:

- Define appropriate filters for your actions.

  If you are creating your own contextual menus through customization files and XML, they are are usually associated with actions (*<action>*) you have created. In that case, you need to define filters appropriately, through the *<filter>* tag to decide when the action is relevant and hence when the contextual menu is 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. To do this, you need to find the name of the menu, which can be done by consulting the list returned by `GPS.Contextual.list()`. This name is also the value of the `<title>` tag for contextual menus you have created. Using this name, you can 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 can create your own dialogs and graphical windows using the Python capabilities provided by GPS.

See the *Help → GPS → Python Extensions* menu, specifically the documentation for `GPS.MDI`, for a sample of code on how to create your own graphical interfaces and integrate them with 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, called, for example, when a new project is loaded, or when a file is edited. You can define your own commands to be executed in such cases.

You can find the list of hooks that GPS currently supports by calling the `Hook.list()` function, which takes no argument and returns a list of the names of all hooks. You can get more advanced description for each hook using the *Help → GPS → Python Extensions* menu:

```
GPS> Hook.list
project_changed
open_file_action_hook
preferences_changed
[...]

Python> GPS.Hook.list()
```

The description of each hook includes a pointer to the type of the hook, which is what parameters the subprograms in this hook receive.

You can find the list of all known hook types can be found using the `Hook.list_types()` function, which takes no argument and returns a list of all known types of hooks. You can find more information for each of these type by calling `Hook.describe_type()`.

Adding commands to hooks

Add your own command to existing hooks by calling the `Hook.add()` function. Whenever the hook is executed by GPS or another script, your command is also executed and is passed the parameters that were specified when the hook is run. The first parameter is always the name of the hook being executed.

This function applies to an instance of the hook class and takes one parameter, the command to be executed. This is a subprogram parameter (see *Subprogram parameters*).

- GPS shell

  The command can be any GPS action (see *Defining Actions*). The arguments for the hook will be passed to the action, and are available as **`%N`**. In the following example, the message "Just executed the hook: project_changed" is printed in the *Shell* console. We are defining the action to be executed inline, but it could be defined in a separate XML customization file:

  ```
  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 are passed to this subprogram. In the following example, the message "The hook project_changed was executed by GPS" is 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 illustrates the simplest type of hook, which does not have any arguments. However, most hooks receive several parameters. For example, the `file_edited()` hook receives the file name as a parameter.

- GPS shell

  The following code prints the name of the hook ("file_edited") and the name of the file in the shell console each time a file is openen in 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 file 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

Hooks whose name ends with **_action_hook** are handled specially by GPS. As opposed to the standard hooks described in the previous section, the execution of the action hooks stops if one of the subprograms returns a True value (**1** or **true**). The subprograms associated with that hook are executed sequentially. If any such subprogram knows how to do the the action for that hook, it should do so and return "1".

Other action hooks expect a string as a return value instead of a boolean. Execution stops when a subprogram returns a non-empty string.

This mechanism is used extensively by GPS internally. For example, whenever a file needs to be opened in an editor, GPS executes the `open_file_action_hook()`. 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 spawn the editor and returns 1. This immediately stops the execution of the `open_file_action_hook()`.

However, if user is not using an external editor, this module returns 0, which keep executing the hook, and in particular executes the source editor module, which always takes an actions 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 example. If the user wants to open such a file, you could, for example, spawn an external command (say, **my_editor**) to edit this file, instead of opening it in GPS.

You can do this with 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

Each module in GPS is responsible for running hooks when appropriate. Most of the time the subprograms exported by GPS to the scripting languages properly run the hook. But you might also need to run them in your own scripts.

As usual, this results 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 by the `Hook.run()` function. It applies to an instance of the `Hook` class and has a variable number of arguments, which 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 stops 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 requests the opening of an editor for the file in GPS and 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, though you can also connect to them. But you can also create your own hooks to report events happening in your own modules and programs. In 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 almost any kind of parameters. In the latter case, GPS checks at runtime to ensure that the subprogram called as a result of running the hook has the right number of parameters. If this is not the case, GPS complains and displays error messages. Such general hooks do not pass their parameters to other scripting languages.

You create a new hook by calling `Hook.register()`. This function takes two arguments: the name of the hook you are creating and , optionally, 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.

When the hook type is omitted, it indicates that the hook is of the general type that allows any number of parameter, of any type. Other scripts are 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 passed to `Hook.run()`. Other scripts in other languages only receive the hook name as a parameter, not the full list of parameters.

When specified, the type of the hook must be one of the values returned by `Hook.list_types()`: it indicates that the hook is of one of the types exported by GPS itself. The advantage of using such explicit types instead of **`general`** is that GPS is able to do more testing of the validity of the parameters. Such hooks can also be connected to from other scripting languages.

A small trick worth noting: if the command bound to a hook does not have the correct number of parameters that this hook provides, the command will not be executed and GPS reports an error. You can make sure that your command is always 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 know in advance how many parameters the call of `Hook.run()` provides:

```
## 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 The Server Mode

To give access to the GPS capabilities from external processes (e.g. **emacs**), you can launch GPS in **server mode**.

The relevant command line switches are **--server** and **--hide**. **--server** opens a socket on the specified port, allowing multiple clients to connect to a running GPS and send GPS shell or Python commands. **--hide** tells GPS not to display its main window when starting. On Unix systems, you still need to have access to the current screen (as determined by the `DISPLAY` environment variable) in this mode. Using both switches 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 prompt between each command. This is needed in order to determine when the output (result) of a command is completed. All GPS shell commands (as defined in *The GPS Shell*) are available from this shell, but their use is discouraged, in favor of the use of Python commands. Those are available through the use of the **python** prefix before a Python command.

For example, sending **pwd** through the socket sends the **pwd** command through the GPS shell and sends the result to 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**

  Inform the GPS server that the connection should be closed.

- **id <string>**

  Register the current session with a given string. This string can then be used within GPS itself (for example via a `.xml` or Python plugin) to display extra information to the client via the socket, using the function `GPS.Socket().send()`.

For example, suppose we start GPS with the **--server=1234** command: this brings 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 is time to logout\\n");
```

At this point, the following is received on the client (telnet) side:

```
GPS>> hello, it is time to logout
```

We can then close the client:

```
logout
Connection closed by foreign host.
```

13.10 Adding project templates

The *Project* template wizard lists a selection of templates. GPS locates the default set in the `share/gps/templates` directory of your GPS installation.

You can register new directories in which GPS looks for templates by using the Python function `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 you have registered with `GPS.ProjectTemplate.add()`. Then, in this directory, create one template description file, which is a text file with the `.gpt` extension and 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 are specified:

- `<name>`

  Name of the template as it appears in the template tree in the *Project* template wizard.

- `<category>`

  Category in which the template is inserted in the template tree. There can be multiple levels of categories, separated with `/`.

- `<variable_1>`

  Name substituted in the template files when deploying the template; see below.

- `<variable_1_default_value>`

  Default value for variable 1, which appears in the project template wizard.

- `<variable_1_description>`

  Description of variable 1.

- `<optional_hook_line>`

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 is run by GPS once, immediately after it deploys the project template.

- <description>

  Short paragraph describing the project template. This paragraph is displayed in the *Project* template wizard when the user selects the template in the tree.

When deploying templates, GPS copies all files and directories present in the directory containing the template description file (except the Python file indicated as post_hook() and the template description file itself) into the destination directory chosen by the user.

As it deploys templates, GPS replaces strings of the form **<variable_name>** with the value of the variable. If **<variable_name>** is all lower case, the substitution is converted to lower-case. If **<variable_name>** is in mixed case, the substitution is converted into mixed case as well. If it is in upper case, the substitution contains the original value specified by the user.

CHAPTER

FOURTEEN

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

   --config=file        Specify the configuration file (.cgpr) to load
   --autoconf           Generate .cgpr automatically if needed
   --configdb=dir       Extra directories for gprconfig
```

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 :command'+line' prefix, e.g. **`gps +40 source.adb`**.

`tools_host` corresponds to a remote host's nickname as defined in *Setup the remote servers*.

By default, files you specify on the command line can have absolute or relative pathnames. If you prepend a filename with the = character, GPS looks for the file in the source search path of the project. If you do not specify a project on the command line, GPS tries to find one. Otherwise, it displays the *welcome dialog*.

14.2 Environment Variables

You can set the following environment variables to override default settings in GPS:

- GPS_HOME

  Overrides 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

  Sets the search path for the documentation. See *Adding documentation*.

  If you installed GPS in a directory different from that of the GNAT compiler, you need to set this variable for GPS to find the documentation for GNAT. In the case of the compiler documentation, for example, the gps_index.xml file installed with GPS assumes *GPS_DOC_PATH* points to the directory containing gnat_ugn.html, so it should contain gnat_prefix/share/doc/gnat/html.

- 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. The convention is to have two spaces between the name and the e-mail, such as "John Does <john.doe@home.com>"

- GPS_STARTUP_PATH

  Contains the value of the **PATH** environment variable just before GPS was started. GPS uses this to restore the proper environment before spawning applications independently of what directories it needs to put into its own path.

- 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 looks for libraries in the subdirectory lib/python<version> of the directory specified.

- GNAT_CODE_PAGE

  You can set this variable to CP_ACP or CP_UTF8. It 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, it may be necessary to set this variable to CP_ACP which is the default Windows ANSI code page.

- GPS_ROOT

  Overrides and hardcodes the default root installation directory. You usually do not need to set this variable unless you are a GPS developer in unusual circumstances. GPS finds all its resource files (e.g., images, plugins, and xml files) from this variable, so setting it to an incorrect value will cause GPS to misbehave.

- GPS_MEMORY_MONITOR

  If set, GPS adds special code on every allocation and deallocation to make it possible to check where the largest amount of memory is allocated using the **GPS.debug_memory_usage** Python command. Setting this variable will slow GPS down.

14.3 Files

- `$HOME/.gps`

  GPS state directory. Defaults to `C:.gps` under Windows systems if the **HOME** or **USERPROFILE** environment variables are not defined.

- `$HOME/.gps/log`

  Log file automatically created by GPS. When GPS is running, it creates a file `log.<pid>`, where `<pid>` is the GPS process id, so multiple GPS sessions do not clobber each other's log. In case of a successful session, this file is renamed to `log` when exiting; in case of an unexpected exit (when bug box is displayed) the log file retains its original name. The name of the log file is configured by the `traces.cfg` file.

- `$HOME/.gps/aliases`

  File containing user-defined aliases (see *Defining text aliases*).

- `$HOME/.gps/plug-ins`

  Directory containing files with user-defined plugins. GPS loads all XML and Python files found under this directory during start up. Create or edit these files to add your own menu and/or tool-bar entries in GPS or to define support for new languages. See *Customizing through XML and Python files* and *Adding support for new languages*.

- `$HOME/.gps/key_themes/`

  Directory containing user defined key themes (XML files). These themes are loaded through the Keyshortcuts editor.

- `$HOME/.gps/keys6.xml`

  Contains all key bindings for the actions defined in GPS or custom files. This file only contains the key bindings overridden through the key shortcuts editor (see *The Key Shortcuts Editor*).

- `$HOME/.gps/gps.css`

  Configuration and theme file for gtk. This file can 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 (created using the *File* → *Save More* → *Desktop* menu). It is loaded automatically if found.

- `$HOME/.gps/locations.xml`

  List of locations GPS previously edited. It corresponds to the history navigation (*Navigate* → *Back* and *Navigate* → *Forward*) menus.

- `$HOME/.gps/properties.xml`

  Stores file-specific properties across GPS sessions. In particular, it contains the encoding to use for files where the default encoding is not appropriate.

- `$HOME/.gps/histories.xml`

  Contains the state and history of combo boxes (for example, 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 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 is not recreated the next time you start GPS.

- `$HOME/.gps/startup.xml`

  List of scripts to load at startup as well as additional code that needs to be executed to set up the scripts.

- `$HOME/.gps/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`

  Prefix directory where GPS is installed, e.g `/opt/gps`.

- `prefix/bin`

  Directory containing the GPS executables.

- `prefix/etc/gps`

  Directory containing global configuration files for GPS.

- `prefix/lib`

  Directory containing the shared libraries used by GPS.

- `prefix/share/doc/gps/html`

  GPS looks for all the documentation files under this directory.

- `prefix/share/examples/gps`

  Directory containing source code examples.

- `prefix/share/examples/gps/language`

  Directory containing 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`

  Directory containing the sources used by the GPS tutorial.

  See gps-tutorial.html.

- `prefix/share/gps/support`

  Directory containing required plugins for GPS that are automatically loaded at startup.

- `prefix/share/gps/plug-ins`

  Directory containing files with system-wide plugins (XML and Python files) that are loaded automatically at start-up.

- `prefix/share/gps/library`

  Directory containing files with system-wide plugins (XML and Python files) that are not loaded automatically at startup but can be selected in the *Plugins* section of the preferences editor dialog.

- `prefix/share/gps/key_themes`

  Directory containing the predefined key themes (XML files). These can be loaded through the Key shortcuts editor.

- `prefix/share/gps/gps-splash.png`

  Splash screen displayed by default when GPS is started.

- `prefix/share/gps/perspectives6.xml`

  Description of the default desktop that GPS uses when the user has not defined any default desktop and no project specific desktop exists. You can modify this file if needed, but keep in mind 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 desired.

- `prefix/share/gps/default.gpr`

  Default project used by GPS, which can be modified after installation to provide defaults 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 a user's 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 encounter a bug, please send 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 needed, and/or a scenario describing the actions performed to reproduce the problem as well as lising all the tools (e.g *debugger*, *compiler*, *call graph*) involved.

The files `$HOME/.gps/log` may also bring some useful information when reporting a bug.

If GPS generates a bug box, the log file is kept under a separate name (`$HOME/.gps/log.<pid>` so 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 _UNEXPECTED_EXCEPTION_ Message: gtk-image.adb:281 access check failed

it means either that there is a conflict with `~/.local/share/mime/mime.cache`, in which case removing this file solves this conflict, or that you need to install the **shared-mime-info** package on your system.

Non-privileged users cannot start GPS

If you have originally installed GPS as root and can run GPS successfully, but normal users cannot, 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 replacing *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 *Debug* → *Initialize* menu), it is usually because the underlying debugger (gdb) cannot be launched properly. To verify this is the problem, try to launch the **gdb** command from a shell (i.e., outside of GPS). If you cannot launch **gdb** from a shell, it usually means you are using the 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 wait a little longer, since some communications between GPS and **gdb** can take significant time to finish. If GPS is still not responding after a few minutes, you can usually get control back in GPS by either typing `Ctrl-C` in the shell where you have started GPS, which should unblock it. If that does not work, kill the :*program:'gdb* process launched by GPS using **ps** and **kill** or the **top** command under Unix

and the Tasks view 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 the **-g** switch is used on the **gnatlink** command (or ;command:*-g* is used in the **gnatmake** command).

The name of the package containing the main program is `b~xxx.ads/adb` where `xxx` is the name of the Ada main unit specified in the **gnatbind** command. Edit and debug this file in the usual manner. You will see a series of calls to the elaboration routines of packages. 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 has not been compiled with debug information, so it needs to be recompiled before you can debug it.

The simplest way is to recompile your application and add the switches **-a** and **-f** to the **gnatmake** command line. This extra step is only required to be done once assuming 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`, which is 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, 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 files, so you cannot have the same filename multiple times in the project hierarchy. This is because GPS needs to know exactly where the file is and cannot reliably guess which occurrence to use.

There are several ways 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. Under these conditions, there is no ambiguity for GPS and the GNAT tools as to 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 to one of the other duplicate files.

Use scenario variables in the project

Here, you define various scenarios in your project (for example compiling in "debug" mode or "production" mode) and change source directories depending on the scenario. Such projects can be edited directly from GPS (in the project properties editor, on the right part of the window, as described in this documentation). On top of the *Project* view (left part of the GPS main window), a combo box is displayed for each variable, allowing you to switch between scenarios depending on what you want to build.

Use extended projects

These projects cannot currently be created through GPS, so you 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 (e.g., 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 :menuselect'System->Preferences->Keyboard' (or launch :program: *gnome-keyboard-properties*).

Select the *Layout* tab and click on *Layout Options*. Then click twice on *Using space key to input non-breakable space character*, select *Usual space at any level*, and then close the dialogs.

File associations or icons disappear or misbehave under Windows

Sometimes file associations get redefined under Windows and no longer behave as a GPS user expects (for example, Ada source files become associated with a stock file icon or double-clicking on a project file opens it like a regular text file.) You may be able to restore the expected behavior by reapplying the associations performed during GPS installation. To do this, locate the file `registry-gps-version.reg` in the root of your GPS installation, and double-click it. Then confirm that you want to apply it in the dialog that appears.

Copy/Paste operations crash GPS running on a forwarded X11 display

It is possible to run GPS on a remote machine using the X11 display forwarding feature of **ssh**. But a copy/paste operation could cause GPS to crash if untrusted forwarding (**ssh -X**) is used. Use the **ssh -Y** option or the ForwardX11Trusted directive in ssh_config to use trusted X11 forwarding and avoid the GPS crash.

Working with Xming

Some old versions of Xming (such as 6.9.0.31) have an issue in that they create "transient" windows larger than the application requests, and do not allow the user to resize these windows. To circumvent this, we have added a command line switch to tell GPS not to store the window sizes and positions: activate this by launching GPS with **--traceoff=STORE_WINDOW_POSITIONS**.

CHAPTER

FIFTEEN

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 can also be used as a reference for the GPS shell.

15.1 Function description

For all functions, the list of parameters is specified. The first parameter is often 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 to always return the same Python instance for a given GUI object. For example, if you were to get another instance of `GPS.EditorBuffer` for the same file as above, you would 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 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 many cases, you need to connect to specific hooks exported by GPS to be aware of events happening in GPS (such as the loading of a file or 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*)
Adds a command to the navigation buttons in the toolbar. When the user presses the *Back* button, this command is executed and puts GPS in a previous state. This is, for example, used while navigating in the HTML browsers to handle their *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*)
Changes the current directory to `dir`.

Parameters **`dir`** – A string

`GPS.`**`compute_xref`**()
Updates 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.**compute_xref_bg**()
Updates cross-reference information stored in GPS in the background.

See also:

GPS.compute_xref()

GPS.**contextual_context**()
Returns the context at the time the contextual menu was open.

This function only returns a valid context while the menu is open or while an action executed from that menu is being executed. You can store your own data in the returned instance so that, for example, 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 the "contextual_menu_open" hook.

Returns An instance of GPS.Context

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%gt;"
      <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 opened.

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**(*refresh=False*)

Returns the current context in GPS. This is the currently selected file, line, column, project, etc. depending on what window is currently active. From one call of this function to the next, a different instance is returned, so you should not store your own data in the instance, since you will not be able to recover it later on

Parameters **refresh** (*boolean*) – If false, the last compute context is returned. The context is set by the views whenever their selection change. You can however set this parameter to true to force a recomputation of the context. This is only useful when your script has executed a number of commands and needs to ensure that the context is properly refresh synchronously.

Returns An instance of GPS.Context

See also:

GPS.Editor.get_line()

GPS.MDI.current:() Access the current window

GPS.contextual_context()

GPS.**delete**(*name*)
Deletes the file or directory name from the file system.

Parameters **name** – A string

GPS.**dir**(*pattern=''*)
Lists files matching 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*)
Dumps string to a temporary file. Return the name of the file. If add_lf is True, appends a line feed at end of the name.

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 the file specified by filename. This is mostly intended for poor shells like the GPS shell which do not have better solutions. In Python, you should use its 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 exec() function instead, it only modifies the current context, which generally has 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*)
Executes 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 waits until the command completes to return 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, etc.

The list of existing actions can be found using the *Edit* → *Actions* menu.

The action is not executed if the current context is not appropriate for it

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*)

Like GPS.execute_action(), but commands that execute external applications or menus are executed asynchronously: this function immediately returns even though external application may 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'*)

Exits 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 means success on most systems.

Parameters

- **force** – A boolean
- **status** – An integer

GPS.**freeze_prefs**()

Prevents the signal "preferences_changed" from being emitted. Call thaw_prefs() to unfreeze.

Freezing/thawing this signal is useful when you are about to modify a large number of preferences in one batch.

See also:

GPS.thaw_prefs()

GPS.**get_build_mode**()

Returns the name of the current build mode. Returns an empty string if no mode is registered.

GPS.**get_build_output**(*target_name*, *shadow*, *background*, *as_string*)

Returns 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_home_dir**()
Returns the directory that contains the user-specific files. This string 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_runtime**()
Returns the runtime currently set in the project or the GPS interface.

Returns a string

GPS.**get_system_dir**()
Returns the installation directory for GPS. This string 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_target**()
Returns the target currently set in the project or the GPS interface.

Returns a string

GPS.**get_tmp_dir**()
Returns the directory where gps creates temporary files. This string always ends with a directory separator.

Returns The install directory for GPS

GPS.**insmod**(*shared_lib*, *module*)
Dynamically registers 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*)
Indicates where the 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**()
Returns the name of the last action executed by GPS. This name is not ultra-precise: it is accurate only when the action is executed through a key binding. Otherwise, an empty string is returned. However, the intent is for a command to be able to check whether it is called multiple times consecutively. For this reason, this function returns the command set by GPS.set_last_command(), if any.

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**()
Returns the list of all known GPS actions, not including menu names. All actions are lower-cased, but the order of 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 example "control-x control-b", returns the list of actions that could be executed. Not all actions would be executed, however, since only the ones for which the filter matches are executed. The names of the actions are always in lower case.

Parameters **key** – A string

Returns A list of strings

See also:

GPS.lookup_actions()

GPS.**ls**(*pattern=''*)
Lists the files matching pattern (all files by default).

Parameters **pattern** – A string

Returns A list of strings

GPS.**lsmod**()
Returns the list of modules 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*)
Loads 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 with 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.**process_all_events**()
Process all the graphical events that have been queue by the system: these events typically involve demands to refresh part of the screen, handle key or mouse events, ... This is mostly useful when writing automatic tests. In plugins, the recommand approach is instead to create actions via gps_utils.interactive(), and run them in the background with GPS.execute_action(). Another possible approach is to use python generators with the yield keyword.

GPS.**pwd**()
Prints name of the current (working) directory.

Returns A string

This function has the same return value as the standard Python function os.getcwd(). The current directory can also be changed through a call to os.chdir("dir").

GPS.**repeat_next**(*count*)
Executes the next action 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 example the text encoding, the programming language, and the debugger breakpoints).

This is done automatically by GPS on exit, so you normally do not have to call this subprogram.

GPS.**send_button_event**(*window=None*, *type=None*, *button=1*, *x=1*, *y=1*, *state=0*)
synthesize and queue an event to simulate a mouse action. This event will be processed later by gtk+ (unless you call gps.process_all_events()). as much as possible, this function should be avoided and you should use gps.execute_action() instead.

Parameters

- **type** (*int*) – the type of event. This defaults to a button press.
- **window** (*GUI*) – the window to which the event should be sent. This defaults to the window that currently has the focus.
- **state** (*int*) – the state of the modified keys (control, shift,...)

GPS.**send_crossing_event**(*window=None*, *type=None*, *x=1*, *y=1*, *state=0*)
synthesize and queue an event to simulate a mouse movement. This event will be processed later by gtk+ (unless you call `gps.process_all_events()`). as much as possible, this function should be avoided and you should use `gps.execute_action()` instead.

