Ada Mode

An Emacs major mode for programming Ada 95 with GNAT
July 2000 for Ada Mode Version 3.5

1 Overview

The Emacs mode for programming in Ada 95 with GNAT helps the user in understanding existing code and facilitates writing new code. It furthermore provides some utility functions for easier integration of standard Emacs features when programming in Ada.

1.1 General features:

- full Integrated Development Environment :
  - support of 'project files' for the configuration (directories, compilation options,...)
  - compiling and stepping through error messages.
  - running and debugging your applications within Emacs.
- easy to use for beginners by pull-down menus,
- user configurable by many user-option variables.

1.2 Ada mode features that help understanding code:

- functions for easy and quick stepping through Ada code,
- getting cross reference information for identifiers (e.g. find the defining place by a keystroke),
- displaying an index menu of types and subprograms and move point to the chosen one,
- automatic color highlighting of the various entities in Ada code.

1.3 Emacs support for writing Ada code:

- switching between spec and body files with eventually auto-generation of body files,
- automatic formating of subprograms parameter lists.
- automatic smart indentation according to Ada syntax,
- automatic completion of identifiers,
- automatic casing of identifiers, keywords, and attributes,
- insertion of statement templates,
- filling comment paragraphs like filling normal text,

2 Installation

If you got the Ada mode as a separate distribution, you should have a look at the 'README' file. It explains the basic steps necessary for a good installation of the emacs Ada mode.

Installing the Ada mode is basically just a matter of copying a few files into the Emacs library directories. Every time you open a file with a file extension of '.ads' or '.adb', Emacs will automatically load and activate the Ada mode.

See the section Chapter 17 [Using non-standard file names], page 21 if your files do not use these extensions and if you want Emacs to automatically start the Ada mode every time you edit an Ada file.

See also the Emacs documentation ⟨undefined⟩ [(emacs)], page ⟨undefined⟩ for general usage variables that you might want to set.

2.1 Required files

This Ada mode works best with Emacs 20.3 or higher (the easy editing features for the project files won't work with any older version), but most of the commands should work with older versions too. Please try to install the most recent version of Emacs on your system before installing the Ada mode.

Although part of the Ada mode is compiler independent, the most advanced features are specific to the Gnat compiler `http://www.gnat.com`.

The following files are provided with the Ada mode distribution:

- '`ada-mode.el`': The main file for the Ada mode. This is the only file which does not require Gnat. It contains the functions for indentation, formatting of parameter lists, stepping through code, comment handling and automatic casing. Emacs versions 20.2 and higher already contain Ada mode version 2.27, which is an older version of this file and should be replaced. Loading '`ada-mode.el`' from the current distribution supersedes the standard installation.
- '`ada-stmt.el`': Contains the statement templates feature.
- '`ada-xref.el`': This file provides the main support for Gnat. This is where the functions for cross-references, completion of identifiers, support for project files and compilation of your application are defined.
- '`ada-prj.el`': The functions to use for easy-edition of the project files. This file is the only one which really requires Emacs at least 20.2. It uses the new widget features from Emacs.

3 Customizing the Ada mode

The ada-mode is fully customizable. Everything, from the file names to the automatic indentation and the automatic casing can be adapted to your own needs.

There are two different kinds of variables that control this customization, both are easy to modify.

The first set of variables are standard Emacs variables. Of course, some are defined only for the Ada mode, whereas others have a more general meaning in Emacs. Please see the Emacs documentation for more information on the latest. In this documentation, we will detail all the variables that are specific to the Ada mode, and a few others. The names will be given, as in `ada-case-identifier`.

Emacs provides an easy way to modify them, through a special mode called customization. To access this mode, select the menu *Ada->Customize*. This will open a new buffer with some fields that you can edit. For instance, you will get something like:

```
Put below the compiler switches.
comp_opt= ____________________________________
```

The first line gives a brief description of the variable. The second line is the name of the variable and the field where you can give a value for this variable. Simply type what you want in the field.

When you are finished modifying the variables, you can simply click on the **Save for future sessions** button at the top of the buffer (click with the middle mouse button). This will save the values in your '`.emacs`' file, so that next time you start Emacs they will have the same values.

To modify a specific variable, you can directly call the function `customize-variable` from Emacs (just type ⟨M-x customize-variable RET⟩ and then type the variable name.

