GPS Programmer's Guide

Version 4.3.0
Document revision level $Revision: 118724 $
Date: $Date: 2007-10-29 15:35:50 +0100 (Mon, 29 Oct 2007) $

AdaCore

Version: 0.27: 19 August 2008

Version: 0.27: 19 August 2008

Table of Contents

1 Introduction

Important note: This document is not ready for release yet.

This document explains how to add your own modules to the GPS programming system.

GPS is a fully open architecture, to which one can add new features ranging from new menu items to launch external tools to full support for new languages, including cross-references.

Some of these additions can be done solely through the use of text files. These are for instance adding new key bindings to various parts of GPS, for instance in the editor. The end-user can also easily add new menus or toolbar buttons. See the customization chapters in the GPS user's guide.

This document will focus on these additions that can only be done through programming languages.

At this point, GPS can only be extended by programming in **Ada**. In addition, it is planned for the near future that extensions in **C** or **C**++ can be done. Work is under way to extend python scripting in GPS.

Likewise, adding basic support for new languages will be made easier, and doable through external text files, requiring no programming. This is not available for this first release of the GPS environment.

2 System Setup

As explained in the introduction, GPS can currently only be extended by programming in Ada. This assumes that a number of tools are available on your system, so that you can recompile your new module.

Most of these external tools and libraries are available from http://libre.act-europe.fr.

GNAT 3.15 or above
: GNAT is the GNU Ada Compiler, integrated into the gcc tool chain, and developed by **Ada Core Technologies** and **ACT Europe**. GPS will not compile with other Ada compilers than GNAT.

Gtk+ 2.2.0 or above
: gtk+ is a C toolkit used for the graphical interface of GPS. It is available on a number of platforms, including most UNIX systems and Windows. Available from http://www.gtk.org.

GPS sources
: The GPS sources include the corresponding GNAT, GtkAda and GVD sources needed to build it. If needed, GNAT, GtkAda and GVD sources can be obtained seperately from anonymous cvs access from http://libre.act-europe.fr

The GPS sources contain an INSTALL file that explains how to recompile GPS itself. GPS knows how to dynamically load a module. As a result, you do not necessarily need to rebuild GPS itself to add new modules, although the dynamic loading hasn't been fully tested yet and might not work on all platforms.

3 The GPS modules

GPS is organized around the concept of modules. The only part of GPS that is mandatory is its kernel (see Chapter 5 [The GPS Kernel], page 7), all the other tools, menus and features are provided in optional modules.

Although currently all modules have to be loaded at startup, some proof of concept for dynamically loadable module was implemented, and will most likely be part of a future version of GPS.

Every new feature you implement will be part of one or more modules. We will go through the details of creating new modules all along this manual, starting from a simple Hello World module to more advanced features like providing new shell or python commands.

Generally speaking, a module provides a limited set of features, and adds new GUI features in the GPS interface, like menus, toolbar buttons, contextual menu entries, new windows,. . . As much as possible, a menu shouldn't directly depend on any other module, only on the GPS kernel itself.

See the file '`gps-kernel-modules.ads`' for more information on modules.

4 Hello World walk through

Creating a new module is best demonstrated by going through the classical and simple example "hello world". This example will be refined as new extension possibilities are described later on in this document.

4.1 Declaring the module

A module is generally implemented in a separate source file, at this point an Ada package. The first thing that needs to be done is to create the specs of this package. Most of the time, a single function has to be exported, which is called Register_Module by convention. Therefore, we have to create a new directory to contain the module (we'll call it 'hello_world'), at the same level as other modules like the source editor.

Still by convention, the sources are put in a directory called 'src', and the object files are kept in a separate directory called 'obj'.

```
mkdir hello_world
mkdir hello_world/src
mkdir hello_world/obj
```

In the source directory, we create the file 'hello_world.ads', which contains the declaration of the Register_Module subprogram.

```
with GPS.Kernel;
package Hello_World is
   procedure Register_Module
      (Kernel : access GPS.Kernel.Kernel_Handle_Record'Class);
end Hello_World;
```

Before going over the details of the implementation of Register_Module, we have to make sure that the rest of GPS knows about this module, and that we know how to compile it

4.2 Publicizing your module

Until GPS provides dynamic modules, you have to modify the main subprogram of GPS to make it aware of your module.