Parameters

- **type** (*int*) – the type of event. This defaults to an Enter notify event.
- **window** (*GUI*) – the window to which the event should be sent. This defaults to the window that currently has the focus.
- **state** (*int*) – the state of the modified keys (control, shift,...)

GPS.**send_key_event**(*keyval*, *window=None*, *primary=False*, *alt=False*, *shift=False*, *control=False*)
synthesize and queue an event to simulate a key press. This event will be processed later by gtk+ (unless you call `gps.process_all_events()`). as much as possible, this function should be avoided and you should use `gps.execute_action()` instead.

Parameters **window** (*GUI*) – the window to which the event should be sent. This defaults to the window that currently has the focus.

GPS.**set_build_mode**(*mode=''*)
Sets the current build mode. If `mode` is not a registered mode, does nothing.

Parameters **mode** – Name of the mode to set

GPS.**set_last_command**(*command*)
Overrides the name of the last command executed by GPS. This new name is the one returned by `GPS.last_command()` until the user performs a different action. Thus, multiple consecutive calls of the same action 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**()
Returns the list of languages for which GPS has special handling. Any file can be opened 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 (starts with an upper case character and is lowercase 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 calling the "preferences_changed" hook.

See also:

GPS.freeze_prefs()

GPS.**version**()
Returns the GPS version as a string.

Returns A string

GPS.**xref_db**()
Returns the location of the xref database. This is an **sqlite** database created by GPS when it parses the .ali files generated by the compiler.

Its location depends mainly on the optional IDE'Artifacts_Dir attribute, which defaults to the project's object directory if not specified.

The location can also depend on the optional IDE'Xref_Database attribute which specifies a complete path to the cross-references database file.

Returns a string

15.5 Classes

15.5.1 GPS.Action

class GPS.**Action**
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.

__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, such as /Edit/Copy.

Parameters **name** (*string*) – A string

button(*toolbar='main'*, *section=''*, *group=''*, *label=''*, *icon=''*, *hide=False*)
Add a new button in some toolbars. When this button is clicked, it executes the action from self.

Parameters

- **toolbar** (*string*) – identifies which toolbar the action should be added to. The default is to add to the main toolbar for the main GPS window and all floating windows. Other possible names are the names of the various views, as listed in the /Tools/Views menu.
- **section** (*string*) – identifies which part of the toolbar the button should be added to. By default, the button is added at the end of the toolbar. However, some toolbars define various sections (see the menus.xml file for valid section names).
- **group** (*string*) – when a group is specified, the new button contains a popup menu. A long click on the menu displays a popup with all actions in that group. A short click executes the last action from this group.
- **label** (*string*) – use this as a label for the button, when the user choses to display labels in toolbars. The default is to use the action's name.
- **icon** (*string*) – override the default icon registered for the action.
- **hide** (*bool*) – if the action is disabled or not applicable to the current context, the button will be hidden instead of simply be disabled.

```
# The following adds a 'Copy to clipboard' button in the Messages
# window's local toolbar:
GPS.Action("Copy to Clipboard").button(
    toolbar='Messages', label='Copy')
```

can_execute()
Return True if the action can be executed in the current context.

Return type boolean

contextual(*path*, *ref=''*, *add_before=True*, *group=0*)
Create a new contextual menu associated with the action.

Parameters

- **path** – A string or a function(GPS.Context):string, which describes the path for the contextual menu.
- **ref** (*string*) – A string
- **group** (*int*) – the group of items in the contextual menu. These groups are ordered numerically, so that all items in group 0 appear before items in group 1, and so on.
- **add_before** (*boolean*) – A boolean

create(*on_activate*, *filter=''*, *category='General'*, *description=''*, *icon=''*)
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 functions interactive (see gps_utils.interactive).

Parameters

- **on_activate** (*() -> None*) – A subprogram
- **filter** (*string|(Context) -> boolean*) – A string or subprogram 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. See `GPS.Filter.list()` to retrieve the list of all defined named filters.
- **category** (*str*) – Category of the command in the Key Shortcuts editor.
- **description** (*str*) – Description of the command that appears in the dialog or in tooltips. If you are using Python, a convenient value is on_activate.__doc__, which avoids duplicating the comment.
- **icon** (*str*) – Name of the icon to use for this action (in toolbars, dialogs, ...). This is the name of an icon file in the GPS icons directory.

destroy_ui()
Remove all elements associated with this action (menus, toolbar buttons, contextual menus,...). The action itself is not destroyed

disable(*disabled=True*)
Prevent the execution of the action, whether through menus, contextual menus, key shortcuts,...

Parameters **disabled** (*bool*) – whether to disable or enable

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.

Return type boolean

exists()
Returns a Boolean indicating if an action has already been created for this name.

key(*key*, *exclusive=True*)
Associate a default key binding with the action. This is ignored if the user defined his own key binding. You can experiment with possible values for keys by using the /Edit/Key Shortcuts dialog.

Parameters

- **key** (*string*) – A string
- **exclusive** (*bool*) – if True, the shortcut will no longer be associated with any action is was previously bound to. If False, the shortcut will be associated with multiple action. The first one for which the filter matches is executed when the user presses those keys.

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** (*string*) – A string If path ends with a '-', a separator line is created, instead of a menu item with text.
- **ref** (*string*) – A string
- **add_before** (*boolean*) – A boolean

Returns The instance of GPS.Menu that was created

Return type Menu

15.5.2 GPS.Activities

class GPS.**Activities**
General interface to version control activities systems

__init__(*name*)
Creates a new activity and returns its instance

Parameters **name** (*string*) – Activity name to be given to this instance

```
a=GPS.Activities("Fix loading order")
print a.id()
```

add_file(*file*)
Adds the file to the activity.

Parameters **file** (`GPS.File`) – The file instance to add to the activity

commit()
Commit the activity.

files()
Returns the activity's file list.

Returns A list of files

static from_file (*file*)
Returns the activity containing the given file.

Parameters **file** (GPS.File) – The instance for which you wish to get the corresponding activity

Returns The corresponding activity

Return type GPS.Activities

static get (*id*)
Returns the activity given its id.

Parameters **id** (*string*) – The unique activity's id

Returns The corresponding instance

Return type GPS.Activities

See also:

GPS.Activities.list()

group_commit ()
Returns true if the activity will be committed atomically.

Return type boolean

has_log ()
Returns true if the activity has a log present.

Return type boolean

id ()
Returns the activity's unique id.

Return type string

is_closed ()
Returns true if the activity is closed.

Return type boolean

static list ()
Returns the list of all activity's id.

Returns A list of all activity's id defined

Return type [string]

log ()
Returns the activity's log content.

Return type string

log_file ()
Returns the activity's log file.

Return type GPS.File

name ()
Returns the activity's name.

Return type string

remove_file (*file*)
Removes the file from the activity.

Parameters **file** (GPS.File) – The file to remove

set_closed(*status*)
Set the activity's status to closed.

Parameters **status** (*bool*) – A boolean

toggle_group_commit()
Changes the activity's group commit status.

vcs()
Returns the activity's VCS name.

Return type string

15.5.3 GPS.Alias

class GPS.**Alias**
This class represents a GPS Alias, a code template to be expanded in an editor. This class allows you to manipulate them programmatically.

static get(*name*)
Gets the alias instance corresponding to name.

15.5.4 GPS.Bookmark

class GPS.**Bookmark**
This class provides access to GPS's bookmarks. 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 locate special boxes in a graphical browser, for example.

__init__()
This function prevents the creation of a bookmark instance directly. You must use GPS.Bookmark.get() instead, which always returns 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 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** (*string*) – The name of the bookmark

Return type 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*)
Retrieves a bookmark by its name. If no such bookmark exists, an exception is raised. The same instance of :class:GPS.Bookmark is always returned for a given bookmark, so 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 may want to save all your data when GPS exits

Parameters **name** (*string*) – The name of the bookmark

Return type 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()
Changes the current context in GPS so 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.

Return type [:class'GPS.Bookmark']

```
# 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.

Return type string

rename (*name*)
Rename an existing bookmark. This updates the bookmarks view automatically, and emits the hooks bookmark_removed and bookmark_added.

Parameters **name** (*string*) – The new name of the bookmark

15.5.5 GPS.BuildTarget

class GPS.**BuildTarget**
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*)

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** (*string*) – The name of the new target

- **new_category** (*string*) – The category in which to place the new target

execute(*main_name=''*, *file=None*, *force=False*, *extra_args=''*, *build_mode=''*, *synchronous=True*, *directory=''*, *quiet=False*, *on_exit=None*)

get_command_line()
Returns a string list containing the current arguments of this BuildTarget.

Note that these arguments are not expanded.

hide()
Hide target from menus and toolbar.

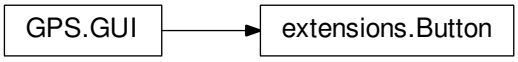
remove()
Remove target from the list of known targets. Any graphical element corresponding to this target is also removed.

show()
Show target in menus and toolbar where it was before hiding.

15.5.6 GPS.Button

class GPS.**Button**
Represents a Button that can be placed in the Toolbar.

This class is provided for backwards compatibility only. Instead of using this, use GPS.Action and GPS.Action.button().



__init__(*id*, *label*, *on_click*)
This class is provided for backwards compatibility only.

Instead of using this, use GPS.Action and GPS.Action.button().

set_text(*label*)
Set the text of the button

15.5.7 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.

Return type [string]

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 do not exist in the buffer.

Parameters

- **text** (*string*) – The content you want to put int the clipboard.
- **append** (*boolean*) – Wether you want to append to the current clipboard content or not.

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 is pasted.

Return type 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** (*integer*) – A null or positive integer
- **index2** (*integer*) – A null or positive integer

15.5.8 GPS.CodeAnalysis

class GPS.**CodeAnalysis**
This class is a toolset that allows handling CodeAnalysis instances.

__init__ ()
Raises an exception to prevent users from creating new instances.

add_all_gcov_project_info ()
Adds coverage information for 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 parsing a .gcov file. The data is read from the cov parameter that should have been created from the specified src file.

Parameters

- **src** (GPS.File) – The corresponding source file
- **cov** (GPS.File) – The corresponding coverage file

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 GNAT project file (`.gpr`) for `prj`.

Parameters **prj** (A `GPS.File` instance) – The corresponding project file

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 containing a representation of the given code analysis.

Parameters **xml** (`GPS.File`) – The output xml file

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** (*string*) – 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.

Return type `GPS.CodeAnalysis`

```
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** (`GPS.File`) – The source xml file

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.

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()

15.5.9 GPS.Codefix

class GPS.**Codefix**
This class gives access to GPS's features for automatically fixing compilation errors.

See also:

GPS.CodefixError()

GPS.Codefix.__init__()

__init__(*category*)
Returns the instance of codefix associated with the given category.

Parameters **category** (*string*) – The corresponding category

error_at(*file*, *line*, *column*, *message=''*)
Returns 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** (GPS.File) – The file where the error is
- **line** (*integer*) – The line where the error is
- **column** (*integer*) – The column where the error is
- **message** (*string*) – The message of the error

Return type GPS.CodefixError

errors()
Lists the fixable errors in the session.

Return type list[GPS.CodefixError]

static parse(*category*, *output*, *regexp=''*, *file_index=-1*, *line_index=-1*, *column_index=-1*, *style_index=-1*, *warning_index=-1*)
Parses 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** (*string*) – A string
- **output** (*string*) – A string
- **regexp** (*string*) – A string
- **file_index** (*integer*) – An integer
- **line_index** (*integer*) – An integer
- **column_index** (*integer*) – An integer
- **style_index** (*integer*) – An integer
- **warning_index** (*integer*) – An integer

See also:

`GPS.Editor.register_highlighting()`

static sessions()

Lists all the existing `Codefix` sessions. The returned values can all be used to create a new instance of `Codefix` through its constructor.

Return type [string]

```
# After a compilation failure:
>>> GPS.Codefix.sessions()
=> ['Builder results']
```

15.5.10 GPS.CodefixError

class GPS.**CodefixError**

This class represents a fixable error in the compilation output.

See also:

`GPS.Codefix()`

`GPS.CodefixError.__init__()`

__init__(*codefix*, *file*, *message=''*)

Describes a new fixable error. If the message is not specified, the first error at that location is returned.

Parameters

- **codefix** (`GPS.Codefix`) – The owning codefix instance
- **file** (`GPS.FileLocation`) – The location of the error
- **message** (*string*) – The message of the error

fix(*choice=0*)

Fixes 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** (*integer*) – 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()
Returns the location of the error.

Return type GPS.FileLocation

message()
Returns the error message, as issue by the tool.

Return type string

possible_fixes()
Lists the possible fixes for the specific error.

Return type [string]

```
for err in GPS.Codefix ("Builder results").errors():
    print err.possible_fixes()
```

15.5.11 GPS.Command

class GPS.**Command**
Interface to GPS command. This class is abstract, and can be subclassed.

static get(*name*)
Returns the list of commands of the name given in the parameter, scheduled or running in the tasks view

Parameters **name** (*string*) – A string

Return type list[GPS.Command]

get_result()
Returns the result of the command, if any. Must be overriden by children.

interrupt()
Interrupts the current command.

static list()
Returns the list of commands scheduled or running in the tasks view.

Return type [GPS.Command]

name()
Return The name of the command

progress()
Returns a list representing the current progress of the command. If current = total, the command has completed.

Returns A list [current, total]

Return type [int]

15.5.12 GPS.CommandWindow

class GPS.**CommandWindow**

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. It can be used to interactively query a parameter for an action, for example.

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*)

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.

Parameters

- **prompt** (*string*) – the short string displayed just before the command line itself. Its goal is to indicate to the user what he is entering.
- **global_window** (*bool*) – 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.
- **on_changed** ((*string, int*) -> *None*) – A subprogram, is called when the user has entered 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** (*(string) -> None*) – A subprogram, is called when the user 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 passed a single parameter, the final input string.
- **on_cancel** (*(string) -> None*) – A subprogram, is called when the user pressed a key that closed the dialog, for example `Esc`. It is passed 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** (*(string, int) -> None*) – 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 passed three parameters, the current input string, the key that was pressed, and the current cursor position.
- **close_on_activate** (*bool*) – A boolean, determines wether the command window has to be closed on pressing enter.

read()
Returns the current contents of the command window.

Return type string

set_background(*color=''*)
Changes the background color of the command window. This can be used to make the command window more obvious or to highlight errors by changing the color. If the color parameter is not specified, the color reverts to its default.

Parameters **color** (*string*) – The new background color

set_prompt(*prompt*)
Changes the prompt displayed before the text field.

Parameters **prompt** (*string*) – The new prompt to display

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 also results 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.

Parameters

- **text** (*string*) – A string
- **cursor** (*integer*) – An integer

15.5.13 GPS.Completion

class `GPS.`**Completion**
This class is used to handle editor completion. See the documentation in the `completion.py` plugin.

static register(*resolver*)
Registers a resolver, which inherits from `CompletionResolver`.

15.5.14 GPS.Console

class GPS.**Console**

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:

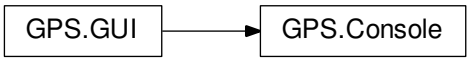
GPS.Process

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, command):
      GPS.Console.__init__(
         command[0],
         on_input=Console_Process.on_input,
         on_destroy=Console_Process.on_destroy,
         force=True)
      GPS.Process.__init__(
```

```
          self, command, ".+",
          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=''*, *manage_prompt=True*, *ansi=False*, *toolbar_name=''*, *give_focus_on_create=True*)

Creates a new instance of GPS.Console. GPS tries to reuse any existing console with the same name. If none exists yet, or the parameter force is set to True, GPS creates a new console.

You cannot create the Python and Shell consoles through this call. If you try, 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, while on_destroy() is 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 function 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 GPS.Console
  - $2: The command to execute

  See the function 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 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 GPS.Console

  The default implementation inserts a tab character, but you can to add additional user input through GPS.Console.add_input() for example.
- The subprogram on_resize() is called whenever the console is resized by the user. It is passed three parameters:
  - $1: the instance of GPS.Console
  - $2: the number of visible rows in the console,

– $3: 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 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 `Ctrl-c` in the console. It receives a single parameter, the instance of `GPS.Console`. By default a `Ctrl-c` is handled by GPS itself by killing 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 usually a good thing and provides advanced functionalities to some programs that lack them. However, there are cases where this gets in the way. For example, 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 prompts. 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 function 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 occasionaly 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 example return, tab, and 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 bitmask, where shift is 1, lock is 2, control is 4 and mod1 is 8.

- :`toolbar_name` is used to register a toolbar for the console. The given name can be used later to register toolbar items (e.g: using the *GPS.Action.button* function).

- :`give_focus_on_create` is only used if a new console is being created. It should be set to True if the newly created console should receive the focus. If it's set to False, the console will not receive the focus: its tab label will be highlighted instead.

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
- **toolbar_name** – A string
- **give_focus_on_create** – A boolean

accept_input()
Returns True if the console accepts input, False otherwise.

Returns A boolean

add_input(*text*)
Adds extra text to the console as if the user had typed it. As opposed to text inserted using `GPS.Console.write()`, this text remains editable by the user.

Parameters **text** – A string

clear()
Clears the current contents of the console.

clear_input()
Removes any user input that the user has started typing (i.e., since the last output inserted through `GPS.Console.write()`).

copy_clipboard()
Copies the selection to the clipboard.

create_link(*regexp*, *on_click*, *foreground='blue'*, *background=''*, *underline=True*)
Registers a regular expression that should be highlighted in this console to provide hyperlinks, which 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()` is highlighted in blue and underlined, just like a hyperlink in a web browser. If the user clicks on that text, `on_click()` is called with one parameter, the text that was clicked on. This can, for example, be used to jump to an editor or 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 you can test for the presence of text before or after the actual hyper link).

Parameters foreground and background specify colors to visualize matched text, while underline turns underscore on.

Parameters

- **regexp** – A string
- **on_click** – A subprogram
- **foreground** – A string

- **background** – A string
- **underline** – A boolean

See also:

GPS.Console.write_with_links()

delete_links()
Drops each regular expression registered with create_link().

enable_input(*enable*)
Makes the console accept or reject input according to the value of "enable".

Parameters **enable** – A boolean

flush()
Does nothing, needed for compatibility with Python's file class.

get_text()
Returns the content of the console.

Returns A string

insert_link(*text*, *on_click*)
Inserts the given text in the console as an hyperlink, using the default hyperlink style. If the user clicks on that text, on_click() is called with one parameter, the text that was clicked on. This can, for example, be used to jump to an editor or open a web browser.

Parameters

- **text** – A string
- **on_click** – A subprogram

isatty()
Returns True if the console behaves like a terminal. Mostly needed for compatibility with Python's file class.

Returns A boolean

read()
Reads the available input in the console. Currently, this behaves exactly like readline().

Returns A String

readline()
Asks 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()
Selects the complete contents of the console.

write(*text*, *mode="'text"'*)
Outputs some text on the console. This text is read-only. If the user started typing some text, that text is temporarily removed, the next text is inserted (read-only), and the user text is put back.

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*)

Outputs some text on the console, highlighting the parts of it that match 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(
      buffer.at(int(matched.group(2)), 1))
```

15.5.15 GPS.Construct

class GPS.**Construct**

One node of the semantic tree when parsing a file for a given programming language. Instances of such classes are only created by GPS internally

file = ''

The GPS.File in which the construct occurs

id = ''

Unique id for the entity

name = ''

The name of the construct

start = (0, 0, 0)

The source location for the beginning of this construct, (line, column offset)

__init__()

Instances are only created by GPS itself

15.5.16 GPS.ConstructsList

class GPS.**ConstructsList**

This class is closely associated with the GPS.Language class, and is used by plug-ins to describe the semantic organization in a source file.

This can be used in particular to populate the Outline view for custom languages (see the python_support.py plugin in the GPS sources).

add_construct(*category*, *is_declaration*, *visiblity*, *name*, *profile*, *sloc_start*, *sloc_end*, *sloc_entity*, *id=''*)

Register a new semantic construct from the file.

Parameters

- **category** (*int*) – the name of the category. It should be one of the CAT_* constants in the `constructs.py` module. If your language has different constructs, you should map them to one of the existing categories.
- **is_declaration** (*bool*) – whether this is the declaration for the entity, or a reference to it.
- **visibility** (*int*) – whether the entity is public, protected or private. It should be one of the constants in the `constructs.py` module.
- **name** (*str*) – the name of the entity
- **profile** (*str*) – a description of its profile (the list of parameters for a subprogram, for instance).
- **sloc_start** ((*int,int,int)*) – the position at which this constructs starts. This is a tuple (line, column, offset), where offset is the number of bytes since the start of the file.
- **sloc_end** ((*int,int,int)*) – the position at which this constructs ends. This is a tuple (line, column, offset).
- **sloc_entity** ((*int,int,int)*) – the position at which the entity name starts. This is a tuple (line, column, offset).
- **id** (*str*) – a unique identifier for this identity. You can retrieve it in calls to `GPS.Language.clicked_on_construct()`, and this is used to identify overloading identifiers in the Outline view when it is refreshed.

15.5.17 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.

__init__()

x.__init__(...) initializes x; see help(type(x)) for signature

directory()

Return the current directory of the context.

Return type string

end_line()

Return the last selected line in the context.

Return type integer

entity(*approximate_search_fallback=True*)

Returns the entity stored in the context. This might be expensive to compute, so it is often recommend to check whether *GPS.Context.entity_name* returns None, first.

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

entity_name()
Return the name of the entity that the context points to. This is None when the context does not contain entity information.

Return type str

file()
Return the name of the file in the context. This method used to be set only for a GPS.Context.

Return type GPS.File

See also:

GPS.Context.files

files()
Return the list of selected files in the context

Return type list[GPS.File]

location()
Return the file location stored in the context.

Return type GPS.FileLocation

message()
Returns the current message that was clicked on

Returntype GPS.Message

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 found from the context.

Return type GPS.Project

set_file(*file*)
Set the file stored in the context. :param GPS.File file:

start_line()
Return the first selected line in the context.

Return type integer

15.5.18 GPS.Contextual

class GPS.**Contextual**
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__()

name = ''
The name of the contextual menu (see __init__)

__init__(*name*)

Initializes a new instance of GPS.Contextual. The name is what 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 example 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*, *filter=None*, *ref=''*, *add_before=True*, *group='0'*, *visibility_filter=None*, *action=None*)

OBSOLETE: please use GPS.Action.contextual() instead. All contextual menus should be associated with a specific action, so that this action can also be associated with a key shortcut, or reused in toolbars,...

Creates a new contextual menu entry. Whenever this menu entry is selected by the user, GPS executes on_activate(), passing one parameter which is the context for which the menu is displayed (this is usually the same as GPS.current_contextual()).

If on_activate is None, a separator is created.

The filter parameter can be used to filter when the entry should be displayed in the menu. It is a function that receives one parameter, an instance of GPS.Context, and returns a boolean. If it returns True, the entry is 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 a parameter and returns a string, which is 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

- **self** – An instance of GPS.Contextual
- **on_activate** – A subprogram with one parameter context
- **label** – A subprogram
- **ref** – A string
- **add_before** – A boolean
- **filter** – A subprogram
- **group** – An integer
- **action** (*GPS.Action*) – An action instance to be executed on menu activation

```
## 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 context.entity_name() is not None

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)

.. code-block:: python

## 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 context.entity_name() is not None

   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.on_label)
```

create_dynamic (*factory*, *on_activate*, *label=''*, *filter=None*, *ref=''*, *add_before=True*, *group='0'*)

Creates 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(contextl):
   return context.entity_name() is not None

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 context.entity_name() is not None

    def on_activate(self, context, choice, choice_index):
       GPS.Console("Messages").write("You selected " + choice)

    def factory(self, context):
       return ["Choice1", "Choice2"]
```

hide()
Makes sure the contextual menu never appears when the user right clicks anywhere in GPS. This is the standard way to disable contextual menus.