Some users might prefer to modify the variables directly in their configuration file, '`.emacs`'. This file is coded in Emacs lisp, and the syntax to set a variable is the following:

```
(setq variable-name value)
```

The second set of variables for customization are set through the use of project files. These variables are specific to a given project, whereas the first set was more general. For more information, please See Chapter 4 [Project files], page 4.

4 Project files

4.1 General overview

Emacs provides a full Integrated Development Environment for GNAT and Ada programmers. That is to say, editing, compiling, executing and debugging can be performed within Emacs in a convenient and natural way.

To take full advantage of this features, it is possible to create a file in the main directory of your application, with a '.adp' extension. This file contain all needed information dealing with the way your application is organized between directories, the commands to compile, run and debug it etc. Creating this file is not mandatory and convenient defaults are automatically provided for simple setups. It only becomes necessary when those above mentioned defaults need customizing.

A simple way to edit this file is provided for Emacs 20.2 or newer, with the following functions, that you can access also through the Ada menu. It is also possible to edit the project file as a regular text file.

Once in the buffer for editing the project file, you can save your modification using the '[OK]' button at the bottom of the buffer, or simply use the usual *C-x C-s* binding. To cancel your modifications, simply kill the buffer or click on the '[CANCEL]' button at the button.

Each buffer using Ada mode will be associated with one project file when there is one available, so that Emacs can easily navigate through related source files for instance.

The exact algorithm to determine which project file should be used is described in the next section, but you can force the project file you want to use by setting one or two variables in your '`.emacs`' file.

- To set up a default project file to use for any directory, anywhere on your system, set the variable `ada-prj-default-project-file` to the name of that file.

  ```
  (set 'ada-prj-default-project-file "/dir1/dir2/file")
  ```

- For a finer controlled, you can set a per-directory project file. This is done through the variable `ada-xref-default-prj-file`.

  ```
  (set 'ada-xref-default-prj-file
       '(("/dir1/dir2" . "/dir3/file1")
         ("/dir4/dir5" . "/dir6/file2")))
  ```

  Note: This has a higher priority than the first variable, so the first choice is to use this variable settings, and otherwise `ada-prj-default-project-file`.

`C-c u ada-customize menu: Ada->Project->New/Edit`
: Create or edit the project file for the current buffer.

`C-c c ada-change-prj`
: Change the project file associated with the current Ada buffer.

`C-c d`
: Change the default project file for the current directory. Every new file opened from this directory will be associated with that file by default.

ada-set-default-project-file menu: Ada->Project->Set Default

Set the default project file to use for *any* Ada file opened anywhere on your system. This sets this file only for the current Emacs session.

4.2 Project file variables

The following variables can be defined in a project file. They all have a default value, so that small projects do not need to create a project file.

Some variables below can be referenced in other variables, using a shell-like notation. For instance, if the variable `comp_cmd` contains a sequence like `${comp_opt}`, the value of that variable will be substituted.

Here is the list of variables:

`src_dir [default: "./"]`

This is a list of directories where the Ada mode will look for source files. These directories are used mainly in two cases, both as a switch for the compiler and for the cross-references.

`obj_dir [default: "./"]`

This is a list of directories where to look for object and library files. The library files are the .ali files generated by Gnat and that contain cross-reference informations.

`comp_opt [default: ""]`

Creates a variable which can be referred to subsequently by using the `${comp_opt}` notation. This is intended to store the default switches given to 'gnatmake' and 'gcc'.

`bind_opt=SWITCHES [default: ""]`

Creates a variable which can be referred to subsequently by using the `${bind_opt}` notation. This is intended to store the default switches given to 'gnatbind'.

`link_opt=SWITCHES [default: ""]`

Creates a variable which can be referred to subsequently by using the `${link_opt}` notation. This is intended to store the default switches given to 'gnatlink'.

`main=EXECUTABLE [default: ""]`

Specifies the name of the executable for the application. This variable can be referred to in the following lines by using the `${main}` notation.

`cross_prefix=PREFIX [default: ""]`

This variable should be set if you are working in a cross-compilation environment. This is the prefix used in front of the gnatmake commands.

`remote_machine=MACHINE [default: ""]`

This is the name of the machine to log into before issuing the compilation command. If this variable is empty, the command will be run on the local machine. This will not work on Windows NT machines, since the Ada mode will simply precede the compilation command with a 'rsh' command, unknown on Windows.

`comp_cmd=COMMAND [default: "${cross_prefix}gcc -c -I${src_dir} -g -gnatq"]`
: Specifies the command used to compile a single file in the application. The name of the file will be added at the end of this command.