This is done by modifying the file 'gps.adb', and adding two statements in there: a with statement that imports 'hello_world'.ads, and a call to Hello_World.Register_Module. See for instance how this is done for the keymanager module.

4.3 Compiling your module

However, after the addition of the two statements in 'gps.adb', the file 'hello_world.ads' will not be found automatically by GPS. Therefore, you need

to create a project file for your new module (we'll call it 'hello_world.gpr'), and add a dependency to it in the root project file of GPS ('gps/gps.gpr'), as is currently done for all other modules.

The project file 'hello_world.gpr' is best created by copying the project file from any other module, for instance the aliases module ('aliases/aliases.gpr'), and changing the name of the project to Hello_World.

You must also create a set of two Makfiles, which are used to add files other than Ada, even if your module only uses Ada files. Once again, this is best done by copying the two Makefiles from the directory 'aliases', renaming them into 'Makefile' and 'Makefile.hello_world', and replacing the strings aliases and ALIASES by resp. hello_world and HELLO_WORLD.

These steps will be made easier in the near future, but in any case are relatively straightforward, and only need to be done once per module. The resulting setup automatically takes into account all sources files that will be added later on to the module, either C or Ada, and compile them with the appropriate compiler.

You might also prefer in your first attempt at creating a new module to add your new files into the 'src' directory of an existing module. In this case, you don't have to create any of the project files or Makefile, nor to modify the 'gps.adb' file.

Once the project file has been created, and a dependency added in 'gps.gpr', you might want to reload the GPS project in GPS, so that the editing of your sources can be done in an Ada-friendly context.

4.4 Registering the module

Back to the source files of your modules. We now need to create a body for the procedure Register_Module. The minimal thing this function has to do is indicate to the GPS kernel that a new module is being declared, and give it a name. If you only do that, there is no direct impact on the rest of GPS. However, as we will see during in this guide, having a specific Module_Id is mandatory for some of the advanced feature, so it is cleaner to always declare one from the start.

This is done by creating the file 'hello_world.adb', with the following contents.

```
with GPS.Kernel.Modules;  use GPS.Kernel, GPS.Kernel.Modules;

package Hello_World is
   procedure Register_Module
      (Kernel : access GPS.Kernel.Kernel_Handle_Record'Class)
   is
```

```
      Module : Module_ID;
   begin
      GPS.Kernel.Modules.Register_Module
         (Module, Kernel, Module_Name => "hello_world");
   end Register_Module;

end Hello_World;
```

At this point, the hello_world module is compilable, only it won't do anything but be loaded in GPS.

The following sections will show how new features can be provided to the rest of GPS.

5 The GPS Kernel

6 Intermodule communication

As described above, GPS is organized into largely independent modules. For instance, the various views, browsers, help, vcs support,... are separate modules, that can either be loaded at startup or not.

When they are not loaded, the correspondings features and menus are not available to the user.

These modules need to communicate with each other so as to provide the best possible integration between the tools. There currently exists a number of ways to send information from one module to another. However, some of these technics depend on Ada-specific types, and thus makes it harder to write modules in different languages like C or Python.

The following communication technics are currently provided:

- Direct calls A module can explicitly specify that it depends on another one. This is done by changing the project file, and adding the necessary "with" statements in the code. This technics is highly not recommended, and should never be used when one of the other technics would do the job, since it defeats the module independency. The only place it is currently used at is for direct calls to the Register_* commands. Most of the time, these Register_* subprograms are also available through XML customization files, and thus limit the direct dependencies between modules, while providing greated extensibility to the final user.
- Shell calls Each module can register new shell commands for the interactive shell window. Any other module can call these commands. There is no direct dependency on the code, although this means that the module that provide the command must be loaded before the other module. This technics is used for instance for the codefix module, that needs a high degree of integration with the source editor module. It will also be used for communicating with Emacs.
- Addition to contextual menus A module is free to add entries to the main menu bar or to any contextual menus within GPS.

  Most of the time, a module will decide to add new entries depending on what the contextual menu applies to (the current context), although it might also decide to do that based on what module is displaying the contextual menu. Modules are identified by their name, which can easily be tested by other menus.
- Context changes Every time a new MDI child is selected, or when a module chooses to emit such a signal, a context change is reported via a gtk+ signal. A context is an Ada tagged type, created by the currently active module. There exists different kinds of contexts, some for files (directories

and project), others for entities (same as before, but with an entity name in addition, other for a location (adding line and column),... New types of contexts can be created by the modules without impacting the rest of GPS. All callbacks must test that the context they receive matches what they can handle.