See also:

GPS.Contextual.show()

static list()
Returns 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*)
Controls whether the contextual menu is grayed-out: False if it should be grayed-out, True otherwise.

Parameters **Sensitivity** – Boolean value

show()
Makes sure the contextual menu is shown when appropriate. The entry might still be invisible if you right clicked on a context where it does not apply, but it will be checked.

See also:

GPS.Contextual.hide()

15.5.19 GPS.Cursor

class GPS.**Cursor**
Interface to a cursor in GPS.EditorBuffer. Only gives access to the insertion mark and to the selection mark of the cursor.

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

location()
Returns the cursor's location :rtype: GPS.EditorLocation

mark()
Returns the cursor's main mark.

NOTE: If you can interact with your cursor via Cursor.move() rather than via manually moving marks, you should prefer that method.

Returns The GPS.EditorMark instance corresponding to the cursor's insert mark

move(*loc*, *extend_selection=False*)
Moves the cursor to the given location.

Parameters

- **loc** – A GPS.EditorLocation that you want to move the cursor to
- **extend_selection** – A boolean. If True, the selection mark will not move so the selection is extended. If False, both marks move simultaneously

sel_mark()
Returns the cursor's selection mark.

NOTE: If you can interact with your cursor via `Cursor.move()` rather than via manually moving marks, you should prefer that method.

Returns The `GPS.EditorMark` instance corresponding to the cursor's selection mark

set_manual_sync()
Sets the buffer in manual sync mode regarding this cursor. This set sync to be manual and all insertions/deletions are considered as originating from this cursor instance. If you do not do this, an action on the buffer (like an insertion) is repercuted on every alive cursor instance.

NOTE: Do not call this manually. Instead, iterate on the results of `EditorBuffer.cursors()` so this method is called for you automatically.

15.5.20 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.

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)
```

breakpoints = []
A read-only property that returns the list of breakpoints currently set in the debugger. This information is updated automatically by GPS whenever a command that might modify this list of breakpoints is executed. The elements in this list are instances of `GPS.DebuggerBreakpoint`

current_file = None
A `GPS.File` which indicates the current file for the debugger. This is the place where the debugger stopped, or when the user selected a new frame, the file corresponding to that frame.

current_line = 0
The current line. See description of *GPS.Debugger.current_file*.

remote_protocol = None
A string set to the debugger's currently used remote protocol. This remote target is either retrieved from

the IDE'Communication_Protocol project atttribute or from a manually sent 'target [remote_protocol] [remote_target]' command.

remote_target = None
A string set to the currently used debugger's remote target. This remote target is either retrieved from the IDE'Protocol_Host project atttribute or from a manually sent 'target [remote_protocol] [remote_target]' command.

__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()`

break_at_location(*file*, *line*)
Set a breakpoint at a specific location. If no debugger is currently running, this commands ensures that a breakpoint will be set when a debugger is actually started.

Equivalent gdb command is "break".

Parameters

- **file** (*GPS.File*) – the file to break into
- **line** (*int*) – the line to break at

close()
Closes the given debugger. This also closes all associated windows (such as the call stack and console).

command()
Returns the command 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

current_frame()
Returns the number of current frame.

Returns integer, the number of frame

frame_down()
Select previous frame.

frame_up()
Select next frame.

frames()
Returns list of frames.

Returns A list of frames subprograms

static get(*id=None*)
Gives access to an already running debugger, and returns 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_console()
Returns the GPS.Console instance associated with the the given debugger's console.

Returns An instance of GPS.Console

get_executable()
Returns the name of the executable currently debugged in the 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 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()
Returns true if the command returned by GPS.Debugger.command() is likely to modify the list of breakpoints after it finishes 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()
Returns true if the command returned by GPS.Debugger.command() is likely to modify the current context (e.g., current task or thread) after it finishes executing.

Returns A boolean

is_exec_command()
Returns true if the command returned by GPS.Debugger.command() is likely to modify the stack trace in the debugger (e.g., "next" or "cont").

Returns A boolean

static **list**()
Returns the list of currently running debuggers.

Returns A list of GPS.Debugger instances

non_blocking_send(*cmd*, *output=True*)
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()

select_frame(*num*)
Select frame by number.

Parameters **num** – The number of frame where 0 is the first frame

send(*cmd*, *output=True*, *show_in_console=False*)
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.

There exists a number of functions that execute specific commands and parse the output appropriately. It is better to use this functions directly, since they will change the actual command emitted depending on which debugger is running, whether it is currently in a C or Ada frame,...

Parameters

- **cmd** – A string
- **output** – A boolean
- **show_in_console** – A boolean

Returns A string

See also:

`GPS.Debugger.non_blocking_send()`

See also:

`GPS.Debugger.value_of()`

See also:

`GPS.Debugger.set_variable()`

See also:

`GPS.Debugger.break_at_location()`

See also:

`GPS.Debugger.unbreak_at_location()`

set_variable(*variable*, *value*)
Set the value of a specific variable in the current context.

Equivalent gdb command is "set variable".

Parameters

- **variable** (*str*) – the name of the variable to set.
- **value** (*str*) – the value to set it to, as a string

static spawn(*executable*, *args=''*, *remote_target=''*, *remote_protocol=''*, *load_executable=False*)
Starts a new debugger. It will debug `executable`. When the program is executed, the extra arguments args are passed.

If `remote_target` and `remote_protocol` are specified and non-empty, the debugger will try to initialize a remote debugging session with these parameters. When not specified, the `IDE'Program_Host` and `IDE'Communication_Protocol` are used if present in the .gpr project file.

When `load_executable` is True, GPS will try to load `executable` on the specified remote target, if any.

Parameters

- **executable** – An instance of GPS.File
- **args** – A string
- **remote_target** – A string
- **remote_protocol** – A string
- **load_executable** – A boolean

Returns An instance of `GPS.Debugger`

unbreak_at_location (*file*, *line*)

Remove any breakpoint set at a specific location.

Equivalent gdb command is "clear". If no debugger is currently running, this commands ensures that no breakpoint will be set at that location when a debugger is actually started.

Parameters

- **file** (*GPS.File*) – the file to break into
- **line** (*int*) – the line to break at

value_of (*expression*)

Compute the value of expression in the current context.

Equivalent gdb command is "print".

Parameters **expression** (*str*) – the expression to evaluate.

Returns a string, or "" if the expression could not be evaluated in the current context.

15.5.21 GPS.DebuggerBreakpoint

class `GPS.`**DebuggerBreakpoint**

Instances of this class represents one breakpoint set in the debugger.

enabled = True

Whether this breakpoint is enabled

file = None

An instance of GPS.File, where the debugger will stop.

line = 0

The line on which the debugger will stop

num = 0

The identifier for this breakpoint

type = ''

Either 'breakpoint' or 'watchpoint'

watched = ''

If the breakpoint is a watchpoint, i.e. monitors changes to a variable, this property gives the name of the variable.

__init__()

x.__init__(...) initializes x; see help(type(x)) for signature

15.5.22 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 it.

Parameters

- **file** – A string
- **start_line** – An integer
- **number_of_lines** – An integer
- **category** – A string

Returns an instance of GPS.EditorMark

static add_case_exception (*name*)
OBSOLESCENT.

Adds name into the case exception dictionary.

Parameters **name** – A string

static block_fold (*file*, *line=None*)
OBSOLESCENT.

Folds 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*)
OBSOLESCENT.

Returns nested level for block enclosing line.

Parameters

- **file** – A string
- **line** – An integer

Returns An integer

static block_get_name (*file*, *line*)
OBSOLESCENT.

Returns name for block enclosing line

Parameters

- **file** – A string
- **line** – An integer

Returns A string

static block_get_start (*file*, *line*)
OBSOLESCENT.

Returns ending line number for block enclosing line.

Parameters

- **file** – A string
- **line** – An integer

Returns An integer

static block_get_type (*file*, *line*)
OBSOLESCENT.

Returns type for block enclosing line.

Parameters

- **file** – A string
- **line** – An integer

Returns A string

static block_unfold (*file*, *line=None*)
OBSOLESCENT.

Unfolds 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.

Closes all file editors for file.

Parameters **file** – A string

static copy ()
OBSOLESCENT.

Copies the selection in the current editor.

static create_mark (*filename*, *line=1*, *column=1*, *length=0*)
Creates 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 An instance of GPS.EditorMark

See also:

GPS.Editor.goto_mark()

GPS.Editor.delete_mark()

static cursor_center (*file*)
OBSOLESCENT.

Scrolls 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.

Sets cursor to position line/column in buffer file.

Parameters

- **file** – A string
- **line** – An integer
- **column** – An integer

static cut ()
OBSOLESCENT.

Cuts the selection in the current editor.

static edit (*filename*, *line=1*, *column=1*, *length=0*, *force=False*, *position=5*)
OBSOLESCENT.

Opens 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.

Gets 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_last_line (*file*)
OBSOLESCENT.

Returns the number of the last line in file.

Parameters **file** – A string

Returns An integer

static goto_mark (*mark*)
OBSOLESCENT.

Jumps to the location of the mark corresponding to identifier.

Parameters **mark** – A instance of GPS.EditorMark

See also:

GPS.Editor.create_mark()

static highlight (*file*, *category*, *line=0*)
OBSOLESCENT

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*)
OBSOLESCENT>

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*)
OBSOLESCENT.

Indents the selection (or the current line if requested) in current editor. Does nothing if the current GPS window is not an editor.

Parameters **current_line_only** – A boolean

static indent_buffer ()
OBSOLESCENT.

Indents the current editor. Does nothing if the current GPS window is not an editor.

static insert_text (*text*)
OBSOLESCENT.

Inserts a text in the current editor at the cursor position.

Parameters **text** – A string

static mark_current_location ()
OBSOLESCENT.

Pushes 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 ()
OBSOLESCENT.

Pastes the selection in the current editor.

static print_line_info (*file*, *line*)
OBSOLESCENT.

Prints 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.

Redoes the last undone editing command for file.

Parameters **file** – A string

static refill ()
OBSOLESCENT.

Refills selected (or current) editor lines. Does nothing if the current GPS window is not an editor.

static register_highlighting (*category*, *color*, *speedbar=False*)
OBSOLESCENT.

Creates 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

Removes blank lines located at mark. If number is specified, remove only the number first lines.

Parameters

- **mark** – an instance of GPS.EditorMark
- **number** – An integer

static remove_case_exception (*name*)
OBSOLESCENT.

Removes name from the case exception dictionary.

Parameters **name** – A string

static replace_text (*file*, *line*, *column*, *text*, *before=-1*, *after=-1*)
OBSOLESCENT.

Replaces 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*)
OBSOLESCENT.

Saves 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*)
OBSOLESCENT.

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 ()
OBSOLESCENT.

Selects the whole editor contents.

static select_text (*first_line*, *last_line*, *start_column=1*, *end_column=0*)
OBSOLESCENT.

Selects 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*)
OBSOLESCENT.

Sets the background color for the editors for file.

Parameters

- **file** – A string
- **color** – A string

static set_synchronized_scrolling (*file1*, *file2*, *file3=''*)
OBSOLESCENT.

Synchronizes the scrolling between multiple editors.

Parameters

- **file1** – A string
- **file2** – A string
- **file3** – A string

static set_title (*file*, *title*, *filename*)
OBSOLESCENT.

Changes 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.

Changes 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.

Undoes the last editing 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.

Removes 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

15.5.23 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.

extend_existing_selection = False
When set to True, this flag puts the editor in a special mode where all cursor moves will create and extend the selection. This is used to emulate the behavior of some editors, like Emacs, or vi's "v" mode".

The default behavior is that cursor moves will cancel any existing selection, unless they are associated with the shift key. In this case, a new selection is created if none exists, and the selection is extended to include the new cursor location.

__init__()
Prevents the direct creation of instances of EditorBuffer. Use GPS.EditorBuffer.get() instead

add_cursor(*location*)
Adds a new slave cursor at the given location.

Return type The resulting Cursor instance

add_special_line(*start_line*, *text*, *category=''*, *name=''*)
Adds one non-editable line to the buffer, starting at line start_line and containing the string text. If category is specified, use it for highlighting. Creates a mark at beginning of block and return it. If name is specified, the returned mark has this name.

Parameters

- **start_line** (*int*) – An integer
- **text** (*string*) – A string
- **category** (*string*) – A string Reference one of the categories that were registered via GPS.Editor.register_highlighting(). This can also be the name of a style defined via GPS.Style
- **name** (*string*) – A string

Return type EditorMark

See also:

GPS.EditorBuffer.get_mark()

apply_overlay (*overlay*, *frm='begining 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** (*EditorOverlay*) – An instance of GPS .EditorOverlay
- **frm** (*EditorLocation*) – An instance of GPS.EditorLocation
- **to** (*EditorLocation*) – An instance of GPS.EditorLocation

See also:

GPS.EditorBuffer.remove_overlay()

at (*line*, *column*)
Returns a new location at the given line and column in the buffer.

Return type EditorLocation

beginning_of_buffer()
Returns a location pointing to the first character in the buffer.

Return type EditorLocation

blocks_fold()
Folds all the blocks in all the views of the buffer. Block folding is a language-dependent feature, where 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 it can be unfolded later.

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.

Return type 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** (*bool*) – A boolean

copy (*frm='beginning of buffer'*, *to='end of buffer'*, *append=False*)
Copies 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.

:param EditorLocation frm : An instance of EditorLocation :param EditorLocation to: An instance of EditorLocation :param bool append: A boolean

See also:

GPS.Clipboard.copy()

create_overlay(*name=''*)
Creates 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 rqendering 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 overlays exit. 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 example.

Parameters **name** (*string*) – A string

Return type 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.

Return type EditorView

cursors(*ed*)

cut(*frm='beginning of buffer'*, *to='end of buffer'*, *append=False*)
Copies 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** (*EditorLocation*) – An instance of EditorLocation
- **to** (*EditorLocation*) – An instance of EditorLocation
- **append** (*bool*) – A boolean

delete(*frm='beginning of buffer'*, *to='end of buffer'*)
Deletes the given range of text from the buffer.

Parameters

- **frm** (*EditorLocation*) – An instance of EditorLocation
- **to** (*EditorLocation*) – An instance of EditorLocation

end_of_buffer()
Returns a location pointing to the last character in the buffer.

Return type EditorLocation

entity_under_cursor()
Shortcut to return a GPS.Entity instance corresponding to the entity under cursor

Return type GPS.Entity

expand_alias(*alias*)
Expands given alias in the editor buffer at the point where the cursor is.

file()
Returns the name of the file edited in this buffer.

Return type File

finish_undo_group()
This is deprecated, use GPS.EditorBuffer.new_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 example, 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 the file is not specified, the current editor is returned, which is the last one that had the keyboard focus.

If the file is not currently open, the behavior depends on `open`:; 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 true, a reload is forced in case the file is already open.

Parameters

- **file** (*File*) – An instance of `GPS.File`
- **force** (*bool*) – A boolean
- **open** (*bool*) – A boolean

Return type `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_analysis_unit()
Returns the corresponding libadalang AnalysisUnit.

Return type libadalang.AnalysisUnit

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** (*EditorLocation*) – An instance of `EditorLocation`
- **to** (*EditorLocation*) – An instance of `EditorLocation`

Return type string

get_cursors()
Returns a list of `Cursor` instances. Note that if you intend to perform actions with tgnores __hem (in particular deletions/insertions), you should call set_manual_sync, on the cursor's instance. Also, if you move any selection mark, you should call update_cursors_selection afterwards.

There is a higher level method, `EditorBuffer.cursors()` that returns a generator that will handle this manual work for you.

Return type list[Cursor]

get_lang()
Return the name of the programming language used for this editor, in particular for syntax highlighting and auto indentation.

Returns a GPS.LanguageInfo instance.

See also:

GPS.EditorBuffer.set_lang()

get_mark(*name*)
Checks 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** (*string*) – A string

Return type GPS.EditorMark

See also:

GPS.EditorLocation.create_mark()

```
ed = GPS.EditorBuffer.get(GPS.File("a.adb"))
loc = ed.at(4, 5)
mark = loc.create_mark("name")
mark.data = "whatever"

# .. anything else

mark = ed.get_mark("name")
# mark.data is still "whatever"
```

static **get_new**()
Opens a new editor on a blank file. This file has no name, and you will have to provide one when you save it.

Return type EditorBuffer

has_slave_cursors()
Returns true if there are any alive slave cursors in the buffer currently.

Return type bool

indent(*frm='beginning of buffer'*, *to='end of buffer'*)
Recomputes the indentation of the given range of text. This feature is language-dependent.

Parameters

- **frm** (*EditorLocation*) – An instance of EditorLocation
- **to** (*EditorLocation*) – An instance of EditorLocation

insert(*loc_or_text*, *text=None*)
Inserts some text in the buffer.

Parameters

- **loc_or_text** (*string|EditorLocation*) – Either where to insert the text, or the text to insert in the buffer
- **text** (*string|None*) – If the first passed parameter was a location, this is the text to be inserted. Else, it can be ignored.

See also:

GPS.EditorBuffer.delete()

is_modified()
Tests whether the buffer has been modified since it was last opened or saved.

Return type bool

is_read_only()
Whether the buffer is editable or not.

Return type bool

See also:

GPS.EditorBuffer.set_read_only()

lines_count()
Returns the total number of lines in the buffer.

Return type int

static **list**()
Returns the list of all editors that are currently open in GPS.

Returns A list of instances of GPS.EditorBuffer

Return type [EditorBuffer]

```
# It is possible to close all editors at once using a command like

for ed in GPS.EditorBuffer.list():
    ed.close()
```

main_cursor()
Returns the main cursor. Generally you should not use this method except if you have a really good reason to perform actions only on the main cursor. Instead, you should iterate on the result of EditorBuffer.cursors().

Returns A Cursor instance

Return type Cursor

new_undo_group()
Create a new undo group.

This returns an object which should be used as a context manager. If you would like N actions to be considered as atomic for undo/redo, use this:

with buffer.new_undo_group(): action 1 ... action N

paste(*location*)
Pastes the contents of the clipboard at the given location in the buffer.

Parameters **location** (*EditorLocation*) – An instance of EditorLocation

redo()
Redoes the last undone command in the editor.

refill(*frm='beginning of buffer'*, *to='end of buffer'*)
Refills the given range of text, i.e., cuts long lines if necessary so that they fit in the limit specified in the GPS preferences.

Parameters

- **frm** (*EditorLocation*) – An instance of EditorLocation
- **to** (*EditorLocation*) – An instance of EditorLocation

remove_all_slave_cursors()
Removes all active slave cursors from the buffer.

remove_overlay(*overlay*, *frm='begining of buffer'*, *to='end of buffer'*)
Removes all instances of the overlay in the given range of text. It is not 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** (*EditorOverlay*) – An instance of EditorOverlay
- **frm** (*EditorLocation*) – An instance of EditorLocation
- **to** (*EditorLocation*) – An instance of EditorLocation

See also:

GPS.EditorBuffer.apply_overlay()

remove_special_lines(*mark*, *lines*)
Removes specified number of special lines at the specified mark. It does not delete the mark.

Parameters

- **mark** (*EditorMark*) – An instance of EditorMark
- **lines** (*int*) – 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** (*bool*) – A boolean
- **file** (*File*) – An instance of 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 is be left on to.

Parameters

- **frm** (*EditorLocation*) – An instance of EditorLocation
- **to** (*EditorLocation*) – An instance of EditorLocation

selection_end()
Returns the character after the end of the selection. This is always located after the start of the selection, no matter what the order of parameters given to GPS.EditorBuffer.select() is. If the selection is empty, EditorBuffer.selection_start() and EditorBuffer.selection_end() will be equal.

Return type 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()
Returns the start of the selection. This is always located before the end of the selection, no matter what the order of parameters passed to GPS.EditorBuffer.select() is.

Return type EditorLocation

set_cursors_auto_sync()
Sets 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 will not be propagated.

set_lang(*lang*)
Set the highlighting programming language. When you open an existing file, GPS automatically computes the best highlighting language based on file extensions and naming schemes defined in your project, or on the language that was set manually via the Properties contextual menu.

This function can be used to override this, or set it for newly created files (GPS.EditorBuffer.get_new())

See also:

GPS.EditorBuffer.get_lang()

set_read_only(*read_only=True*)
Indicates whether the user should be able to edit the buffer interactively (through any view).

Parameters **read_only** (*bool*) – A boolean

See also:

GPS.EditorBuffer.is_read_only()

start_undo_group()
This is deprecated. Use GPS.EditorBuffer.new_undo_group

This is done via a context manager:

with buffer.new_undo_group(): action 1 ... action N

undo()
Undoes the last command in the editor.

unselect()
Cancels the current selection in the buffer.

update_cursors_selection()
Updates the overlay used to show the multi cursor's current selection. This must be called after any operation on multi cursor selection marks

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 EditorView instances

Return type list[EditorView]

15.5.24 GPS.EditorHighlighter

class GPS.**EditorHighlighter**

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.

15.5.25 GPS.EditorLocation

class GPS.**EditorLocation**

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 = ed.at(line=4, column=5)
loc.data = "MY OWN DATA"
loc2 = ed.at(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()

Returns a location at the beginning of the line on which `self` is.

Returns A new instance of GPS.EditorLocation

block_end()

Returns the location of the end of the current block.

Returns An instance of GPS.EditorLocation

block_end_line()

Returns 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()

Folds the block containing the location, i.e., makes it invisible on the screen, except for its first line. Clicking on the icon next to this first line unfolds the block ands make it visible to the user.

See also:

GPS.EditorLocation.block_unfold()

block_level()

Returns 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()

Returns the name of the bock surrounding the location. The definition of a block depends on the specific language of the source file.

Returns A string

block_start()

Returns the location of the beginning of the current block.

Returns An instance of GPS.EditorLocation

block_start_line()
Returns the first line of the block surrounding the location. The definition of a block depends on the programming language.

Returns An integer

block_type()
Returns the type of the block surrounding the location. This type indicates whether the block is, e.g., subprogram, an if statement, etc.

Returns A string

block_unfold()
Unfolds the block containing the location, i.e., makes visible any information that was hidden as a result of running GPS.EditorLocation.block_fold().

See also:

GPS.EditorLocation.block_fold()

buffer()
Returns the buffer in which the location is found.

Returns An instance of GPS.EditorBuffer

column()
Returns the column of the location.

Returns An integer

create_mark(*name=''*, *left_gravity=True*)
Creates a mark at that location in the buffer. The mark will stay permanently at that location, and follows it if the buffer is modified. In fact, even if the buffer is closed and then reopened, the mark will keep track of the location, but of course not if the file is edited outside of GPS.

Parameters

- **name** (*str*) – The name of the mark. If specified, this creates a named mark, which can later 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.
- **left_gravity** (*bool*) – decides whether the mark is moved towards the left or the right when text that contains the mark is deleted, or some text is inserted at that location.

Returns An instance of GPS.EditorMark

See also:

GPS.EditorBuffer.get_mark()

```
buffer = GPS.EditorBuffer.get(GPS.File("a.adb"))
loc = buffer.at(3, 4)
mark = loc.create_mark()
buffer.insert(loc, "text")
loc = mark.location()
# loc.column() is now 8
```

end_of_line()
Returns a location located at the end of the line on which self is.

Returns A new instance of GPS.EditorLocation

ends_word()
Returns true if self is currently at the end of a word. The definition of a word depends on the language used.

Returns A boolean

entity()
Returns a GPS.Entity instance at the given location.

If there is no entity that can be resolved at this location, returns None

Return type GPS.Entity

forward_char(*count*)
Returns 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*)
Returns a new location located count lines after self. The location is moved back to the beginning of the line. If 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 (i.e., 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*)
Returns 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.

Parameters **count** – An integer

Returns A new instance of GPS.EditorLocation

get_char()
Returns 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 in 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()
Returns the list of all 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

get_word()
This will return the word that contains this location, if there is one, the empty string otherwise. This is a shortcut method that uses the inside_word, starts_word and ends_word methods of *GPS.EditorLocation*.

Returns A tuple (word, start location, end location)

Return type (unicode, GPS.EditorLocation, GPS.EditorLocation)

has_overlay(*overlay*)
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()
Returns true if self is currently inside a word. The definition of a word depends on the language.

Returns A boolean

line()
Returns the line number of the location.

Returns An integer

offset()
Returns the offset of the location in the buffer, i.e., 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*)
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, 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()
Returns true if self is currently at the start of a word. The definition of a word depends on the language.

Returns A boolean

subprogram_name()
Returns the name of the subprogram containing the location.

Returns A string

15.5.26 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 example, 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()

column = 0
Read only property that gives the location of the mark. :type: int

file = None
Read only property that gives the location of the mark. :type: GPS.File

line = 0
Read only property that gives the location of the mark. :type: int

__init__()
Always raises 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 have not 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. Calling this function will open the corresponding source editor.

Returns An instance of GPS.EditorLocation

```
ed = GPS.EditorBuffer.get(GPS.File("a.adb"))
loc = ed.at(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

15.5.27 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; 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.highilghter.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*)

Retrieves 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()

Returns 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*)

Changes 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)

Changes the foreground color of the text.