`make_cmd=COMMAND [default: "${cross_prefix}gnatmake ${main} -aI${src_dir} -aO${obj_dir} -g -gnatq -cargs ${comp_opt} -bargs ${bind_opt} -largs ${link_opt}"]'`
: Specifies the command used to recompile the whole application.

`run_cmd=COMMAND [default: "${main}"]`
: Specifies the command used to run the application.

`debug_cmd=COMMAND [default: "${cross_prefix}gdb ${main}"]`
: Specifies the command used to debug the application

4.3 Detailed algorithm

This section gives more details on the project file setup and is only of interest for advanced users.

Usually, an Ada file is part of a larger application, whose sources and objects can be spread over multiple directories. The first time emacs is asked to compile, run or debug an application, or when a cross reference function is used (goto declaration for instance), the following steps are taken:

- find the appropriate project file, open and parse it. All the fields read in the project file are then stored by emacs locally. Finding the project file requires a few steps:
  - if a file from the same directory was already associated with a project file, use the same one. This is the variable `ada-xref-default-prj-file` described above.
  - if the variable `ada-prj-default-project-file` is set, use the project file specified in this variable.
  - if there is a project file whose name is the same as the source file except for the suffix, use this one.
  - if there's only one project file in the source directory, use that one.
  - if there are more than one project file in the source directory, ask the user.
  - if there are no project files in the source directory use standard default values.

  The first project file that is selected in a given directory becomes the default project file for this directory and is used implicitly for other sources unless specified otherwise by the user.
- look for the corresponding .ali file in the `obj_dir` defined in the project file. If this file can not be found, emacs proposes to compile the source using the `comp_cmd` defined in the project file in order to create the ali file.
- when cross referencing is requested, the .ali file is parsed to determine the file and line of the identifier definition. It is possible for the .ali file to be older than the source file, in which case it will be recompiled if the variable `ada-xref-create-ali` is set, otherwise the reference is searched in the obsolete ali file with possible inaccurate results.
- look for the file containing the declaration using the source path `src_dir` defined in the project file. Put the cursor at the correct position and display this new cursor.

5 Syntax highlighting

The Ada mode is made to help you understand the structure of your source files. Some people like having colors or different fonts depending on the context: commands should be displayed differently than keywords, which should also be different from strings, ...

Emacs is able to display in a different way the following syntactic entities:

- keywords
- commands
- strings
- gnatprep statements (preprocessor)
- types (under certain conditions)
- other words

This is not the default behavior for Emacs. You have to explicitly activate it. This requires that you add a new line in your '`.emacs`' file (if this file does not exist, just create it).

```
(global-font-lock-mode t)
```

But the default colors might not be the ones you like. Fortunately, there is a very easy way to change them. Just select the menu *`Help->Customize->Specific Face...`* and press *`Return`*. This will display a buffer will all the "faces" (the colors) that Emacs knows about. You can change any of them.

6 Moving Through Ada Code

There are several easy to use commands to stroll through Ada code. All these functions are available through the Ada menu, and you can also use the following key bindings or the command names:

M-C-e `ada-next-procedure`
: Move to the next function/procedure/task, which ever comes next.

M-C-a `ada-previous-procedure`
: Move to previous function/procedure/task.

`ada-next-package`
: Move to next package.

`ada-prev-package`
: Move to previous package.

C-c C-a `ada-move-to-start`
: Move to matching start of `end`. If point is at the end of a subprogram, this command jumps to the corresponding `begin` if the user option `ada-move-to-declaration` is `nil` (default), it jumps to the subprogram declaration otherwise.

C-c C-e `ada-move-to-end`
: Move point to end of current block.

C-c o `ff-find-other-file`
: Switch between corresponding spec and body file. If the cursor is on a subprogram, switch between declaration and body.

C-c c-d
: Move from any reference to its declaration and switch between declaration and body (for procedures, tasks, private and incomplete types).

C-c C-r `ada-find-references`
: runs the '`gnatfind`' command to search for all references to the entity pointed by the cursor. Use 'next-error' function, or C-x ', to visit each reference (as for compilation errors).

These functions use the information in the output of the Gnat Ada compiler. However, if your application was compiled with the `-gnatx` switch, these functions will not work, since no extra information is generated by GNAT. See GNAT documentation for further information.

Emacs will try to run Gnat for you whenever the cross-reference informations are older than your source file (provided the `ada-xref-create-ali` variable is non nil). Gnat then produces a file with the same name as the current Ada file but with the extension changed to `.ali`. This files are normally used by the binder, but they will also contain additional cross-referencing information.