These contexts are also used for the contextual menus

A module can choose to emit the signal to report changes to its context by emitting the signal. Other modules can they update their content accordingly. This is how the switches editor updates the project/directory/file it is showing when a new selection is done in the project view.

- hooks and action hooks Hooks are similar to the usual gtk+ signals. Each hook is a named collection of subprograms to be called when the hook is executed. Such hooks are executed by various parts of GPS when some actions take place, like reloading the project, loading a file,. . .

  These are the most powerful way for a module to react to actions taking place in other parts of GPS, and to act appropriately.

  In most cases, all the subprograms in a hook are executed in turn, and thus they all get a chance to act.

  However, in some other cases, the subprograms are only executed until one of them indicates that it has accomplished a useful action, and that no other subprogram from this hook should be called. These are called **action hooks**. This is the fundamental mechanism used by GPS to request for instance the edition of a file: the module that wishes to display a file executes the hook `"open_file_action_hook"` with the appropriate argument. At this point, all subprograms bound to this hook are executed, until one of them acknowledge that it knows how to edit this file (and hopefully opens an editor). Then no other subprogram from this hook is called, so that the file is not open multiple times.

  This mechanism is used for instance by the module that handles the external editors. It is setup so that it binds to the `"open_file_action_hook"` hook. Any time a file needs to be open, the callback from this module is called first. If the user has indicated that the external editor should always be used, this external editors module opens the appropriate editor, and stops the execution of the hook. However, if the user didn't wish to use an external editor, this module does nothing, so that the callback from the source editor module is called in turn, and can thus open the file itself.

  See '`gps-kernel-hooks.ads`' for more information on hooks.

7 Documenting your module

All modules should be documented, so that the users are aware of all its capabilities.

There are several levels of documentation:

- Tooltips It is recommended that all new preferences and as much of the GUI as possible be documented through tooltips. This is the only help that most users will read.

  Tooltips are easily added directly with gtk+: Just call `Gtk.Tooltips.Set_Tooltip` with the appropriate parameters. The kernel itself contains a tooltip group, which should be used when setting new tooltips. This is so that a common timeout is used for all tooltips in the application: when a user has waited long enough for the first tooltip to be displayed, he won't have to wait again for the other tooltips.
- extended documentation Extended documentation should be written in HTML. See the GPS user's guide on how to make new documentation available to users.

8 Debugging

8.1 X11 server

If you are developing on a linux system, it is recommended that you reconfigure your X11 server with the following setup (see the file '/etc/X11/XF86Config-4'):

```
Section "ServerFlags"
        Option "AllowDeactivateGrabs" "true"   # Ctrl+Alt+Keypad *
        Option "AllowClosedownGrabs"  "true"   # Ctrl+Alt+Keypad /
EndSection
```

The two key bindings described above are used to release any grab that a GUI application might have. This is especially useful when debugging through gdb: it might happen that the breakpoint happens while such a grab is in place, and would therefore prevent any input (mouse or keyboard) to any application in your X11 session, in particular the debugger.

8.2 gtk+ library

It is also recommended that you recompile your own gtk+ library (on systems where this is easily doable such as Unix systems), with the following configure command:

```
./configure --with-debug=yes
```

In addition to providing the usual debugging information in the debugger, this also activates several environment variables which might be used to monitor the actions in gtk+ and its associated libraries.

These variables are the following:

```
export GTK_DEBUG=misc:plugsocket:text:tree:updates:keybindings;
export GDK_DEBUG=updates:nograbs:events:dnd:misc:xim:colormap:gdkrb:gc:pixmap:image:input:cursor;
export GOBJECT_DEBUG=objects:signals;
```

Some of the values for these variables can be omitted. The exact semantic (or even the exact list) of such variables depends on your version of gtk+, and you should therefore consult its documentation.

8.3 debugger

When debugging with gdb, it is recommended that you always specify the flag --sync to gps. This forces any gtk+ application, and in particular GPS, to process X11 events synchronously, and therefore makes it easier to debug possible problems.

If your application is printing some gtk+ warnings on the console, you should do the following in the debugger:

```
(gdb) set args --sync
(gdb) begin
(gdb) break g_log
(gdb) cont
```

This will stop the application as soon as the gtk+ warning is printed.

9 Contexts

Index