•*background* (value is a string with the color name)

Changes the background color of the text.

•*paragraph-background* (value is a string with the color name)

Changes the background color of 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 text font.

•*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.

•*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()`).

Parameters

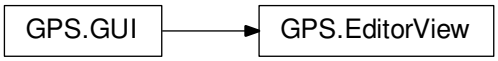
- **`name`** – A string
- **`value`** – A string or a boolean, depending on the property

15.5.28 GPS.EditorView

class GPS.**EditorView**

One view of an editor, i.e., 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*)

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** (*EditorBuffer*) – 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.

Return type EditorBuffer

center(*location='location of cursor'*)

Scrolls the view so that the location is centered.

Parameters **location** (*EditorLocation*) – An instance of GPS .EditorLocation

cursor()

Returns the current location of the cursor in this view.

Return type EditorLocation

get_extend_selection()

See GPS.EditorBuffer.extend_existing_selection()

goto(*location*, *extend_selection=False*)

Moves the cursor to 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** (*EditorLocation*) – An instance of GPS .EditorLocation
- **extend_selection** (*bool*) – A Boolean

is_read_only()

Whether the view is editable or not. This property is shared by all views of the same buffer.

Return type bool

See also:

GPS.EditorBuffer.is_read_only()

set_extend_selection(*extend*)
See GPS.EditorBuffer.extend_existing_selection()

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.

Parameters **read_only** (*bool*) – 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** (*bool*) – A boolean

15.5.29 GPS.Entity

class GPS.**Entity**
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()
Returns various boolean attributes of the entity: is the entity global, static, etc.

Returns A htable with the following keys: - 'global': whether the entity is a global entity - 'static': whether the entity is a local static variable (C/C++) - 'in': for an in parameter for an Ada subprogram - 'out': for an out parameter for an Ada subprogram - 'inout': for an in-out parameter for an Ada subprogram - 'access': for an access parameter for an Ada subprogram

body (*nth='1'*)
Returns 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*)
Displays 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. 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 ()
Opens the call graph browser to show what entities call self.

calls (*dispatching_calls=False*)
Displays 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, calls done through dispatching will result in multiple entities being listed (i.e., 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 ()
Returns the category of a given entity. Possible values include: label, literal, object, subprogram, package, namespace, type, and 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(), etc.

Returns A string

child_types ()
Return the list of entities that extend self (in the object-oriented sense)

Returns a list of GPS.Entity

declaration ()
Returns the location of the declaration for the entity. The file's name is 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()
Returns 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 A list of GPS.Entity

discriminants()
Returns the list of discriminants for entity. This is a list of entities, empty if the type has no discriminant or if this notion does not apply to the language.

Returns A list of instances of GPS.Entity

documentation(*extended=False*)
Returns 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, the returned documentation includes formatting and the full entity description.

Parameters **extended** – A boolean

Returns A string

end_of_scope()
Returns the location at which the end of the entity is found.

Returns An instance of GPS.FileLocation

fields()
Returns the list of fields for the 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 A list of instances of GPS.Entity

find_all_refs(*include_implicit=False*)
Displays in the *Locations* view all the references to the entity. If include_implicit is true, implicit uses of the entity are also referenced, for example 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()
Returns 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

get_called_entities()
Return the list of entities referenced within the scope of self.

Returns a list of GPS.Entity

instance_of()
If self is an instantiation of some other generic entity, this returns that entity. For instance, if the Ada code contains:

```
procedure Foo is new Generic_Proc (Integer);
```

and *e* is an instance of GPS.Entity for Foo, then *e.instance_of()* returns an entity for Generic_Proc.

Returns an instance of GPS.Entity or None

is_access()
Whether self is a pointer or access (variable or type)

Returns A boolean

is_array()
Whether self is an array type or variable.

Returns A boolean

is_container()
Whether self contains other entities (such as a package or a record).

Returns A boolean

is_generic()
Whether the entity is a generic.

Returns A boolean

is_global()
Whether self is a global entity.

Returns 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.

Returns A boolean

is_type()
Whether self is a type declaration (as opposed to a variable).

Returns A boolean

literals()
Returns the list of literals for an enumeration type.

Returns A list of instances of GPS.Entity

methods(*include_inherited=False*)
Returns 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()
Returns 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*)
Refactors the code at the location, to add named parameters. This only work if the language has support for such parameters, namely Ada for now.

Parameters **location** – An instance of GPS.FileLocation

```
GPS.Entity("foo", GPS.File("decl.ads")).rename_parameters(
    GPS.FileLocation(GPS.File("file.adb"), 23, 34))
```

overrides()
Returns the entity that self overrides.

Return type GPS.Entity

parameters()
Returns the list of parameters for entity. This is a list of entities. This applies to subprograms.

Returns A list of instances of GPS.Entity

parent_types()
Returns the list of parent types when self is a type. For example, 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()
Returns 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()
Returns the list of type for which self is a primitive operation (or a method, in other languages than Ada).

Returns A list of instances of GPS.Entity or []

references(*include_implicit=False*, *synchronous=True*, *show_kind=False*, *in_file=None*, *kind_in=''*)
Lists all references to the entity in the project sources. If include_implicit is true, implicit uses of

the entity are also referenced, for example when the entity appears as an implicit parameter to a generic instantiation in Ada.

If `synchronous` is True, the result is returned directly, otherwise a command is returned and its result is accessible with `get_result()`. The result, in that case, 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 (such as declaration, body, label, or end-of-spec). `in_file` can be used to limit the search to references in a particular file, which is. `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 A 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*)

Renames the entity everywhere 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, 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 *make_writable*: 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 but unsaved.

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()

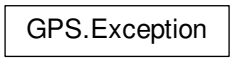
Displays in the type browser the informations known about the entity, such as the list of fields for records, list of primitive subprograms or methods, and list of parameters.

type()
Returns the type of the entity. For a variable, this 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`

15.5.30 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**
Represents a source file of your application.

See also:

`GPS.File.__init__()`

executable_path = None
Return a `File` instance of the executable associated with this file.

The result may be meaningless if the given `File` is not supposed to produce an executable.

path = ''
The absolute path name for the current instance of `GPS.File`, including directories from the root of the filesystem.

__init__(*name*, *local=False*)
Initializes a new instance of the class `File`. This does not need to be called explicitly, since GPS calls it automatically when you create such an instance. If name is a base file name (no directory is specified), GPS attempts to search for this file in the list of source directories of the project. If a directory is specified, or the base file name was not 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=''*)
Compiles the current file. This call returns only after 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()
Returns 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*)
Returns the list of entities that are either referenced (`local` is false) or declared (`local` is true) in self.

Parameters **local** – A boolean

Returns A list of GPS.Entity

generate_doc()
Generates the documentation fo the file and displays it in the default browsers.

get_property(*name*)
Returns 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*)
Returns 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, implicit dependencies are also returned. If `include_system` is true, dependent 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*)
Returns the the list of files that self depends on. If include_implicit is true, implicit dependencies are also returned. If include_system is true, then system files from the compiler runtime are also considered.

Parameters

- **include_implicit** – A boolean
- **include_system** – A boolean

Returns A list of files

See also:

GPS.File.imported_by()

language()
Returns 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=''*)
Compiles and links the file and all its dependencies. This call returns only after 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'*)
Returns 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, the function returns the equivalent path on the specified server. GPS_Server (default) is always the local machine. This argument is currently ignored.

This function returns the same value as the *self.path* property, and the latter might lead to more readable code.

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()
Returns 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*)

Returns the project to which file belongs. If file is not one of the souces of the project, the returned value depends on `default_to_root`: 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'*)

Returns 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 A list of GPS.FileLocation instances

See also:

GPS.EditorLocation.search()

GPS.File.search_next()

search_next(*pattern*, *case_sensitive=False*, *regexp=False*)

Returns 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()

unit()

Return the unit name for this file. For Ada source files, this is the unit name (i.e. the name of the package or the library-level subprogram). For other languages, this function always returns the empty string.

Returns a string

used_by()

Displays 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()
Displays in the dependency browser the list of files that file_name depends on.

See also:

GPS.File.used_by()

15.5.32 GPS.FileLocation

class GPS.**FileLocation**
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()
Returns the column of the location.

Returns An integer, the column of the location

See also:

GPS.FileLocation.file()

GPS.FileLocation.line()

file()
Returns 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()
Returns the line number of the location.

Returns An integer

See also:

GPS.FileLocation.file()

GPS.FileLocation.column()

15.5.33 GPS.FileTemplate

class GPS.**FileTemplate**
This class allows the user to create file templates from registered aliases.

static register (*alias_name*, *label*, *unit_param*, *language*, *is_impl*)
Register a new file template and create a 'New/create label contextual menu allowing users to create a new file from it for a given directory.

A file template is associated with the registered alias retrieved from alias_name: when clicking on the file template's contextual menu, a dialog asks the user to enter the alias parameters values and the expanded text of the alias is then used to prefill the new file.

The base name of the newly created file is deduced from the unit_param alias parameter value and the naming sheme deduced from the given language. Finally, the extension is computed from the is_impl boolean parameter, which indicates if the file is an implementation file or a specification file. The file is then placed in the directory from which the contextual menu was spawned.

Parameters

- **alias_name** (*str*) – the name of the alias to use
- **label** (*str*) – label used for displaying purposes
- **unit_param** (*str*) – the alias parameter to use for naming
- **language** (*str*) – the file template's language
- **is_impl** (*bool*) – whether it's an implementation file or not

15.5.34 GPS.Filter

class GPS.**Filter**
This class gives access to various aspects of the filters that are used by GPS to compute whether an action (and thus a menu, contextual menu or toolbar button) can be activated by the user at the current time.

static list ()
Return the list of all registered named filters. Instead of duplicating their implementation, it is better to reuse existing filters when possible, since their result is cached by GPS. Since lots of filters might be evaluated when computing the contextual menu, it will be faster when using named filters in such a case.

The returned named can be used in GPS.Action.create() for instance.

Returns a list of strings (the name of the filters)

15.5.35 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 disappears 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 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, the user will not be able to click on it.

Parameters **sensitive** (*boolean*) – A boolean

See also:

GPS.GUI.is_sensitive()

show()
Show again the graphical element that was hidden by hide().

See also:

GPS.GUI.hide()

15.5.36 GPS.HTML

class GPS.**HTML**

This class gives access to the help system of GPS as well as the integrated browser.

static add_doc_directory(*directory*)

Adds 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 containing the documentation

static browse(*URL*, *anchor=''*, *navigation=True*)

Opens the GPS HTML viewer, and loads the given URL. If anchor matches a <a> tag in this file, GPS jumps to it. If URL is not an absolute file name, it is searched in the path set by the environment variable GPS_DOC_PATH.

If navigation is True, the URL is saved in the navigation list, so 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 username 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
```

15.5.37 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%lt;/a%gt;
```

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()
Returns the name of the file that contains the description of the shell commands. You should not have to access it yourself, since you can do so using GPS.Help().getdoc() instead.

Returns A string

See also:

GPS.Help.getdoc()

getdoc(*name*, *html=False*)
Searches in the XML file :file'shell_commands.xml' for the documentation for this specific command or entity. If no documentation is found, an error is raised. If html is true, the documentation is formated 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()
Frees 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().

15.5.38 GPS.History

class GPS.**History**

This class gives access to GPS internal settings. These settings are used in addition to the preferences, and are used to keep information such as the list of files recently opened, or the state of various check boxes in the interface so that GPS can display them again in the same state when it is restarted.

__init__()

No instances of this class can be created.

static add(*key*, *value*)

Update the value of one of the settings. The new value is added to the list (for instance for recently opened files), and the oldest previous value might be removed, depending on the maximum number of elements that GPS wants to preserve for that key.

15.5.39 GPS.Hook

class GPS.**Hook**

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.

All standard hooks are documented in the GPS.Predefined_Hooks class.

__init__(*name*)

Creates a new hook instance, referring to one of the already defined hooks.

Parameters **name** – A string, the name of the hook

add(*function_name*, *last=True*)

Connects a new function to a specific hook. Any time this hook is run through run_hook(), this function is called with the same parameters passed to run_hook(). If last is True, this function is called after all functions currently added to this hook. If false, it is called before.

Parameters

- **function_name** – A subprogram, see the "Subprogram Parameters" section in the GPS documentation
- **last** – A boolean

See also:

GPS.Hook.remove()

```
def filed_edited(hook_name, file):
   print "File edited hook=" + hook_name + " file=" + file.name()
   GPS.Hook("file_edited").add(file_edited)
```

describe_functions()

Lists all the functions executed when the hook is executed. The returned list might contain <<internal> strings, which indicate that an Ada function is connected to this hook.

Returns A list of strings

static list()

Lists 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**()
Lists all defined hook types.

Returns A list of strings

See also:

GPS.Hook.register()

static **register**(*name*, *type=''*)
Definess 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 is described by the optional second parameter. The value of this parameter should be either the empty string for a hook that does not 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*)
Removes 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*)
Runs the hook. Calls all the functions that attached to that hook, and returns 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*)
Applies to hooks returning a boolean. Executes all functions attached to this hook until one returns False, in which case no further function is called. 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*)
Applies to hooks returning a boolean. 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()`

15.5.40 GPS.Predefined_Hooks

class GPS.**Predefined_Hooks**

This class is not available in GPS itself. It is included in this documentation as a way to describe all the predefined hooks that GPS exports.

Each function below describes the name of the hook (as should be used as parameter to GPS.Hook constructor), as well as the list of parameters that are passed by GPS.

activity_checked_hook(*name*)

Emitted when an activity status has been checked, the last step done after the activity has been committed. It is at this point that the activity closed status is updated.

Parameters **name** (*str*) –

after_character_added(*name*, *file*, *char*, *interactive*)

Emitted when a character has been added in the editor. This hook is also called for the backspace key.

See also:

`GPS.Predefined_Hooks.character_added()`

See also:

`GPS.Predefined_Hooks.word_added()`

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –

after_file_changed_detected(*name*)

Emitted when one or more opened file have been changed outside of GPS, and GPS needed to resynchronize it. This is called even when the user declined to synchronize.

Parameters **name** (*str*) –

annotation_parsed_hook(*name*)

Emitted when the last annotation has been parsed

Parameters **name** (*str*) –

before_exit_action_hook(*name*)

Emitted when GPS is about to exit. If the function returns False, the exit is aborted, and you should display a dialog to explain why

Parameters **name** (*str*) –

Returns bool

before_file_saved(*name*, *file*)
Emitted immediately before a file is saved

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –

bookmark_added(*name*, *str*)
Emitted when a new bookmark has been created by the user. The parameter is the name of the bookmark

Parameters

- **name** (*str*) –
- **str** (*str*) –

bookmark_removed(*name*, *str*)
Emitted when a bookmark has been removed by the user. The parameter is the name of the bookmark

Parameters

- **name** (*str*) –
- **str** (*str*) –

buffer_edited(*name*, *file*)
Emitted after the user has stopped modifying the contents of an editor

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –

build_mode_changed(*name*, *str*)

Parameters

- **name** (*str*) –
- **str** (*str*) –

build_server_connected_hook(*name*)
Emitted when GPS connects to the build server in remote mode

Parameters **name** (*str*) –

character_added(*name*, *file*, *char*, *interactive*)
Emitted 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).

See also:

`GPS.Predefined_Hooks.after_character_added()`

See also:

`GPS.Predefined_Hooks.word_added()`

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –

clipboard_changed(*name*)
Emitted when the contents of the clipboard has changed, either because the user added a new entry to it (Copy or Cut) or because the index of the last paste operation has changed (Paste Previous)

Parameters **name** (*str*) –

compilation_finished(*name*, *category*, *target*, *mode*, *shadow*, *background*, *status*)
Emitted 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"

Parameters

- **name** (*str*) –
- **category** (*str*) – location or highlighting category that contains the compilation output
- **target** (*str*) –
- **mode** (*str*) –
- **status** (*int*) –

compilation_starting(*name*, *category*, *quiet*, *shadow*, *background*)
Emitted when a compilation 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, no visible modification should be done in the MDI, such as raising consoles or clearing their content, since the compilation should happen in background mode.

Funtions 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.

```
# The following code adds a confirmation dialog to all
# compilation commands.
import gps_utils
@gps_utils.hook("compilation_starting")
def __compilation_starting(hook, category, quiet, *args):
   if not quiet:
      return MDI.yes_no_dialog("Confirm compilation ?")
   else:
      return True
```

```
# 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")
```

Parameters

- **name** (*str*) –
- **category** (*str*) – location or highlighting category that contains the compilation output
- **quiet** (*bool*) – If False, nothing should be reported to the user unless it is an error
- **shadow** (*bool*) – Whether the build launched was a Shadow builds, i.e. a secondary build launched automatically by GPS after a real build. For instance, when multiple toolchains mode is activated, the builds generating xref are Shadow builds
- **background** (*bool*) –

Returns bool

compute_build_targets (*name*, *str*)

Emitted 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), the name of the target and the full path for the project.

If *str* is not known, it should return an empty list.

The *str* parameter is the name of the target, for instance 'main', 'exec' or 'make'.

```
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)
```

Parameters

- **name** (*str*) –
- **str** (*str*) –

context_changed (*name*, *context*)

Emitted when the current context changes in GPS, such as when a new file or entity is selected, or a window is created

Parameters

- **name** (*str*) –
- **context** (*GPS.Context*) –

contextual_menu_close (*name*)

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 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:

`GPS.Predefined_Hooks.contextual_menu_open()`

Parameters **name** (*str*) –

contextual_menu_open(*name*)
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:

GPS.Predefined_Hooks.contextual_menu_close()

Parameters **name** (*str*) –

debugger_breakpoints_changed(*name*, *debugger*)
The list of breakpoints set in the debugger was reloaded. It might not have changed since the last time. The Debugger given in argument might actually be set to None when the list of breakpoints is changed before the debugger starts.

Parameters

- **name** (*str*) –
- **debugger** (*GPS.Debugger*) –

debugger_command_action_hook(*name*, *debugger*, *str*)
Called when the user types a command in the debugger console, or emits the command 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 (which is printed in the debugger console), or Debugger.Command_Intercepted to indicate the command was handled (but this is not output in the console)

```
## 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)
```

Parameters

- **name** (*str*) –
- **debugger** (*GPS.Debugger*) –
- **str** (*str*) –

Returns str

debugger_context_changed(*name*, *debugger*)
Emitted when the context of the debuggee has changed, for instance after thread switching, frame selection,...

Parameters

- **name** (*str*) –

- **debugger** (*GPS.Debugger*) –

debugger_executable_changed(*name*, *debugger*)
Emitted when the executable associated with the debugger has changed, for instance via /Debug/Debug/Open File. This is also called initially when the executable is given on the command line.

Parameters

- **name** (*str*) –
- **debugger** (*GPS.Debugger*) –

debugger_location_changed(*name*, *debugger*)
Emitted whenever the debugger reports a new current location, for instance when it stops at a breakpoint, when the user switches frame or thread,...

Parameters

- **name** (*str*) –
- **debugger** (*GPS.Debugger*) –

debugger_process_stopped(*name*, *debugger*)
Called when the debugger has ran and has stopped, for example 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".

See also:

`GPS.Predefined_Hooks.debugger_stated_changed()`

Parameters

- **name** (*str*) –
- **debugger** (*GPS.Debugger*) –

debugger_process_terminated(*name*, *debugger*)
' Emitted when the debugged process has finished

Parameters

- **name** (*str*) –
- **debugger** (*GPS.Debugger*) –

debugger_question_action_hook(*name*, *debugger*, *str*)
Emitted just before displaying an interactive dialog, when the underlying 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 any command to the debugger in this hook. The string parameter contains the debugger question.

```
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()
debug.send("print &foo")
```

Parameters

- **name** (*str*) –
- **debugger** (*GPS.Debugger*) –
- **str** (*str*) –

Returns str

debugger_started (*name*, *debugger*)
Emitted after the debugger has been spawned, and when it is possible to send commands to it. Better to use debugger_state_changed

See also:

`GPS.Predefined_Hooks.debugger_stated_changed()`

Parameters

- **name** (*str*) –
- **debugger** (*GPS.Debugger*) –

debugger_state_changed (*name*, *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 also called 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.

Parameters

- **name** (*str*) –
- **debugger** (*GPS.Debugger*) –
- **new_state** (*str*) –

debugger_terminated (*name*, *debugger*)
Emitted just before the connection to the debugger is closed. It is still possible to send commands. Better to use debugger_state_changed

Parameters

- **name** (*str*) –
- **debugger** (*GPS.Debugger*) –

desktop_loaded (*name*)

Parameters **name** (*str*) –

diff_action_hook (*name*, *vcs_file*, *orig_file*, *new_file*, *diff_file*, *title*)
Emitted to request the display of the comparison window

Parameters

- **name** (*str*) –
- **vcs_file** (*GPS.File*) –
- **orig_file** (*GPS.File*) –
- **new_file** (*GPS.File*) –
- **diff_file** (*GPS.File*) –
- **title** (*str*) – (default: "")

Returns bool

file_changed_detected(*name*, *file*)

Emitted 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.

This hook stops propagating as soon as a handler returns True. If you want get noticed systematically, use the *after_file_changed_detected* instead.

```
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)
```

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –

Returns bool

file_changed_on_disk(*name*, *file*)

Emitted 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.

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –

file_closed(*name*, *file*)

Emitted 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*.

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –

file_deleted (*name*, *file*)
Emitted 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.

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –

file_deleting (*name*, *file*)
+Emitted before GPS delete a file.

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –

file_edited (*name*, *file*)
Emitted when a file editor has been opened for a file that was not already opened before. Do not confuse with the hook open_file_action, which is used to request the opening of a file.

See also:

`GPS.Predefined_Hooks.open_file_action_hook()`

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –

file_line_action_hook (*name*, *identifier*, *file*, *every_line*, *tooltip*, *info*, *icon_name*)
Emitted to request the display of new information on the side of the editors. You usually will not 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. If Info is null or empty, existing line information is removed. If the first index in Info is 0, then space is reserved on the side for a new column, but no information is added yet. The first item should provide info to compute the maximum width of the column (text + icon). If the first index is -1, then extra information is added in the status bar (not on the side), using the provided Icon_Nam and Tooltip. Otherwise, information is added for all the lines with a corresponding entry in Info.

Parameters

- **name** (*str*) –
- **identifier** (*str*) –
- **file** (*GPS.File*) –
- **every_line** (*bool*) – (default: True)

file_renamed (*name*, *file*, *file2*)
Emitted whenever a GPS action renamed a file on the disk. *file* indicates the initial location of the file, while `renamed` 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.

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –
- **file2** (*GPS.File*) –

file_saved (*name*, *file*)
Emitted whenever a file has been saved

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –

file_status_changed (*name*, *file*, *status*)
Emitted when a file status has changed. The value for the status could be one of "UNMODIFIED", "MODIFIED" or "SAVED".

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –
- **status** (*str*) –

gps_started (*name*)
Emitted when GPS is fully loaded and its window is visible to the user. You should not 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.

Parameters **name** (*str*) –

html_action_hook (*name*, *url_or_file*, *enable_navigation*, *anchor*)
Emitted 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.

Parameters

- **name** (*str*) –
- **url_or_file** (*str*) –
- **enable_navigation** (*bool*) – (default: True)
- **anchor** (*str*) – (default: "")

Returns bool

location_changed (*name*, *file*, *line*, *column*, *project*)
Emitted when the location in the current editor has changed, and the cursor has stopped moving.

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –
- **line** (*int*) –
- **column** (*int*) –

log_parsed_hook (*name*)
Emitted when the last log has been parsed

Parameters **name** (*str*) –

marker_added_to_history (*name*, *location*)
Emitted when a new marker is added to the history list of previous locations, where the user can navigate backwards and forwards.

Parameters

- **name** (*str*) –
- **location** (*GPS.Location*) –

message_selected(*name*, *message*)

Parameters

- **name** (*str*) –
- **message** (*GPS.Message*) –

open_file_action_hook(*name*, *file*, *line*, *column*, *column_end*, *enable_navigation*, *new_file*, *force_reload*, *focus*, *project*, *group*, *initial_position*, *Areas*, *Title*)

Emitted when GPS needs to open a file. You can connect to this hook if you want to have your own editor open, instead of GPS's internal editor. Your function should return 1 if it did open the file or 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. *enable_navigation* 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.

See also:

GPS.Predefined_Hooks.file_edited()

```
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
```

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –
- **line** (*int*) – (default: 1) If -1, all editors for this file will be closed instead
- **column** (*int*) – (default: 1)
- **column_end** (*int*) –
- **enable_navigation** (*bool*) – (default: True)
- **new_file** (*bool*) – (default: True)
- **force_reload** (*bool*) –
- **focus** (*bool*) – (default: True)
- **project** (*GPS.Project*) –

Returns bool

preferences_changed(*name*, *pref*)
Emitted when the value of some of the preferences changes. Modules should refresh themselves dynamically.

Parameters **name** (*str*) –

project_changed(*name*)
Emitted 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:

GPS.Predefined_Hooks.project_view_changed()

Parameters **name** (*str*) –

project_changing(*name*, *file*)
Emitted just before a new project is loaded

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –

project_editor(*name*)
Emitted before the *Project Editor* is opened. This allows a custom module to perform specific actions before the actual creation of this dialog.

Parameters **name** (*str*) –

project_saved(*name*, *project*)
Emitted when a project is saved to disk. It is called for each project in the hierarchy.

Parameters

- **name** (*str*) –
- **project** (*GPS.Project*) –

project_view_changed(*name*)
Emitted 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.

Parameters **name** (*str*) –

revision_parsed_hook(*name*)
Emitted when the last revision has been parsed

Parameters **name** (*str*) –

rsync_action_hook(*name*, *synchronous*, *force*, *to_remote*, *print_output*, *print_command*, *tool_name*, *host_name*, *queue_id*, *file*)
internal use only

Parameters

- **name** (*str*) –
- **synchronous** (*bool*) –
- **force** (*bool*) –

- **to_remote** (*bool*) –
- **print_output** (*bool*) –
- **print_command** (*bool*) –
- **tool_name** (*str*) –
- **host_name** (*str*) –
- **queue_id** (*str*) –
- **file** (*GPS.File*) –

Returns bool

rsync_finished(*name*)

Parameters **name** (*str*) –

search_functions_changed(*name*)
Emitted when the list of registered search functions changes.

Parameters **name** (*str*) –

search_regexps_changed(*name*)
Emitted when a new regexp has been added to the list of predefined search patterns.

Parameters **name** (*str*) –

search_reset(*name*)
Emitted 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.

Parameters **name** (*str*) –

semantic_tree_updated(*name*, *file*)
Emitted when the semantic_tree for a file has been updated.

Parameters

- **name** (*str*) –
- **file** (*GPS.File*) –

server_config_hook(*name*, *server*, *nickname*)
Emitted when a server is assigned to a server operations category.

The *server_type* parameter is the server operations category. It can take the values "BUILD_SERVER", "EXECUTION_SERVER" or "DEBUG_SERVER".

Parameters

- **name** (*str*) –
- **server** (*str*) –
- **nickname** (*str*) –

server_list_hook(*name*)
Emitted when the list of configured servers has changed.

Parameters **name** (*str*) –

source_lines_folded(*name*, *context*, *line1*, *line2*)

Parameters

- **name** (*str*) –

- **line1** (*int*) –
- **line2** (*int*) –

source_lines_unfolded(*name*, *context*, *line1*, *line2*)

Parameters

- **name** (*str*) –
- **line1** (*int*) –
- **line2** (*int*) –

status_parsed_hook(*name*)
Emitted when the last status has been parsed

Parameters **name** (*str*) –

stop_macro_action_hook(*name*)
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.

Parameters **name** (*str*) –

task_started(*name*)
Emitted when a new background task is started

Parameters **name** (*str*) –

variable_changed(*name*)
Emitted when one of the scenario variables has been renamed, removed or when one of its possible values has changed.

Parameters **name** (*str*) –

vcs_active_changed(*name*)
Emitted when the active VCS has changed. This is the VCS on which operations like commit and log happen.

Parameters **name** (*str*) –

vcs_file_status_changed(*VCS*, *files*, *props*)
Emitted when the VCS status of a file has been recomputed. The file might now be up to date, staged for commit, locally modified,... It might also have a different version number, for file-based systems. This hook is only called on actual change of the status, and provides basic information on the new status. Check GPS.VCS.file_status to get more details.

Parameters

- **VCS** (*GPS.VCS*) –
- **files** (*[GPS.File]*) –
- **props** (*int*) –

vcs_refresh(*name*)
Run this hook to force a refresh of all VCS-related views. They will resynchronize their contents from the disk, rather than rely on cached information

Parameters **name** (*str*) –

word_added(*name*, *file*)
Emitted when a word has been added in an editor.

See also:

GPS.Predefined_Hooks.character_added()

Parameters

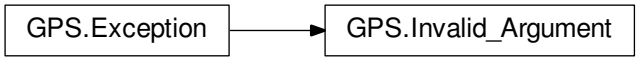
- **name** (*str*) –
- **file** (*GPS.File*) –

xref_updated (*name*)
Emitted when the cross-reference information has been updated.

Parameters **name** (*str*) –

15.5.41 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 example).



15.5.42 GPS.Language

class GPS.**Language**
A few methods can be overridden when you create your own child class of GPS.Language, to provide support for the Outline view. They are not defined by default, and thus the documentation is given below:

def parse_constructs(self, constructs_list, gps_file, content_string): ‘” Abstract method that has to be implemented by the subclasses.

Given an empty list of constructs, a file instance and a string containing the contents of the file, this needs to populate the list of language constructs. In turn this will give support for a number of features in GPS including:

- Outline support
- Block highlighting/folding support
- Entity search support

param GPS.ConstructList constructs_list The list of constructs to populate.

param GPS.File gps_file the name of the file to parse.

param str content_string The content of the file

‘’‘

def should_refresh_constructs(self, file): ‘‘‘ Whether GPS should call parse_constructs to refresh the list. This is called when the file has not changed on the disk, but GPS thinks there might be a need to refresh because various hooks have been run. By default, this returns False, so that parse_constructs is only called when the file changes on the disk.

param GPS.File file the file to test

return a bool

’’’

def clicked_on_construct(self, construct): ‘‘‘ Called when the user wants to jump to a specific construct. The default is to open an editor for the file/line/column.

param GPS.Construct construct the construct as build in `GPS.Language.parse_constructs()`.

’’’

__init__()

This constructor is provided to prevent the initialisation of any object of the Language class, because it is abstract. The consequence of this is that subclassers of Language must reimplement __init__ to avoid having an exception raised at instance construction time

static get(*name*)

Return a description of the language, from its name. For instance:

```
GPS.Language.get('ada').keywords
```

or:

```
GPS.EditorBuffer.get().get_lang().keywords
```

Returns a `GPS.LanguageInfo`

static register(*instance*, *name*, *body_suffix*, *spec_suffix=’‘*, *obj_suffix=’‘*, *indentation_kind=1*)

Register an instance of language in GPS.

Parameters

- **instance** (*Language*) – The instance you want to register
- **name** (*string*) – The name of the language
- **body_suffix** – The file suffix for the language - ”.c” for the C language for example
- **spec_suffix** – The file suffix for specification files for the language, if it applies - ”.h” for the C language.
- **obj_suffix** – The suffix for object files produced for the language, if it applies - ”.o” for the C language.
- **indentation_kind** (*int*) – One of the INDENTATION_NONE, INDENTATION_SIMPLE or INDENTATION_EXTENDED constants defined in the constructs module, defining the way the language will be indented.

15.5.43 GPS.LanguageInfo

class GPS.**LanguageInfo**

This class gives access to various information known about the programing languages supported by GPS.

keywords = ''
Return a regular expression that can be used to test whether a string is a keyword for the language. The regexp is anchored with '^' and ends with '\b' (word separator).

name = ''
Return the name of the language

15.5.44 GPS.Libclang

class GPS.**Libclang**

static get_translation_unit (*file*, *project=None*)
Returns the clang translation unit corresponding to this file. You can use that as a full libclang translation unit.

Parameters **file** (*GPS.File*) – The file to get the translation unit for

Return type clang.cindex.TranslationUnit

15.5.45 GPS.Locations

class GPS.**Locations**
General interface to the *Locations* view.

static add (*category*, *file*, *line*, *column*, *message*, *highlight=''*, *length='0'*, *look_for_secondary=False*)
Adds a new entry to the *Locations* view. Nodes are created as needed for `category` or `file`. If `highlight` is specified as a non-empty string, the enter line is highlighted in the file with a color determined by that highlight category (see `register_highlighting()` for more information). `length` is the length of the highlighting; the default of 0 indicates 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*)
Dumps the contents of the *Locations* view to the specified file, in XML format.

Parameters **file** – A string

static list_categories()
Returns the list of all categories currently displayed in the *Locations* view. These are the top-level nodes used to group information generally related to one command, such as the result of a compilation.

Returns A list of strings

See also:

GPS.Locations.remove_category()

static list_locations(*category*, *file*)
Returns 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'*)
Parses the contents of the string, which is supposedly the output of some tool, and adds the errors and warnings to the *Locations* view. A new category is created in the locations window if it does not exist. Preexisting contents for that category are 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*)
Removes a category from the *Locations* view. 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 overrided by user.

Parameters **category** – A string ("Chronological" or "Alphabetical")

15.5.46 GPS.Logger

class GPS.**Logger**
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, `$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 or whether timestamps should be logged with each message.

active = True
Whether this logging stream is active

count = None

__init__ (*name*)
Creates 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 creates 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` is False, `error_message` is logged in the log file. If True, `success_message` is logged if present.

Parameters

- **condition** – A boolean
- **error_message** – A string
- **success_message** – A string

```
log=GPS.Logger("my_script")
log.check(1 == 2, "Invalid operation")
```

log (*message*)
Logs a message in the GPS log file.

Parameters **message** – A string

set_active (*active*)
Activates or deactivates a logging stream. The default for a sttream 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

15.5.47 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 the pygobject package is installed, 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 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()
```

FLAGS_ALL_BUTTONS = 4

FLAGS_ALWAYS_DESTROY_FLOAT = 16

FLAGS_DESTROY_BUTTON = 4

FLAGS_FLOAT_AS_TRANSIENT = 8

FLAGS_FLOAT_TO_MAIN = 32

GROUP_CONSOLES = 107

GROUP_DEBUGGER_DATA = 104

GROUP_DEBUGGER_STACK = 103

GROUP_DEFAULT = 0

GROUP_GRAPHS = 101

GROUP_VCS_ACTIVITIES = 105

GROUP_VCS_EXPLORER = 102

GROUP_VIEW = 106

POSITION_AUTOMATIC = 0

POSITION_BOTTOM = 1

POSITION_FLOAT = 5

POSITION_LEFT = 3

POSITION_RIGHT = 4

POSITION_TOP = 2

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 be manipulated by the user like any other standard GPS window. For example, it can be split, floated, or 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).

This function has no effect if the widget is already in the MDI. In particular, the save_desktop parameter will not be taken into account in such a case.

Parameters

- **widget** – A widget, created by pygobject, or an instance of GPS.GUI or one of the derived classes.
- **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()
Returns all the windows currently in the MDI.

Returns A list of GPS.MDIWindow

static current()
Returns the window that currently has the focus, or None if there is none.

Returns An instance of GPS.MDIWindow or None

static current_perspective()
The name of the current perspective.

Returns str

static dialog(*msg*)
Displays 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'*)
Displays 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*)
Returns 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*)
Returns the window that contains `child` or raises 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 information_popup(*text=''*, *icon=''*)
Display a temporary information popup on the screen. This popup automatically disappears after a short while, so should only be used to indicate success or failure for an action, for instance.

Parameters

- **text** (*str*) – The text to display.
- **icon** (*str*) – The name of an icon to display beside the text.

static input_dialog(*msg*, **args*)
Displays a modal dialog and requests 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 prepend 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 load_perspective(*name*)
Change the current perspective to the one designated by `name`. This function does nothing if `name` does not refer to any known perspective.

Parameters **name** – A string

static save_all(*force=False*)
Saves all currently unsaved windows. This includes open editors, the project, and any other window that has registered some save callbacks.

If `force` is false, a confirmation dialog is displayed so 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*)
Displays 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.

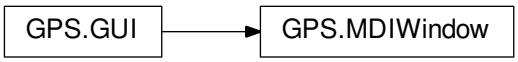
Parameters **msg** – A string

Returns A boolean

```
if GPS.MDI.yes_no_dialog("Do you want to print?"):
    print "You pressed yes"
```

15.5.48 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.

close(*force=False*)
Close the window. If force is False, give the window an opportunity to prevent its deletion (for instance through a save confirmation dialog).

Parameters **force** – A boolean

float(*float=True*)
Floats the window, i.e., creates 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()
Returns 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()
Returns True if the window is currently floating (i.e., in its own toplevel window) or False if the window is integrated into the main GPS window.

Returns A boolean

name(*short=False*)
Returns the name of the window. If short is False, the long name is returned, the one that appears in the title bar. If True, the short name is returned, the one that appears in notebook tabs.

Parameters **short** – A boolean

Returns A string

next(*visible_only=True*)
Returns the next window in the MDI, or the current window if there is no other window. If visible_only is True, only the windows currently visible to the user can be returned. This always returns floating windows.

Parameters **visible_only** – A boolean

Returns An instance of GPS.MDIWindow

raise_window()
Raises the window so that it becomes visible to the user. The window also gains the focus.

rename (*name*, *short=''*)
Changes the title used for a window. `name` is the long title, as it appears in the title bar, and `short`, if specified, is the name that appears in notebook tabs.

Using this function may be dangereous in some contexts, since GPS keeps track of editors through their name.

Parameters

- **name** – A string
- **short** – A string

split (*vertically=True*, *reuse=False*, *new_view=False*)
Splits the window in two parts, either horizontally (side by side), or vertically (one below the other).

Parameters

- **vertically** (*bool*) –
- **reuse** (*bool*) – whether to reuse an existing space to the side of current window, rather than splitting the current window. This should be used to avoid ending up with too small windows.
- **new_view** (*bool*) – whether to create a new view when the current window is an editor.

See also:

`GPS.MDIWindow.single()`

15.5.49 GPS.MemoryUsageProvider

class `GPS.`**MemoryUsageProvider**
General interface used to populate the GPS Memory Usage View.

In practice, this class is derived in the code to provide memory usage providers that are specific to ones or more external tools (e.g: a memory usage provider that fetches data generated from the ld linker).

15.5.50 GPS.MemoryUsageProviderVisitor

class `GPS.`**MemoryUsageProviderVisitor**
This class is used to notify GPS of events that occur during a memory usage provider task (e.g: when a memory usage provider has finished to fetch all the memory usage data needed by the Memory Usage View).

on_memory_usage_data_fetched (*regions*, *sections*, *modules*)
Report when a `GPS.MemoryUsageProvider` finished to fetch all the memory usage data of the last built executable (i.e: memory regions and memory sections and modules).

This method is called in *GPS.MemoryUsageProvider.async_fetch_memory_usage_data*.

Note that the given `GPS.MemoryUsageProviderVisitor` instance is freed after calling this method.

Parameters

- **regions** – a list of (name, origin, length) tuples describing memory regions.
- **sections** – a list of (name, origin, length, region_name) tuples describing memory sections.

- **modules** – a list of (obj_file, lib_file, origin, size,

  region_name, section_name) tuples describing modules, which are file based split of ressources consumed: obj_file and lib_file repectively correspond to the full paths of the object file and, if any, of the library file for which this artifact was compiled.

15.5.51 GPS.Menu

class GPS.**Menu**

This class is a general interface to the menu system in GPS. It gives you control over such things as which menus should be active and what should be executed when the menu is selected by the user.

See also:

GPS.Menu.__init__()

action = None

The GPS.Action executed by this menu

__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 you always get the same instance of GPS.Menu when refering to a given menu in GPS and so you can store your own specific data with the menu.

static create(*path*, *on_activate=''*, *ref=''*, *add_before=True*, *filter=None*, *group=''*)

Creates 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 path is valid.

It is recommended now to use gps_utils.interactive instead of creating menus explicitly. The latter creates GPS actions, to which keybindings can be associated with the user. They can also be executed more conveniently using keyboard only with the omni-search.

If on_activate is specified, it is 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/-", 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 example, if you create the menu "/_File", then the user can open the menu by pressing Alt-f. But the underscore itself is 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
```

destroy()
Remove the menu and all other graphical elements linked to the same action.

static get(*path*)
Returns the menu found at the given path. `path` is similar to file paths, starting with the main GPS menu ('/'), down to each submenus. For example, '/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)
```

hide()
Disable the action associated with the menu

pywidget()

set_sensitive(*sensitive=True*)
Disable the action associated with the menu

show()
Enable the action associated with the menu

15.5.52 GPS.Message

class GPS.**Message**
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

__init__(*category*, *file*, *line*, *column*, *text*, *show_on_editor_side=True*, *show_in_locations=True*, *allow_auto_jump_to_first=True*)
Adds 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 containg the message text
- **show_on_editor_side** (*bool*) – Whether to show the message in the editor's gutter
- **show_in_locations** (*bool*) – Whether to show the message in the locations view
- **allow_auto_jump_to_first** (*bool*) – If True, then adding a message that is the first for its category will auto jump the editor to it, if the corresponding preference is activated

```
# Create a message

m=GPS.Message("default", GPS.File("gps-main.adb"),
      1841, 20, "test message")

# Remove the message
m.remove()
```

execute_action()
If the message has an associated action, executesa it.

get_category()
Returns the message's category.

get_column()
Returns the message's column.

get_file()
Returns the message's file.

get_flags()
Returns an integer representing 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, this set of flags can be extended in the future, so they should be viewed as bits that are "or"ed together.

get_line()
Returns the message's line.

get_mark()
Returns an `EditorMark` which was created with the message and keeps track of the location when the file is edited.

get_text()
Returns the message's text.

static list(*file=None*, *category=None*)
Returns 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()
Removes the message from GPS.

set_action(*action*, *image*, *tooltip=None*)
Adds an action item to the message. This adds an icon to the message; Clicking on the icon executes `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 containing 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 the *Locations* view.

Parameters

- **category** – Name of messages category
- **hint** – Default sorting method ("chronological" or "alphabetical")

set_style(*style*, *len*)
Sets the style of the message. `len` is the length in number of characters to highlight. If 0, highlight the whole line. If omitted, the length of the message highlighting is not modified.

Parameters

- **style** – An integer
- **len** – An integer

set_subprogram(*subprogram*, *image*, *tooltip=None*)
Adds an action item to the message. This adds an icon to the message. Clicking on this icon calls `subprogram`, with the message passed as its parameter.

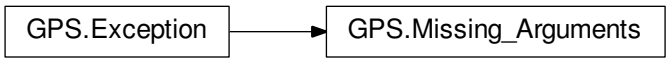
Parameters

- **subprogram** – A subprogram in the scripting language. This subprogram takes on parameter, which is a 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()]
```

15.5.53 GPS.Missing_Arguments

class `GPS.`**Missing_Arguments**
An exception raised by GPS. Raised when calling a subprogram from the GPS module with missing arguments.



15.5.54 GPS.OutputParserWrapper

class GPS.**OutputParserWrapper**

This class is used to handle user-defined tool output parsers. Parsers are organized in chains. Output of one parser is passed as input to next one. Chains of parser could be attached to a build target. This class is for internal use only. Instead users should inherit custom parser from `OutputParser` defined in `tool_output.py`, but their methods 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 specifying it in an XML file.

```
<target model="myTarget" category="_Run" name="My Target">
   <output-parsers>[default] popupparser</output-parsers>
</target>
```

Where [default] abbreviates names of all parsers predefined in GPS.

__init__(*child=None*)

Creates a new parser and initialize its child reference, if provided.

on_exit(*status*, *command*)

Called when all output is parsed to flush any buffered data at end of the stream.

on_stderr(*text*, *command*)

Like `on_stdout()`, but for the error stream.

on_stdout(*text*, *command*)

Called each time a portion of output text is ready to parse. Takes the portion of `text` as a parameter and passes filtered portion to its child.

15.5.55 GPS.Preference

class GPS.**Preference**

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 &amp;amp; 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 name, which is the one that is found in the $HOME/.gps/preferences file. When you are creating a new preference, this name can include '/' characters, which results in subpages created in the *Preferences* dialog. The name after the last '/' should only include letters and '-' characters. You can also specify a group before the last '/', by appending a ':' delimitor followed by the name of the preference's group. If the name starts with '/' and contains no other '/', the preference is not visible in the *Preferences* dialog, though it can be manipulated as usual and is loaded automatically by GPS on startup.

Parameters **name** – A string

create (*label*, *type*, *doc=''*, *default=''*, **args*)

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. label qualifies the preference and doc appears as a tooltip to explain the preference to users. 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 "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 an "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

Returns The preference itself

create_style (*label*, *doc=''*, *default_fg=''*, *default_bg='white'*, *default_font_style='default'*)
Creates a new text style preference, which enables the user to choose between different text style characteristics, namely foreground color, background color, and wether the text is bold, italic, both, or neither.

Parameters

- **label** (*string*) – The label of the preference
- **doc** (*string*) – The documentation of the preference
- **default_fg** (*string*) – The default foreground color for this preference, as a CSS-like color.
- **default_bg** (*string*) – The default background color for this preference, as a CSS-like color
- **default_font_style** (*string*) – The style, one of "default", "normal", "bold", "italic" or "bold_italic"

get ()
Gets value of 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*)
Sets `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.

15.5.56 GPS.PreferencesPage

class GPS.**PreferencesPage**
Interface to the GPS preferences pages, as set in the *Edit → Preferences* dialog.

This interface can be used to create custom preferences pages.

static create (*name*, *get_widget*, *priority=-1*, *is_integrated=False*)
Create a new preferences page and makes it visible in the *Preferences* dialog, adding an entry with the given `'name`.

Each time the page is selected, the PyGtk widget returned by the `get_widget''function is displayed. Note that this widget is destroyed when closing the preferences dialog: thus, ''get_widget` can't return the same widget twice and should create a new one each time it is called instead.

The `priority` is used to order the preferences pages in the *Preferences* dialog tree view, using the following policy:

•Pages with higher priorities are listed at the top of the tree view.

•If two pages have the same priority, the alphabetical order determines which page will appear first.

when `is_integrated` is True, the preferences editor dialog will not create an entry for this page in its left tree view. This is generally needed for pages that are integrated in another visible preferences pages or for pages displayed in the GPS preferences assistant.

Parameters

- **name** – A string
- **get_widget** – function returning a PyGtk widget
- **priority** – integer defining the page's priority
- **is_integrated** – A boolean

15.5.57 GPS.Process

class GPS.**Process**

Interface to **expect**-related commands. This class can be used to spawn new processes and communicate with them later. It is similar to what GPS uses to communicate with **gdb**. This is a subclass of GPS.Command.

See also:

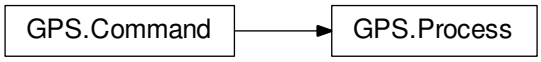
GPS.Process.__init__()

GPS.Command()

```
# The following example launches a gdb process, lets 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 does not 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*, *active=False*, *directory=''*, *block_exit=True*)

Spawns `command`, which can include triple-quoted strings, similar to Python, which are always preserved as one argument.

The external process might not start immediately. Instead, it will start whenever GPS starts processing events again (once your script gives the hand back to GPS), or when you call `expect()` or `get_result()` below.

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 "^" 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 tasks view, 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 True, the process will be visible in the GPS tasks view and can be interrupted or paused by users. Otherwise, it is running in the background and never visible to the user. If progress_regexp is specified, the output of the process will be scanned for this regexp. The part that matches 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 example, 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 tasks view of GPS, and will allow users to monitor commands.

An exception is raised if the process could not be spawned.

param command A string or list of strings. The list of strings is preferred, since it provides a better handling of arguments with spaces (like filenames). When you are using a string, you need to quote such arguments.

param regexp A string

param on_match A subprogram, see the section "Subprogram parameters" in the GPS documentation

param on_exit A subprogram

param task_manager A boolean

param progress_regexp A string

param progress_current An integer

param progress_total An integer

param before_kill A subprogram

param str remote_server Possible values are "GPS_Server", the empty string (equivalent to "GPS_Server"), "Build_Server", "Debug_Server", "Execution_Server" and "Tools_Server". This 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.

param bool show_command if True, the command line used to spawn the new process is displayed in the *Messages* console.

param bool single_line_regexp

param bool case_sensitive_regexp

param bool strip_cr If true, the output of the process will have all its ASCII.CR removed before the string is passed 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 example, those that drive an ANSI terminal).

param bool active Whether GPS should actively monitor the state of the process. This will require more CPU (and might make the GUI less reactive while the process runs), but ensures that events like on_exit will be called earlier.

param str directory The directory in which the external process should be started.

param bool block_exit If true, then GPS will display a dialog when the user wants to exit, asking whether to kill this process.

See also:

GPS.Process

expect(*regexp*, *timeout=-1*)

Blocks execution of the script until either `regexp` has been seen in the output of the command or `timeout` has expired. If `timeout` is negative, wait forever until we see `regexp` or the process completes 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 and up to the end of the text that matched regexp. Note that it also includes the output sent to the on_match callback while it is running. It does not, however, include output already returned by a previous call to this function (nor does it guarantee that two successive calls 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 call).

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()

Waits untill the process terminates and returns its output. This is the output since the last call to this function, so if you call it after performing some calls to `expect()`, the returned string does not contain the output already returned by `expect()`.

Returns A string

interrupt()

Interrupts a process controlled by GPS.

kill()

Terminates a process controlled by GPS.

send(*command*, *add_lf=True*)

Sends 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 example 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()

Blocks 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

15.5.58 GPS.Project

class GPS.**Project**

Represents a project file. Also see 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, such as when one of its attributes was changed by the user or the environment variables are changed.

This is a good time to test the list of languages (GPS.Project.languages()) that the project supports and do language-specific customizations

•project_changed

Called when a new project was loaded. The hook above is called after this one.

target = None

Returns the Target project attribute value or an empty string if not defined.

If the given project extends from another project, the attribute is also looked up in the extended project.

__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*)

Adds some values to an attribute. You can add 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()

For example:

```
GPS.Project.root().add_attribute_values(
    "Default_Switches", "Compiler", "ada", "-gnatwa", "-gnatwe");
```

add_dependency (*path*)

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*)

Adds some main units to the current project 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=''*)

Adds some predefined directories to the source path or the objects path. These are 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 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*)

Adds a new source directory to the project. The new directory is added in front of the source path. You should call GPS.Project.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 is overriden 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()
Returns the list of projects that might contain sources that depend on the project's sources. When doing extensive searches it is not 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*)

Clears 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*)
Returns the list of projects on which self depends (either directly if `recursive` is False, or including indirect dependencies if True).

Parameters **recursive** – A boolean

Returns A list of `GPS.Project` instances

exec_dir()
Return the directory that contains the executables generated for the main programs of this project. This is either Exec_Dir or Object_Dir.

Returns A string

external_sources()
Return the list of all sources visible to the builder, but that are not part of a project. This includes sources found in one of the predefined directories for the builder, or sources found in the directories references in the ADA_SOURCE_PATH environment variable.

Returns A list of instances of `GPS.File`

file()
Returns the project file.

Returns An instance of `GPS.File`

generate_doc(*recursive=False*)
Generates the documentation for the projet (and its subprojects if `recursive` is True) and displays it in the default browser.

get_attribute_as_list(*attribute*, *package=''*, *index=''*)
Fetches the value of the attribute in the project.

If `package` is not specified, the attribute at the toplevel of the project is queried.

`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=''*)

Fetches the value of the attribute in the project.

If `package` is not specified, the attribute at the toplevel of the project is queried.

`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 explicitely 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*)
Returns the name of the executable, either read from the project or computed from *main*.

Parameters **main** (*GPS.File*) – the main source file.

Returns A string

get_property(*name*)
Returns 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 exists for the project.

Parameters **name** – A string

Returns A string

See also:

GPS.Project.set_property()

get_tool_switches_as_list(*tool*)
Like get_attribute_as_list(), but specialized for the switches of 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*)

Like GPS.Project.get_attribute_as_string(), but specialized for a 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_harness_project()

Returns True if the project is a harness project generated by gnattest tool.

Returns A boolean

is_modified(*recursive=False*)

Returns True if the project has been modified but not saved yet. If recursive is true, the return value takes into account all projects imported by self.

Parameters **recursive** – A boolean

Returns A boolean

languages(*recursive=False*)

Returns the list of languages 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*)

Loads a new project, which replaces the current root project, and returns 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, the user will not be asked whether to save the current project, whether it was modified or not.

If `keep_desktop` is False, load the saved desktop configuration, otherwise keep the current one.

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()

Returns the name of the project. This does not include directory information; use `self.file().name()` if you want to access that information.

Returns A string, the name of the project

object_dirs (*recursive=False*)

Returns 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

original_project()

For given harness project returns the name of the original user project, which harness was generated from. Returns No_Project if this is not a harness project.

Returns An instance of `GPS.Project`

properties_editor()

Launches a graphical properties editor for the project.

static recompute()

Recomputes 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.

```
GPS.Project.recompute()
```

remove_attribute_values (*attribute*, *package*, *index*, *value*)

Removes specific values from an attribute. You can specify as many values you need at the end of the param list.

If `package` is not specified, the attribute at the toplevel of the project is queried.

`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()`

For example:

```
GPS.Project.root().remove_attribute_values(
    "Default_Switches", "Compiler", "ada", "-gnatwa", "-gnatwe");
```

remove_dependency(*imported*)

Removes 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*)

Removes a source directory from the project. You should call `GPS.Project.recompute()` after calling this method to recompute the list of source files. The directory is added only for the current value of the scenario variables.

Parameters **directory** – A string

See also:

`GPS.Project.add_source_dir()`

rename(*name*, *path='<current path>'*)

Renames and moves a project file (the project is only put in the new directory when it is saved, but is not removed from its original directory). You must call `GPS.Project.recompute()` at some point after changing the name.

Parameters

- **name** – A string
- **path** – A string

static root()

Returns the root project currently loaded in GPS.

Returns An instance of GPS.Project

```
print "Current project is " + GPS.Project.root().name()
```

static scenario_variables()

Returns the list of scenario variables for the current project hierarchy and their current values. 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 does not change the value of the scenario variable.

Returns hash table associating variable names and values

See also:

GPS.Project.set_scenario_variable()

For example:

```
GPS.Project.scenario_variables()["foo"]
=> returns the current value for the variable foo
```

static scenario_variables_cmd_line(*prefix=''*)

Returns a concatenation of VARIABLE=VALUE, each preceded by `prefix`. This string is generally used when calling external tools, for example, **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_cmd_line()</shell> \
#  <external>make %1</external> \
# </action>
```

static scenario_variables_values()

Returns a hash table where keys are the various scenario variables defined in the current project and values the different values that this variable can accept.

Returns A hash table of strings

search(*pattern*, *case_sensitive=False*, *regexp=False*, *scope='whole'*, *recursive=True*)

Returns the list of matches for pattern in all the files belonging to the project (and its imported projects if recursive is true, which is the 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
- **regexp** – 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 `package` is not specified, the attribute at the toplevel of the project is queried. `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` is 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*)
Changes 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).

If name does not correspond to an actual scenario variable in your project (i.e. the name of the variable in an "external(...)" typed expression), the corresponding environment variable is still changed. This might impact the reloading of the project, for instance when "external(...)" is used to construct the name of a directory, as in:

```
for Object_Dir use external("BASE") & "/obj";
```

Parameters

- **name** – A string
- **value** – A string

See also:

GPS.Project.scenario_variables()

source_dirs(*recursive=False*)

Returns 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*)

Returns the list of source files for this project. If recursive is true, 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 see while traversing the project hierarchy.

Parameters **recursive** – A boolean

Returns A list of instances of GPS.File

15.5.59 GPS.ProjectTemplate

class GPS.**ProjectTemplate**

This class is used to manipulate GPS Project Templates.

static add_templates_dir(*noname*)

Adds 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.

15.5.60 GPS.ReferencesCommand

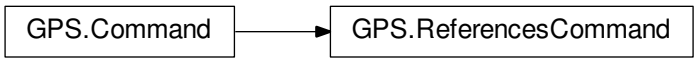
class GPS.**ReferencesCommand**

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()

15.5.61 GPS.Revision

class GPS.**Revision**
General interface to the revision browser.

static add_link(*file*, *revision_1*, *revision_2*)
Creates a link between revision_1 and revision_2 for file.

Parameters

- **file** – A string
- **revision_1** – A string
- **revision_2** – A string

static add_log(*file*, *revision*, *author*, *date*, *log*)
Adds 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*)
Registers a new symbolic name (tag or branches) corresponding to the revision of file.

Parameters

- **file** – A string
- **revision** – A string
- **symbolic_name** – A string

static clear_view(*file*)
Clears the revision view of file.

Parameters **file** – A string

15.5.62 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, and actions, etc.

This class provides facilities exported directly by Ada, so you can for example look for file names by writting:

```
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*)
Looks up one of the existing search factories.

Parameters **name** – a string, one of, e.g., GPS.Search.FILE_NAMES, GPS.Search.SOURCES

next()
Results the next non-null result. This might take longer than `get()`, since it keeps looking until it actually finds a new result. It raises StopIteration when there are no more results.

static register(*name*, *factory*, *rank=-1*)
Registers a new custom search. This will be available to users via the omni-search in GPS, or 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.
- **rank** – the search order for the provider. If negative, the new provider is added last. Other providers might be registered later, though, so the rank could change. User preferences will also override that rank.

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 example GPS.Search.SOURCES
- **pattern** – a string
- **flags** – an integer, see GPS.Search.set_pattern()

set_pattern (*pattern*, *flags=0*)
Sets 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

15.5.63 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 ()
Executes the action associated with the result. This action depends on where you were searching. For example, 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.

15.5.64 GPS.SemanticTree

class GPS.**SemanticTree**
This class represents the semantic information known to GPS for a given file.

__init__ (*file*)
Creates a SemanticTree.

Parameters **file** – A File.

is_ready()
Return True if and only if the semantic tree for this file is available

Returns A boolean.

update()
Ask for an immediate recomputation of the sematic tree.

This should be used by custom implementations of semantic trees, to force GPS to ask for the new contents of the tree.

15.5.65 GPS.Style

class GPS.**Style**
This class is used to manipulate GPS Styles, which are used, for example, 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*)
Creates 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()
Returns 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()
Returns a list of all styles currently registered in GPS.

Returns a list of GPS.Style

set_background(*noname*)
Sets the background of style to the given color.

Parameters **noname** – A string representing a color, for instance "blue" or "#0000ff"

set_foreground(*noname*)
Sets 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

15.5.66 GPS.SwitchesChooser

class GPS.**SwitchesChooser**
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()
Returns the tool's command line parameter.

Returns A string

set_cmd_line(*cmd_line*)
Modifies the widget's aspect to reflect the command line.

Parameters **cmd_line** – A string

15.5.67 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 *Tasks* view.

EXECUTE_AGAIN = 'execute_again'

FAILURE = 'failure'
Whether the task has a visible progress bar in GPS's toolbar or the Tasks view.

SUCCESS = 'success'

visible = False

__init__ (*name*, *execute*, *active=False*, *block_exit=False*)
Create a task.

Parameters

- **name** – A string identifying the task.
- **execute** – a function which takes the task as parameter and returns one of the constants:

  GPS.Task.EXECUTE_AGAIN if execute should be reexecuted by GPS GPS.Task.SUCCESS if the task has terminated successfully GPS.Task.FAILURE if the task has terminated unsuccessfully
- **active** – A boolean. By default the 'execute' functions are executed in the background approximately every 100ms - setting this to True makes GPS run the 'execute' function much more aggressively, every time the GUI is idle. Use this with caution, as this might impact the responsiveness of the user interface.
- **block_exit** – A boolean. Set this to True if a confirmation popup should appear when the task is running and GPS has been asked to quit.

block_exit ()
Returns True if and only if this task should block the exit of GPS.

Returns A boolean

interrupt ()
Interrupts the task.

static list ()

Returns a list of GPS.Task, all running tasks

name ()
Returns the name of the task.

Returns A string

pause ()
Pauses the task.

progress ()
Returns the current progress of the task.

Returns A list containing the current step and the total steps

resume ()
Resumes the paused task.

set_progress (*current*, *total*)
Sets the progress indication for this task.

Parameters

- **current** – an integer, the current progress.
- **total** – an integer, the total progress.

status ()
Returns the status of the task.

Returns A string

15.5.68 GPS.Timeout

class GPS.**Timeout**

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):
   if not hasattr(timeout, "occur"):
       return True

   timeout.occur += 1
   print "A timeout occur=" + `timeout.occur`
   if timeout.occur == 3:
      timeout.remove()

t = GPS.Timeout(500, callback)
t.occur = 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, 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()

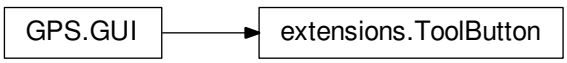
Unregisters a timeout.

15.5.69 GPS.ToolButton

class GPS.**ToolButton**

Represents a Button that can be placed in the Toolbar.

This class is provided for backwards compatibility only. Instead of using this, use GPS.Action and GPS.Action.button().



__init__(*stock_id*, *label*, *on_click*)

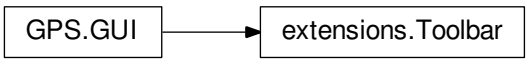
This class is provided for backwards compatibility only.

Instead of using this, use GPS.Action and GPS.Action.button().

15.5.70 GPS.Toolbar

class GPS.**Toolbar**
The main GPS Toolbar.

This class is provided for backwards compatibility only. Instead of using this, use GPS.Action and GPS.Action.button().



__init__()
This class is provided for backwards compatibility only.

Instead of using this, use GPS.Action and GPS.Action.button().

append(*widget*, *tooltip=''*)
Append a widget to the Toolbar.

The widget must be either a GPS.ToolButton or a GPS.Button.

get(*id*)
Does nothing and writes an error message

This function is to help detect compatibility errors in plugins.

get_by_pos(*pos*)
Does nothing and writes an error message

This function is to help detect compatibility errors in plugins.

insert(*widget*, *pos*, *tooltip*)
Append a widget to the Toolbar.

To insert a widget at a specific position in the toolbar, use GPS.Action.button instead, and set a value to the 'section' parameter.

15.5.71 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.

15.5.72 GPS.VCS

class GPS.**VCS**
General interface to version control systems.

static annotate (*file*)
Displays 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 example), and adds the corresponding information to the left of the editor.

Parameters

- **vcs_identifier** – A string
- **file** – A string
- **output** – A string

static commit (*file*)
Commits file.

Parameters **file** – A string

static diff_head (*file*)
Shows differences between the local copy of file and the head revision.

Parameters **file** – A string

static diff_working (*file*)
Shows differences between the local copy of file and the working revision.

Parameters **file** – A string

static get_current_vcs ()
Returns 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*)
Queries the status of file.

Parameters **file** – A string

static log (*file*, *revision*)
Gets 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*)
Removes the annotations for `file`.

Parameters **file** – A string

static repository_dir (*tag_name=''*)
Returns the repository root directory, (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` (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*)
Records a reference file (the file on which a diff buffer is based, for example) for `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". `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()
Shows the list of supported VCS systems.

Returns List of strings

static update(*file*)
Updates file.

Parameters **file** – A string

static update_parse(*vcs_identifier*, *string*, *dir=''*)
Parses 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

15.5.73 GPS.VCS2

class GPS.**VCS2**
An interface to a version control engine.

One project is always associated with at most one version control, which is used for all of its sources.

However, in a given tree of projects, there can be multiple such engines, if different repositories are used for the sources (for instance a third-party repository is imported), possibly for different implementations of version control (one for git, one for subversion...)

As a result, for a given source file you first need to find the relevant engine, via a call to *GPS.VCS2.get*.

For efficiency, GPS caches the status of files locally, and only refreshes at specific points. To get the status of a file as currently cached, use *GPS.VCS2.get_file_status*. This will always return a valid status, even if the cache has never been initialized by querying the actual VCS on the disk. To do this, call one of *GPS.VCS2.ensure_status_for_** methods. These methods will eventually run the *vcs_file_status_update* hook to let you know that the status has changed. This is all implemented asynchronously though, since such a query might take time.

This class provides the user view for VCS engines.

In practice, it is further derived in the code, to provide support for various VCS engines like git, CVS, subversion, clearcase,... The hierarchy is:

```
     GPS.VCS2
        |
     vcs2.VCS   (abstract)
        |
  +-----+-----+
  |     |     |
Git   CVS    Subversion
```

Actions = <GPS.__enum_proxy object at 0x7f089a108e90>

Branch = <class GPS.Branch at 0x7f089a124328>

Commit = <class GPS.Commit at 0x7f089a124390>

Status = <GPS.__enum_proxy object at 0x7f089a108ed0>

name
Return the name of the VCS (as could be set in the project's IDE.VCS_Kind attribute). This is always lower-cased.

Type str

static active_vcs()
Return the currently active VCS. When the project uses a single VCS, it will always be the same instance. But when the project tree has multiple VCS, or the same VCS but multiple working directories, this will return the instance selected by the user in the local toolbar of the VCS views.

Return type *GPS.VCS2*

ensure_status_for_all_source_files()
Ensure that all source files in any of the loaded project have a known status in self's cache. This doesn't ensure that the status for files that are under version control but not part of the project sources is also computed, although in most cases the VCS engine will indeed compute them.

This is computed asynhronously.

ensure_status_for_files(*files*)
Make sure that all files has a known status in self's cache. This is computed asynhronously.

Parameters **files** (*List[GPS.File]*) –

ensure_status_for_project(*project*)
Make sure that all source files of the project have a known status in self's cache. This is computed asynhronously.

Parameters **project** (*GPS.Project*) –

static get(*project*)
Return the VCS to use for the files in a given project. Each project can have its own VCS, if for instance it is imported from another repository.

Parameters **project** (*GPS.Project*) –

Return type *GPS.VCS2*

get_file_status(*file*)
Return the file status, as seen in self'cache.

Parameters **file** (*GPS.File*) –

Returntype a tuple (GPS.VCS2.Status, str, str) where the two strings are the version and the repository version

invalidate_status_cache()
Mark all entries in self's cache as no longer valid. This will force a refresh next time one of the *ensure_status_** method is called.

set_run_in_background(*background*)
Should be called to let GPS know when background commands are executing. This is used to queue commands instead of running several of them in parallel. Do not call this function directly. Instead, use the python function vcs2.core.run_in_background which provides a higher-level API for this purpose.

static supported_systems()
Shows the list of supported VCS systems.

Returns List of strings

static vcs_in_use()
Return the list of all VCS in use for the loaded project and its imported projects.

Return type [GPS.VCS2]

15.5.74 GPS.VCS2_Task_Visitor

class GPS.**VCS2_Task_Visitor**
A class used in *GPS.VCS2.async_fetch_history*. This is only used when writing your own support for VCS engines.

annotations(*file*, *first_line*, *ids*, *annotations*)
Report annotations to add to the side of the editors. Such annotations should provide author, last modification date, commit id,... for each line.

Parameters

- **file** (*GPS.File*) – the file for which we add annotations
- **first_line** (*int*) – the first line number for which we return information.
- **ids** (*List(str)*) – the commit ids, for each line.
- **annotations** (*List(str)*) – the annotations. The first entry is for *first_line*, then the next line, and so on.

branches(*category*, *iconname*, *can_rename*, *branches*)
Report a list of branches available for the current VCS.

Parameters

- **category** (*str*) – the name of the category, as displayed in the Branches view. You can call this method several times for the same category if need be. If the category is 'BRANCHES', it will be expanded in the GUI to show all the branches within.
- **iconname** (*str*) – icon to use for this category.
- **can_rename** (*bool*) – true if the branches can be renamed.
- **branches** (*List*) – a list of branches (see *GPS.VCS2.Branch*).

diff_computed(*diff*)
Used to report a diff, from *GPS.VCS2.async_diff*.

Parameters **diff** (*str*) – the diff, using standard diff format.

file_computed(*contents*)
Used to provide the contents of a file at a specific version.

Parameters **contents** (*str*) – the contents of the file.

history_lines(*list*)
Report when a new line for the VCS history was seen. Used from *GPS.VCS2.async_fetch_history*.

Parameters **list** (*List(GPS.VCS2.Commit)*) – a list of lines from the history. This doesn't have to be the whole log, though, although it is more efficient to send bigger chunks.

set_details(*id*, *header*, *message*)
Used to provide details on one specific commit, from the *GPS.VCS2.async_fetch_commit_details* method.

Parameters

- **id** (*str*) – the commit for which we are reporting details

- **header** (*str*) – a multi-string piece of information to display in the History view. This should show the commit id, the date and author of the commit,...
- **message** (*str*) – a multi-string description of the commit message, and possibly a diff of what has changed.

success (*msg=''*)
This should be called whenever an action succeed. It is used to perform various cleanups on the Ada side (for instance recomputing the status of files).

Parameters **msg** (*str*) – If specified, a temporary popup is displayed to the user showing the message. The popup automatically disappears after a short while.

tooltip (*text*)
Report additonal text to display in tooltips. In particular, this is called in the Branches view, as a result of calling the VCS engine's *async_action_on_branch* method.

Parameters **text** (*str*) – additional text for the tooltip

15.5.75 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 ()
Closes all editors involved in a visual diff.

static create (*file1*, *file2*, *file3=''*)
If none of the files given as parameter is already used in a visual diff, 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 ()
Returns the list of files used in a visual diff.

Returns A list of GPS.File

static get (*file1*, *file2=''*, *file3=''*)
Returns an instance of an already exisiting visual diff. If an instance already exists for this visual diff, it is returned. All files passed as parameters must be part of the visual diff but not all files of the visual diff must 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, is returned even if it is a two or three file visual diff.

Parameters

- **file1** – An instance of GPS.File
- **file2** – An instance of GPS.File
- **file3** – An instance of GPS.File

static list()

Returns the list of visual diffs 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()

Recomputes 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).

15.5.76 GPS.XMLViewer

class GPS.**XMLViewer**

This class represents Tree-based views for XML files.

__init__(*name*, *columns=3*, *parser=None*, *on_click=None*, *on_select=None*, *sorted=False*)

Creates a new XMLViewer, named 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 not specified, 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 called every time the user double-clicks 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, 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))
```

static create_metric(*name*)

Creates a new `XMLViewer` for an XML file generated by gnatmetric. `name` is the name for the window.

Parameters **name** (*string*) – A string

static get_existing(*name*)

Returns a `XMLViewer` instance if `name` corresponds to an existing XMLViewer window. If no `XMLViewer`'window has been found for the given ``name`, returns None instead.

Parameters **name** (*string*) – A string

parse (*filename*)
Replaces the contents of self by that of the XML file.

Parameters **filename** – An XML file

parse_string (*str*)
Replaces the contents of self by that of the XML string `str`.

Parameters **str** – A string

CHAPTER

SIXTEEN

SCRIPTING API REFERENCE FOR *GPS.BROWSERS*

Interface to the graph drawing API in GPS.

16.1 Classes

16.1.1 GPS.Browsers.AbstractItem

class GPS.Browsers.**AbstractItem**

This abstract class represents either items or links displayed in the canvas, and provide common behavior.

height

The height of the item (in its own coordinate space)

Type (read-write) int

is_link

Whether the item is a link.

Type (read-only) bool

parent

The parent item (to which self was added)

Type (read-only) GPS.Browsers.Item

style

The style applied to the item

Type (read-write) GPS.Browsers.Style

width

The width of the item (in its own coordinate space)

Type (read-write) int

x

The position of the item. For a toplevel item, this is the position within the diagram. For an item that was added to another item, this is the position within its parent.

Type (read-write) int

y

The position of the item. For a toplevel item, this is the position within the diagram. For an item that was added to another item, this is the position within its parent.

Type (read-write) int

hide()
Temporarily hide the item, until *GPS.Browsers.Item.show* is called.

show()
Show an item that has been hidden.

16.1.2 GPS.Browsers.Diagram

class GPS.Browsers.**Diagram**
A diagram contains a set of items.

You can extend this class with your own, and declare the following special subprograms which are called automatically by GPS:

```
def on_selection_changed(self, item, *args):
    '''
    Called when the selection status of item has changed.

    :param GPS.Browsers.Item item: the item that was selected or
      unselected. This is set to None when all items were
      unselected.
    '''
```

Selection = <class 'gps_utils.Enum NONE=0, MULTIPLE=2, SINGLE=1'>

items = None
The list of all GPS.Browsers.Item in the diagram. This only include toplevel items, you will need to iterate their own children if you need access to them. This also includes links.

selected = None
The list of selected GPS.Browsers.Item

__init__()
Creates a new empty diagram.

add(*item*)
Add a new item to the diagram. The coordinates of the item are set through item.set_position().

Parameters **item** (*GPS.Browsers.Item*) – the item to add

changed()
This method should be called whenever the contents of the diagram has changed, or some items have been modified. This will trigger a re-display of the diagram. Not needed when you only added a new item to the diagram.

clear()
Remove all items from the diagram.

clear_selection()
Unselect all items in the diagram.

is_selected(*item*)
Whether the item is selected.

Parameters **item** (*GPS.Browsers.Item*) – the item to check

Returns a boolean

links(*item*)
Return the incoming or outgoing links for item (i.e. all links for which item is a source or a target)

Parameters **item** (*GPS.Browsers.AbstractItem*) – the source of the links

Returns list of GPS.Browsers.AbstractItem

static load_json (*file*, *diagramFactory=None*)

Load a JSON file into a series of diagrams.

The format of the file is described below, and is basically a serialization of all the style and object attributes mentioned elsewhere in this documentation.

The JSON file should contain an object ("{...}") with the following attributes:

•*styles*: this is a dict of style objects. Each style is itself described as an object whose attributes are any of the valid parameters for GPS.Browsers.Style.__init__().

There are a few special ids that can be used to set the default properties or styles for the items defined in this json file. For instance, if the id starts with "__default_props_" followed by one of the valid item type, the object can define the default attribute for any of the valid attributes of items.

If the id starts with "__default_style_", the object defines the style properties for any of the item type.

For instance:

```
{"styles": {
   '__default_props_text': {"margin": [0, 5, 0, 5]},
   '__default_props_rect': {"minWidth": 200},
   '__default_style_text': {"fontName": "arial 10"},
   'customStyle1': {"stroke": "blue"}
}}
```

•*templates*: this is a dictionary of diagram objects (as described below. They are not created or inserted in the diagram. However, if a later description inside *diagrams* contains an attribute 'template', then that attribute will be replaced by all the attributes defined in the template. For instance:

```
'templates': {
    '#tmpl0': { 'type': 'rect'; 'width': 10 }
},
'diagrams': [
    { 'template': '#tmp10',
      'height': 20
    }
]
```

is equivalent to putting the type and the width inline in the objet definition, but provide better sharing.

•*diagrams*: this is a list of diagram object. A single file can contain multiple diagrams, but a browser window always only displays a single diagram.

Each diagram object has two possible attributes:

–*items*: this is a list of item objects (see below)

–*links*: this is a list of link objects (see below)

An item object is itself described with a JSON object, with the following possible attributes:

•*id*: an optional string, which defines an id for an object. This id can be used when creating links. It is also stored as an id attribute in the instance of GPS.Browsers.Item that is created.

- *data*: this field is stored as is in the generated instance of GPS.Browsers.Item. It can be used to store any application specific data, in particular since the instance will be passed to the callbacks to handle click events.

- *x*, *y*, *anchorx*, *anchory*: optional float attributes. See the description of `GPS.Browsers.Item.set_position()`.

- *width*, *height*: the size of the item. This can be specified as a positive float to specify a size in pixels. Alternatively it can be set to *GPS.Browsers.Item.Size.FIT* (-1) so that the item and its margins occupy the full width of its parent (for a vertical layout) or the full height (for a horizontal layout). It can also be set to *GPS.Browsers.Item.Size.AUTO* (-2) so that the item occupies the minimal size needed for all of its children. The default is FIT.

- *style*: an optional string (which then references the id of one of the styles defined in the "styles" attribute described above, or an inline JSON object that describes a style as above.

- *type*: the type of the item. Valid values are "rect", "hr", "ellipse", "polyline" or "text" (see the corresponding classes in this documentation). This attribute is optional, and will be guessed automatically in some cases. For instance, when the object also has a *text* attribute, it is considered as a text item. If it has a "points" attribute it is considered as a polyline item. The default *type* is "rect".

- *minWidth* and *minHeight*: the minimal size for an item. See `GPS.Browsers.Item.set_min_size()`.

- *vbox* and *hbox*: list of items, which are the children of the current item. These children are organized either vertically or horizontally (only one of the two attributes can be specified, vbox takes precedence).

When an item is created as a child of another one (in its parent's *vbox* or *hbox*), it may have the following additional attributes. See `GPS.Browsers.Item.add()` for more information.

- *margin*: the margins around the item. This is either a single float (in which case all margins are equal), or a list of four floats which give the top, right, bottom and left margins respectively.

- *align*: one of the values from `GPS.Browsers.Item.Align`. In a JSON file, though, you can only use the corresponding integer values.

- *float*: whether the item is set as floating

- *overflow*: one of the integer values for `GPS.Browers.Item.Overflow`

Text objects (corresponding to `GPS.Browsers.TextItem`) have the following additional attributes:

- *text*: the text to display.

- *directed*: one of the values from `GPS.Browsers.TextItem.Text_Arrow`, which indicates that an extra arrow should be displayed next to the text. This is mostly relevant when the text is used as a label on a link, in which case the actual arrow will be computed automatically from the orientation of the link.

Hr objects (corresponding to `GPS.Browsers.HrItem`) have the following additional attributes:

- *text*: the text to display.

Polyline objects (corresponding to `GPS.Browsers.PolylineItem`) have the following additional attributes:

- *points*: the points that describe the contour of the object, as a list of floats. They are grouped into pairs, each of which describes the coordinates of a point.

- *close*: whether the last point should automatically be linked to the first.

- *relative*: an optional boolean (defaults to false) that indicates whether the points are coordinates relative to the item's topleft corner, or are relative to the previous point.

Rect items (corresponding to GPS.Browsers.RectItem) have the following additional attributes:

- *radius*: optional float indicating the radius for the corners of the rectangle.

A link object contains the following attributes (see GPS.Browser.Link for more information on the parameters):

- *id*: an optional string id for the link, if you need to create links to it.
- *from* and *to*: these are record with one mandatory field, *ref*, which is the id of one of the objects or links created previously. In addition, these records can also specify the following attributes:
  - *anchorx*, *anchory*: floats in the range 0.0 .. 1.0 which specify where the item is attached in its target item
  - *side* takes its value from GPS.Browsers.Link.Side, and is used to force the link to emerge from one specific side of the item.
  - *label* is a JSON object describing an item, as described earlier. This will generally be a text item.
- *label*: one of GPS.Browsers.Item, to display in the middle of the link. This will generally be a text item. If it is directed, the arrow will be computed from the orientation of the link.
- *route*: one of GPS.Browsers.Link.Routing (as integer) to indicate how the link is displayed.
- *waypoints*: a list of floats, which will be grouped into pairs to define waypoints. See GPS.Browsers.Link.set_waypoints(). An alternative definition is to use an object with two fields, *points* which is the list of floats, and *relative* which is a bool indicating whether the points are in absolute coordinates or relative to the previous point.

Here is an example which draws two items linked together:

```
{"styles": {},
 "diagrams": [
   {"items": [
      {"x": 0, "y": 0, "style": "customStyle1",
       "id": "first item",
       "width": 100, "height": 100,
       "vbox": [
          {"text": "Name",
           "style": {"fontName": "arial 20", stroke:null}},
          {"type": "hr", "text":"attributes"},
          {"text": "+attr:integer"}
       ]
      },
      {"x": 100, "y":200",
       "id": "second item",
       "text": "Annotation"},
    ],
    "links": [
      {"from": {"ref": "first item"},
       "to":   {"ref": "second item"},
       "style": {"stroke": "blue", "dashes": [4, 4]}}
    ]
   }
]
}
```

Parameters

- **file** (*str*) – an object that has a read() function, or the name of a file as a string.

- **diagramFactory** (*callable*) – a callback that creates a new instance of GPS.Browsers.Diagram or one of its derived classes. Typically, you could pass the class itself, since calling it will create a new instance. As a special case, the diagrams returned by this function include a special *ids* field, which is a dict mapping the id to the actual item.

Returns the list of GPS.Browsers.Diagram objects created.

static load_json_data(*data*, *diagramFactory=None*)
Load a JSON description and display the corresponding data. See GPS.Browsers.Diagram.load_json() for more information on the format of the JSON file.

lower_item(*item*)
Lower the item, so that it is displayed below all other items.

Parameters **item** (*GPS.Browsers.Item*) – the item to lower.

raise_item(*item*)
Raise the item, so that it is displayed above all other items.

Parameters **item** (*GPS.Browsers.Item*) – the item to raise.

remove(*item*)
Remove an item from the diagram.

Parameters **item** (*GPS.Browsers.Item*) – the item to remove.

select(*item*)
Select the item. If the diagram is set up for single selection, any previously selected item is first unselected.

Parameters **item** (*GPS.Browsers.Item*) – the item to select.

set_selection_mode(*mode='GPS.Browsers.Diagram.Selection.SINGLE'*)
Controls the selection in the views associated with this buffer. It can be used to indicate whether a single item or multiple items can be selected.

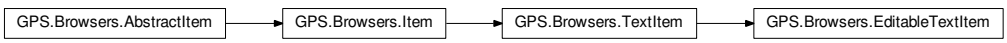
Parameters **mode** (*GPS.Browsers.Diagram.Selection*) – the type of selection

unselect(*item*)
Unselect the item.

Parameters **item** (*GPS.Browsers.Item*) – the item to select.

16.1.3 GPS.Browsers.EditableTextItem

class GPS.Browsers.**EditableTextItem**
A text item (GPS.Browsers.TextItem) that can be double-clicked on to edit its text. See GPS.Browsers.View.start_editing() if you need to start the editing from the script, instead of having the user double-click on the item.



editable = True
Whether this item is currently editable by the user

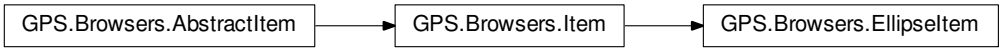
__init__ (*style*, *text*, *directed='GPS.Browsers.TextItem.Text_Arrow.NONE'*, *on_edited=None*)
Creates a new editable text item

Parameters

- **style** (*GPS.Browsers.Style*) – how to draw the item
- **text** (*str*) – the text to display.
- **directed** (*bool*) – whether to draw an additional arrow next to the text. This can be used for instance when the text is next to a link, and indicates in which direction the text applies.
- **on_edited** – A callback whenever the text has been modified interactively by the user. The profile is *on_edited(textitem, old_text)*

16.1.4 GPS.Browsers.EllipseItem

class GPS.Browsers.**EllipseItem**
An item which displays an ellipse or a circle. It can contain children.



__init__ (*style*, *width=-1.0*, *height=-1.0*)
Create a new ellipse/circle item, inscribed in the box given by the width and height.

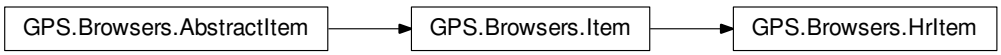
Parameters

- **style** (*GPS.Browsers.Style*) – how to draw the item
- **width** (*float*) – used to force a specific width for the item. If negative or null, the width is computed from the children of the item.
- **height** (*float*) – similar to width.

16.1.5 GPS.Browsers.HrItem

class GPS.Browsers.**HrItem**
A horizontal-line item, with optional text in the middle. This is basically represented as:

—— text ——



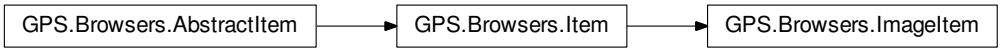
__init__ (*style*, *text=''*)
Creates a new horizontal line.

Parameters

- **style** (*GPS.Browsers.Style*) – how to draw the item
- **text** (*str*) – the text to display.

16.1.6 GPS.Browsers.ImageItem

class GPS.Browsers.**ImageItem**
An item that shows an image.



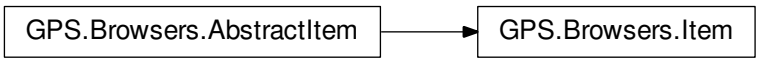
__init__ (*style*, *filename*, *width=-1.0*, *height=1.0*)
Creates a new image item.

Parameters

- **style** (*GPS.Browsers.Style*) – how to draw the border of the item
- **filename** (*str*) – the filename to load.
- **width** (*float*) – forcing a size for the item (the default is to use the image's own size)
- **height** (*float*) – forcing a size for the item (the default is to use the image's own size)

16.1.7 GPS.Browsers.Item

class GPS.Browsers.**Item**
This abstract class represents any of the items that can be displayed in a browser. Such items have an outline, whose form depends on the type of the item (rectangular, polygone,...) They can almost all contain children. Unless you specified an explicit size for the item, its size will be computed to include all the children. The children are stacked either vertically or horizontally within their container, so that one child appears by default immediately below or to the right of the previous one. Extra margins can be specified to force extra space.



Align = <class 'gps_utils.Enum START=0, END=2, MIDDLE=1'>

Layout = <class 'gps_utils.Enum HORIZONTAL=0, VERTICAL=1'>

Overflow = <class 'gps_utils.Enum PREVENT=0, HIDE=1'>

Size = <class 'gps_utils.Enum AUTO=-2, FIT=-1'>
Describes the size of an item. In general, the size is given as a number of pixels. There are however a few special values.

•FIT indicates that the item is sized so that it fits exactly in its parent container, including the child margins. So for instance given a parent with a vertical layout, of width 200px, and a child with 10px margins both on left and right, then the child's width will be set to 180px.

•AUTO indicates that the item's size is computed so that all of its children fit exactly inside the item.

children = None
The list of GPS.Browsers.Item that were added to the item. This property is not writable.

__init__()
Will raise an exception, this is an abstract class.

add(*item*, *align='GPS.Browsers.Item.Align.START'*, *margin=(0, 0, 0, 0)*, *float=False*, *overflow='GPS.Browsers.Item.Overflow.PREVENT'*)
Add a child item. This child will be displayed as part of the item, and will move with it. The size of the child will impact the size of its parent, unless you have forced a specific size for the latter.

Parameters

- **item** (*GPS.Browsers.Item*) – the item to add
- **align** (*GPS.Browsers.Item.Align*) – How the item should be aligned within its parent. When the size of the child is computed automatically, it will have the same width as its parent (for vertical layout) or the same height as its parent (for horizontal layout), minus the margins. In this case, the align parameter will play no role. But when the child is smaller than its parent, the align parameter indicates on which side an extra margin is added.
- **margin** (*list_of_float*) – Extra margin to each side of the item (resp. top, right, bottom and left margin). These a float values.
- **float** (*bool*) – Whether the child should be floating within its parent. This impacts the layout: by default, in a vertical layout, all children are put below one another, so that they do not overlap. However, when a child is floating, it will be put at the current y coordinate, but the next item will be put at the same coordinate as if the child was not there.
- **overflow** (*GPS.Browsers.Item.Overflow*) – Whether the child's size should impact the parent size. For instance, you might want to display as much text as possible in a box. When overflow is set to PREVENT, the box will be made as large as needed to have the whole text visible (unless you have specified an explicit size for the box, as usual). But when the overflow is set to HIDE, the box will get its size from the other children, and the text will simply be ellipsized if it does not fit in the box.

get_parent_with_id()
Returns either self (if it has an "id" attribute), or its first parent that does. It might return None if nothing is found. Such "id" attributes are in general used to identify items with semantic information, rather than just display purposes.

recurse()
A generator that returns self and all its child items. For instance:

for it in item.recurse(): ...

set_child_layout(*layout='GPS.Browsers.Item.Layout.VERTICAL'*)
Choose how children are organized within this item.

Parameters **layout** (*GPS.Browsers.Item.Layout*) – if set to VERTICAL, then the child items are put below one another, otherwise they are put next to one another.

set_height_range(*min=-1.0*, *max=-1.0*)
Constrain the range of height for self. Self could be make larger if its children request a larger size, but will not smaller than the given size. The default is for items to use the full width of their parent (for vertical

layout) or the full height (for horizontal layout), and the size required by their children for the other axis. This overrides any previous call to set_size

Parameters

- **min** (*float*) – minimal height
- **max** (*float*) – maximal height

set_position (*x=None*, *y=None*, *anchorx=0.0*, *anchory=0.0*)

Indicates the position of the item. This is the position within its parent item, or if there is no parent this is the absolute position of the item within the diagram. Calling this function should always be done for toplevel items, but is optional for children, since their position is computed automatically by their container (which is especially useful with text items, whose size might be hard to compute).

Parameters

- **x** (*float*) – coordinates relative to parent or browser.
- **y** (*float*) – coordinates relative to parent or browser.
- **anchorx** (*float*) – what position within the item x is refering to. If set to 0.0, x indicates the left side of the item. If set to 0.5, x indicates the middle of the item. 1.0 indicates the right side of the item.
- **anchory** (*float*) – what position within the item y is refering to.

set_size (*width=-1.0*, *height=-1.0*)

Forces a specific size for self. This overrides any previous call to set_width_range and set_height_range.

Parameters

- **width** (*float*) – actual width. If -1, the width will be computed automatically.
- **height** (*float*) – actual height. If -1, the height will be computed automatically.

set_width_range (*min=-1.0*, *max=-1.0*)

Constrain the range of width for self. Self could be make larger if its children request a larger size, but will not smaller than the given size. The default is for items to use the full width of their parent (for vertical layout) or the full height (for horizontal layout), and the size required by their children for the other axis. This overrides any previous call to set_size

Parameters

- **min** (*float*) – minimal width
- **max** (*float*) – maximal width

toplevel ()

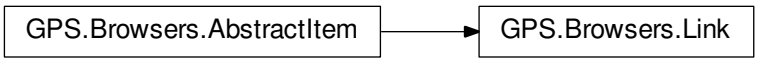
The item that contains self, or self itself if it is already a toplevel item. This is simply computed by using the `GPS.Browsers.AbstractItem.parent()` property.

Returns a `GPS.Browsers.AbstractItem`

16.1.8 GPS.Browsers.Link

class `GPS.Browsers.`**Link**

A line between two items. When the items are moved, the line is automatically adjusted to stay connected to those items.



Routing = <class 'gps_utils.Enum STRAIGHT=1, ARC=2, CURVE=3, ORTHOGONAL=0'>

Side = <class 'gps_utils.Enum RIGHT=2, BOTTOM=3, AUTO=0, TOP=1, NO_CLIP=5, LEFT=4'>

fromLabel = None
Returns a *GPS.Browsers.Item* element (or None) corresponding to the link's source label.

label = None
Returns a *GPS.Browsers.Item* element (or None) corresponding to the link's label.

source = None
The source *GPS.Browsers.Item* for the link

target = None
The target *GPS.Browsers.Item* for the link

toLabel = None
Returns a *GPS.Browsers.Item* element (or None) corresponding to the link's target label.

__init__(*origin*, *to*, *style*, *routing='GPS.Browsers.Link.Routing.STRAIGHT'*, *label=None*, *fromX=0.5*, *fromY=0.5*, *fromSide='GPS.Browsers.Link.Side.AUTO'*, *fromLabel=None*, *toX=0.5*, *toY=0.5*, *toSide='GPS.Browsers.Link.Side.AUTO'*, *toLabel=None*)
Creates a new link attached to the two items FROM and TO.

Parameters

- **from** (*GPS.Browsers.Item*) – the origin of the link.
- **to** (*GPS.Browsers.Item*) – the target of the link.
- **style** (*GPS.Browsers.Style*) – how to draw the item
- **routing** (*GPS.Browsers.Link.Routing*) – the routing algorithm to use to compute how the link is displayed. A STRAIGHT link takes the shortest route between the two items, whereas an ORTHOGONAL link only uses horizontal and vertical lines to do so. A CURVE is almost the same as a straight link, but is slightly curves, which is useful when several links between the same two items exist. An ORTHOCURVE link uses bezier curves to link the two items.
- **fromX** (*float*) – the position within the origin where the link is attached. This is a float in the range 0.0 .. 1.0. The link itself is not displayed on top of the origin box, but changing the attachment point will change the point at which the link exits from the origin box.
- **fromY** (*float*) – similar to fromY, for the vertical axis
- **fromSide** (*GPS.Browsers.Link.Side*) – This can be used to force the link to exist from a specific side of its toplevel container. For instance, if you set fromX to 0.0, the link will always exit from the left side of self. But if self itself is contained within another item, it is possible that the line from the left side of self to the target of the link will in fact exit from some other place in the container item. Setting fromSide to GPS.Browsers.Link.Side.LEFT will make sure the link exits from the left side of the parent item too.

- **label** (*GPS.Browsers.Item*) – a label (in general a TextItem) to display in the middle of the link.
- **fromLabel** (*GPS.Browsers.Item*) – a label (in general a TextItem) to display next to the origin of the link.
- **toX** (*float*) – This plays a similar role to fromX, but it can also have a negative value. In such a case, GPS will try to move the end position of the link along the border of the item, so as to get a horizontal or vertical segment. If this is not possible, the usual behavior applies to abs(toX) to find the attachment of the link in the item.
- **toY** (*float*) – similar to toX
- **toLabel** (*GPS.Browsers.Item*) – a label (in general a TextItem) to display next to the target of the link.

recurse()
Returns self and its labels. This matches *GPS.Browsers.Item.recurse*. Note that a label doesn't have self as its parent, so you cannot get back to the link from one of its labels.

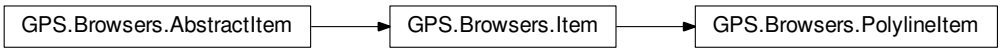
set_waypoints(*points*, *relative=False*)
Force specific waypoints for link. The lines will pass through each of these points when it is routed as straight or orthogonal lines. By default, the coordinates are absolute (in the same coordinate system as the items). If however you specify relative coordinates, then each point is relative to the previous one (and the first one is relative to the link's attachment point on its origin item).

Parameters

- **points** (*list_of_float*) – The floats are grouped into pair, each of which describes the coordinates of one points.
- **relative** (*bool*) – whether the coordinates are relative to the previous point, or absolte.

16.1.9 GPS.Browsers.PolylineItem

class GPS.Browsers.**PolylineItem**
An item which displays a set of connected lines. It can be used to draw polygons for instance, or simple lines.



__init__(*style*, *points*, *close=False*)
Create a new polyilne item.

Parameters

- **style** (*GPS.Browsers.Style*) – how to draw the item
- **points** (*list_of_float*) – each points at which line ends. The coordinates are relative to the top-left corner of the item (so that (0,0) is the top-left corner itself). The floats are grouped into pair, each of which describes the coordinates of one points. For instance:

  (0,0, 100,100)

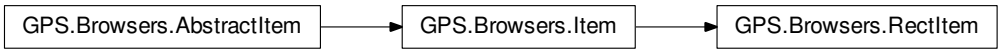
  displays a single line which goes from the top-left corner to a point at coordinates (100,100).

- **close** (*bool*) – whether the last point should automatically be linked to the first. This ensures the polygone is properly closed and can be filled.

16.1.10 GPS.Browsers.RectItem

class GPS.Browsers.**RectItem**

A special kind of item, which displays a rectangular box, optionally with rounded corner. It can contain children items.



__init__ (*style*, *width=-1.0*, *height=-1.0*, *radius=0.0*)

Creates a new rectangular item.

Parameters

- **style** (*GPS.Browsers.Style*) – how to draw the item
- **width** (*float*) – used to force a specific width for the item. If this is null or negative, the width will be computed from the children of the item.
- **height** (*float*) – used to force a specific height, which will be computed automatically if height is negative or null
- **radius** (*float*) – the radius for the angles.

16.1.11 GPS.Browsers.Style

class GPS.Browsers.**Style**

This class provides multiple drawing attributes that are used when drawing on a browser. This includes colors, dash patterns, arrows and symbols,... Very often, such a style will be reused for multiple objects.

Align = <class 'gps_utils.Enum MIDDLE=1, RIGHT=2, LEFT=0'>

Arrow = <class 'gps_utils.Enum SOLID=2, NONE=0, DIAMOND=3, OPEN=1'>

Symbol = <class 'gps_utils.Enum STRIKE=2, DOUBLE_STRIKE=3, NONE=0, CROSS=1'>

Underline = <class 'gps_utils.Enum DOUBLE=2, NONE=0, SINGLE=1, LOW=3'>

__init__ (*stroke='black'*, *fill=''*, *lineWidth=1.0*, *dashes=[]*, *sloppy=False*, *fontName='sans 9'*, *fontUnderline='GPS.Browsers.Style.Underline.NONE'*, *fontStrike=False*, *fontColor='black'*, *fontLineSpacing=0*, *fontHalign='GPS.Browsers.Style.Align.LEFT'*, *arrowFrom='GPS.Browsers.Style.Arrow.NONE'*, *arrowFromLength=8.0*, *arrowFromAngle=0.4*, *arrowFromStroke='black'*, *arrowFromFill=''*, *arrowFromWidth=1.0*, *arrowTo='GPS.Browsers.Style.Arrow.NONE'*, *arrowToLength=8.0*, *arrowToAngle=0.4*, *arrowToStroke='black'*, *arrowToFill=''*, *arrowToWidth=1.0*, *symbolFrom='GPS.Browsers.Style.Symbol.NONE'*, *symbolFromStroke='black'*, *symbolFromDist=16.0*, *symbolFromWidth=1.0*, *symbolTo='GPS.Browsers.Style.Symbol.NONE'*, *symbolToStroke='black'*, *symbolToDist=16.0*, *symbolToWidth=1.0*, *shadowColor=None*, *shadowOffsetX=2.0*, *shadowOffsetY=2.0*)

Constructs a new style. This function takes a very large number of parameters, but they are all optional. Not all of them apply to all objects either.

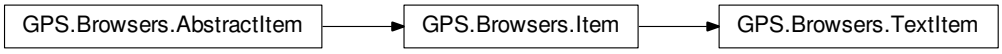
Parameters

- **stroke** (*str*) – the color to use to draw lines. The format is one of "rgb(255,255,255)", "rgba(255,255,255,1.0)" or "#123456". Transparency is supported when using rgba.
- **fill** (*str*) – how closed objects should be filled. The format is either a color, as for stroke, or a string describing a gradient as in *"linear x0 y0 x1 y1 offset1 color1 offset2 color2 ..."*

  where (x0,y0) and (x1,y1) define the orientation of the gradient. It is recommended that they are defined in the range 0..1, since the gradient will be automatically transformed to extend to the whole size of the object. The rest of the parameters are used to define each color the gradient is going through, and the offset (in the range 0..1) where the color must occur. For instance *"linear 0 0 1 1 0 rgba(255,0,0,0.2) 1 rgba(0,255,0,0.2)"*
- **lineWidth** (*float*) – the width of a line
- **dashes** (*list_of_floats*) – a dash pattern. The first number is the number of pixels which should get ink, then the number of transparent pixels, then again the number of inked pixels,... The pattern will automatically repeat itself.
- **sloppy** (*boolean*) – when true, no straight line is displayed. They are instead approximated with curves. Combined with a hand-drawing font, this makes the display look as if it has been hand-drawn.
- **fontName** (*str*) – the font to use and its size.
- **fontUnderline** (*GPS.Browsers.Style.Underline*) – The underline to use for the text
- **fontStrike** (*boolean*) – whether to strikethrough the text.
- **fontColor** (*str*) – the color to use for text.
- **fontLineSpacing** (*int*) – extra number of pixels to insert between each lines of text.
- **fontHalign** (*GPS.Browsers.Style.Align*) – How text should be aligned horizontally within its bounding box.
- **arrowFrom** (*GPS.Browsers.Style.Arrow*) – How should arrows be displayed on the origin of a line.
- **arrowFromLength** (*float*) – the size of an arrow.
- **arrowFromAngle** (*float*) – the angle for an arrow.
- **arrowFromStroke** (*str*) – the stroke color for the arrow.
- **arrowFromFill** (*str*) – the fill pattern for the arrow.
- **arrowTo** (*GPS.Browsers.Style.Arrow*) – same as arrowFrom, but for the end of a line
- **arrowToLength** (*float*) – similar to arrowFromLength
- **arrowToAngle** (*float*) – similar to arrowFromAngle
- **arrowToStroke** (*str*) – similar to arrowFromStroke
- **arrowToFill** (*str*) – similar to arrowFromFill
- **symbolFrom** (*GPS.Browsers.Style.Symbol*) – the extra symbol to display near the origin of a line.
- **symbolFromStroke** (*str*) – the stroke color to use for the symbol.
- **symbolFromDist** (*float*) – the distance from the end of the line at which the symbol should be displayed.
- **shadowColor** (*str*) – the color to use for shadows

16.1.12 GPS.Browsers.TextItem

class GPS.Browsers.**TextItem**
An item that displays text (optionaly within a rectangular box).



TextArrow = <class 'gps_utils.Enum DOWN=2, NONE=0, RIGHT=4, UP=1, LEFT=3'>

text = None
The text to display. This can be modified as needed, but you then need to call *GPS.Browsers.Diagram.changed* to let GPS know of the change.

__init__ (*style*, *text*, *directed='GPS.Browsers.TextItem.Text_Arrow.NONE'*)
Creates a new text item

Parameters

- **style** (*GPS.Browsers.Style*) – how to draw the item
- **text** (*str*) – the text to display.
- **directed** (*bool*) – whether to draw an additional arrow next to the text. This can be used for instance when the text is next to a link, and indicates in which direction the text applies.

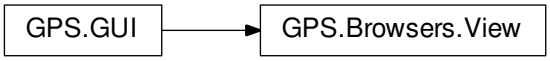
16.1.13 GPS.Browsers.View

class GPS.Browsers.**View**
A view shows a part of a diagram and its objects. Multiple views can be associated with the same diagram.

Although this class can be instantiated as is, it is most useful, in general, to extend it, in particular by adding a method on_item_clicked:

```
class My_View(GPS.Browsers.View):

    def on_item_double_clicked(self, topitem, item, x, y, *args):
        '''
        Called when the user double clicks on a specific item.
        It is recommended to add "*args" in the list of parameters,
        since new parameters might be added in the future.

        :param GPS.Browsers.Item topitem: the toplevel item clicked on
        :param GPS.Browsers.Item item: the item clicked on.
           When the item was created from a JSON file (see
           :func:`GPS.Browsers.Diagram.load_json`), it contains
           additional fields like `data` and `id` that were extracted
           from JSON.
           This item is either topitem or a child of it.
        :param float x: the coordinates of the mouse within the item.
        :param float y: the coordinates of the mouse within the item.
        '''
```

```
    def on_item_clicked(self, topitem, item, x, y, *args):
        '''
        Called when the user releases the mouse button on a specific
        item.
        This method is called twice for a double-click (once per
        actual click).
        The parameters are the same as above.
        '''

    def on_create_context(self, context, topitem, item, x, y, *args):
        '''
        Called when the user has right-clicked on an item.
        This function should prepare the context for contextual menus,
        although it does not directly add contextual menu entries.
        Instead, declare those menus as usual with
        :class:`GPS.Contextual` or :func:`gps_utils.make_interactive`.

        :param GPS.Context context: the context.
           The function should add custom fields to this context.
           These fields can then be tested for the filter or the
           action associated with a :class:`GPS.Contextual`.
        '''

    def on_key(self, topitem, item, key, *args):
        '''
        Called when the user presses a keyboard key while the view has
        the focus.

        :param GPS.Browsers.Item topitem: the toplevel item clicked on
        :param GPS.Browsers.Item item: the specific item clicked on.
        :param int key: the key that was pressed (see Gdk.KEY_*
           constants)
        '''
```



Background = <class 'gps_utils.Enum COLOR=1, DOTS=3, NONE=0, GRID=2'>

diagram = None
The GPS.Browsers.Diagram that the view is referring to. This is a writable property.

editing_in_progress = False
A read-only property that indicates whether the user is currently interactively modifying the contents of an item (for instance an EditableTextItem). See GPS.Browsers.View.start_editing()

scale = 1.0
The scaling factor for the view. This is a writable property

topleft = (0.0, 0.0)
The coordinates of the top-left corner of the view. This is a writable property

__init__()
Creates the python class instance, but does not associate it with any window yet. Before any of the other functions can be called, you first need to call self.create() to create the actual window. This creation is done in two steps, so that you can create your own class extending view. This is in particular needed to provide support for user interaction.

animate_item_position(*item*, *x*, *y*, *duration=0.4*)
Moves an item to a new position. This is animated, i.e. the item will go through various steps before reaching that position.

Parameters

- **item** (*GPS.Browsers.Item*) – the item to move
- **x** (*float*) – coordinates relative to parent or browser.
- **y** (*float*) – coordinates relative to parent or browser.
- **duration** (*float*) – how long the animation should take

cancel_editing()
Cancel any interactive editing that might be taking place

center_on(*point*, *xpos=0.5*, *ypos=0.5*)
Scrolls the view so that the point is at the given position within the view (in the middle if xpos and ypos are both 0.5, or to the left if xpos is 0.0, or the right if xpos is 1.0, and so on).

Parameters **point** (*tuple*) – the point

create(*diagram*, *title*, *save_desktop=None*, *snap_to_grid=True*, *snap_to_guides=False*, *toolbar='Browser'*)
Creates a new view that shows the given diagram. This view is automatically made visible in GPS.

Parameters

- **diagram** (*GPS.Browsers.Diagram*) – the diagram to display.
- **title** (*str*) – the title used for the notebook tab.
- **save_desktop** (*func(child)*) – An optional callback when GPS is saving the desktop. It is passed the MDIWindow in which the view was put, and should return a string to store in the desktop (see `GPS.MDI.add()` for more information). This is generally not useful, unless you are writting a custom python module, in which case the parameter should be set to "module._save_desktop". See the documentation for `modules.py` for more information.
- **snap_to_grid** (*bool*) – whether items should preferably align on the grid when they are moved (ie they might be moved an additional small amount so that they are properly aligned). Snapping is systematically disabled when the user moves the item while pressing shift.
- **snap_to_guides** (*bool*) – whether items should preferably align on smart guides. These guides are generated for particular points of interests of the other items (typically the top, middle and bottom of the bounding box). This feature is useful to help user align items.
- **toolbar** (*str*) – The name of the toolbar definition to use for the local toolbar of the browser. Such definitions are found in the file share/gps/menus.xml in the GPS install. If the name is not defined in that file, a new definition will be used that extends the standard browsers toolbar. You can later add custom actions to the toolbar by using `GPS.Action.button()`

export_pdf (*filename*, *format='a4'*, *visible_only=True*)
Creates a PDF file with the contents of the view.

Parameters

- **filename** (*GPS.File*) – the name of the file to create or override.
- **format** (*str*) – one of "a4", "a4_portrait", "a4_landscape", or similar variants with "a3" and "letter". It can also be a string "width,height" where the size is given in inches.
- **visible_only** (*bool*) – if True, the output will match was is visible in the view. If False, the output will include the whole contents of the diagram.

scale_to_fit (*max_scale=4.0*)
Scale and scroll the view so that all the items in the model are visible.

Parameters **max_scale** (*float*) – maximum scaling factor to allow.

scroll_into_view (*item*, *duration=0.0*)
Scrolls the view as little as possible so that item becomes visible.

Parameters

- **item** (*GPS.Browsers.Item*) – the item to show
- **duration** (*float*) – if not null, scrolling will be animated over that period of time.

set_background (*type*, *style=None*, *size=20.0*)
Set the type of background to display in the view.

Parameters

- **type** (*GPS.Browsers.View.Background*) – the type of background to display.
- **style** (*GPS.Browsers.Style*) – the style to use for the drawing. Depending on the style, the stroke or the fill should be set (or perhaps both). For instance the COLOR background uses the fill pattern (color or gradient). If you are using GRID, the fill should be unset, but the stroke should be set. When using a gradient, it is not resized to the size of the view (as opposed to what is done for items for instance), so it should be something like *linear 0 0 1000 1000 0 black 1 yellow*
- **size** (*float*) – the size of the grid, when using GRID or DOTS.

set_read_only (*readonly=True*)
A read-only view does not allow users to move items. It is still possible to click on items though, or zoom and scroll the view.

Parameters **readonly** (*bool*) – whether the view is read-only.

set_selection_style (*style*)
The style used to highlight the selected items (see `GPS.Browsers.Diagram.set_selection_mode()`).

Parameters **style** (*GPS.Browsers.Style*) – the style to use.

start_editing (*item*)
If *item* is editable, start interactive editing, as if the user had clicked on it.

CHAPTER

SEVENTEEN

USEFUL PLUGINS

17.1 User plugins

GPS comes with a number of plugins, some of which are activated by default. Control which plugins are activated by using the *Edit* → *Preferences* → *Plugins* menu.

This section discusses a few of the many plugins built in to GPS. The preferences dialog shows their description, so you can decide whether you want them enabled.

17.1.1 The `auto_highlight_occurrences.py` module

This plugin 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 plugin 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 plugin 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.

17.1.2 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.

17.2 Helper plugins

A number of plugins are useful when you want to create your own plugins.

17.2.1 The gps_utils module

class gps_utils.**Chainmap**(**maps*)
Combine multiple mappings for sequential lookup.

For example, to emulate Python's normal lookup sequence:

import __builtin__ pylookup = Chainmap(locals(), globals(), vars(__builtin__))

gps_utils.**execute_for_all_cursors**(*ed*, *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.**get_gnat_driver_cmd**()
Return the name of the GNAT driver that is suitable for the current project's target. For instance: "gnat" for native targets or "powerpc-elf-gnat" for cross PowerPC ELF targets.

class gps_utils.**hook**(*hook*, *last=True*)
A decorator that makes it easier to connect to hooks:

```
@hook("gps_started")
def my_function(*args, **kwargs):
    pass
```

Note that the function does not receive the hook as the first parameter. The function should however accept any number of parameters, for future extensions, since some hooks might receive extra arguments.

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**(**args*, ***kwargs*)
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=''*, *contextual_ref=''*, *icon=''*, *description=''*, *toolbar=''*, *toolbar_section=''*, *button_label=''*, *key_exclusive=True*)

Declare a new GPS action (an interactive function, in Emacs talk), associated with an optional menu and default key.

Parameters

- **callback** – This is the code that gets executed when the user executes the action. Such an action is executed via a menu, a toolbar button, a key shortcut, or by entering its name in the omnisearch.

  The callback can be one of:

  – a standard 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).

  – 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.

  – a generator function. Those are functions that use the yield keyword to temporarily suspend and give back control to the caller. These are convenient when chaining background tasks. See the workflows/promises.py package for more on generators and workflows

- **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*.
- **icon** (*str*) – Name of the icon to use for this action, for instance in toolbars or in various dialogs. This is the name of a file (minus extension) found in the icons directory of GPS.
- **contextual** – Path for the contextual menu This is either a string, for instance '/Menu/Submenu' or '/Menu/Submenu %f', which supports a number of parameter substitution; or a function that receives a GPS.Context as parameter and returns a string.
- **name** (*str*) – The name for the action. The default is to use the callback's name.
- **key_exclusive** (*bool*) – Only applies when a key is specified. If true, the key will no longer execute any action it was previously bound to. If false, the key will be bound to multiple actions.
- **description** (*str*) – the description for this action, as visible to the user. If not specified, the callback's own documentation will be used.
- **toolbar** (*str*) – If specified, inserts a button in the corresponding toolbar (either 'main' or the name of the view as found in the /Tools/Views menu)
- **toolbar_section** (*str*) – Where, in the toolbar, to insert the button. See GPS.Action.button()
- **button_label** (*str*) – The label to use for the button (defaults to the name of the action).

Returns a tuple (GPS.Action, GPS.Menu) The menu might be None if you did not request its creation.

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 unamed) 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
```

17.2.2 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 interfer 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

As for Background_Highlight, you need to override the process() function to perform actual work.

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.

must_highlight(*buffer*)

Parameters **buffer** (*GPS.EditorBuffer*) – The buffer to test.

Returns whether to highlight this buffer. The default is to higlight 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=None*, *whole_line=False*, *speedbar=False*, *style=None*, ***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"
- **editable** (*boolean*) – whether the text is editable by the user interactively.
- **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.
- **style** (*GPS.Style*) – the style to apply to the overlay.
- **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 preferrably 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 preferrably 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.

17.2.3 The gps_utils.console_process.py module

class gps_utils.console_process.**ANSI_Console_Process**(*command*)

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**(*command*, *close_on_exit=True*, *force=False*, *ansi=False*, *manage_prompt=True*, *task_manager=False*)

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, command):
     Console_Process.__init__ (self, command)
```

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 interfer with external programs that do their own screen management through ANSI commands (like a Unix shell for instance).
- **task_manager** (*boolean*) – If True, the process will be visible in the GPS tasks view and can be interrupted or paused by users. Otherwise, it is running in the background and never

visible to the user.

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

17.3 Plugins for external tools

17.3.1 QGen

The QGen Debugger manual can be found within the QGen User guide.

CHAPTER

EIGHTEEN

GNU FREE DOCUMENTATION LICENSE

Version 1.1, March 2000

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

18.1 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. Secondarily, 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.

18.2 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.

18.3 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.

18.4 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.

18.5 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.

18.6 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.'

18.7 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.

18.8 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.

18.9 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.

18.10 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.

18.11 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.

18.12 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.

CHAPTER

NINETEEN

INDICES AND TABLES

- *genindex*

a

d

g

h

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y