7 Identifier completion

7.1 Overview

Emacs and the Ada mode provide two general ways for the completion of identifiers. This is an easy way to type faster: you just have to type the first few letters of an identifiers, and then loop through all the possible completions.

The first method is general for Emacs. It will work both with Ada buffers, but also in C buffers, Java buffers, ... The idea is to parse all the opened buffers for possible completions.

For instance, if the following words are present in any of the opened files: my_identifier, my_subprogam, then you will have this scenario:

```
You type:  my⟨M-/⟩
Emacs will display:  my_identifier
If you press ⟨M-/⟩ once again, Emacs will replace my_identifier with
my_subprogram.
Pressing ⟨M-/⟩ once more will bring you back to my_identifier.
```

This is a very fast way to do completion, and the casing of words will also be respected.

The second method is specific to Ada buffer, and even to users of the Gnat compiler. Emacs will search the cross-information found in the .ali files generated by Gnat for possible completions.

The main advantage is that this completion is more accurate: only existing identifier will be suggested, you don't need to have a file opened that already contains this identifiers,...

On the other hand, this completion is a little bit slower and requires that you have compiled your file at least once since you created that identifier.

7.2 Summary of commands

C-TAB *ada-complete-identifier*
: complete accurately current identifier using information in .ali file

M-/
: complete identifier using buffer information (not ada specific)

8 Index Menu of Subprograms

You can display a choice menu with all procedure/function/task declarations in the file and choose an item by mouse click to get to its declaration. This function is accessible through the 'Ada' menu when editing a Ada file, or simply through the following key binding :

C-S-mouse-3
: display index menu

9 File Browser

Emacs provides a special mode, called `speedbar`. When this mode is activated, a new frame is displayed, with a file browser. The files from the current directory are displayed, and you can click on them as you would with any file browser. The following commands are then available.

You can click on a directory name or file name to open it. The editor will automatically select the best possible mode for this file, including of course the ada-mode for files written in Ada

If you click on the [+] symbol near a file name, all the symbols (types, variables and subprograms) defined in that file will be displayed, and you can directly click on them to open the right file at the right place.

You can activate this mode by typing ⟨M-x speedbar⟩ in the editor. This will open a new frame. A better way might be to assicate the following key binding

```
(global-set-key [f7]  'speedbar-get-focus)
```

Every time you press ⟨f7⟩, the mouse will automatically move to the speedbar frame (which will be created if it does not exist).

10 Automatic Smart Indentation

The Ada mode comes with a full set of rules for automatic indentation. You can of course configure the indentation as you want, by setting the value of a few variables.

As always, the preferred way to modify variables is to use the `Ada->Customize` menu (don't forget to save your changes!). This will also show you some example of code where this variable is used, and hopefully make things clearer.

The relevant variables are the following:

`ada-broken-indent (default value: 2)`
: Number of columns to indent the continuation of a broken line

`ada-indent (default value: 3)`
: Width of the default indentation

`ada-indent-record-rel-type (default value: 3)`
: Indentation for 'record' relative to 'type' or 'use'

`ada-indent-return (default value: 0)`
: Indentation for 'return' relative to 'function' (if ada-indent-return is greater than 0), or the open parenthesis (if ada-indent-return is negative or null). Note that in the second case, when there is no open parenthesis, the indentation is done relative to 'function' with the value of ada-broken-indent.

`ada-indent-renames (default value: 2)`
: Indentation for 'renames' relative to 'function'. This behaves exactly the same as ada-indent-return.

`ada-label-indent (default value: -4)`
: Number of columns to indent a label

`ada-stmt-end-indent (default value: 0)`
: Number of columns to indent a statement 'end' keyword on a separate line

`ada-when-indent (default value: 3)`
: Indentation for 'when' relative to 'exception' or 'case'

`ada-indent-is-separate (default value: t)`
: Non-nil means indent 'is separate' or 'is abstract' if on a single line

`ada-indent-to-open-paren (default value: t)`
: Non-nil means indent according to the innermost open parenthesis

`ada-indent-after-return (default value: t)`
: Non-nil means that the current line will also be re-indented before inserting a newline, when you press ***Return***.

Most of the time, the indentation will be automatic, i.e when you will press ***Return***, the cursor will move to the correct column on the next line.

However, you might want or need sometimes to re-indent the current line or a set of lines. For this, you can simply go to that line, or select the lines, and then press ***TAB***. This will automatically re-indent the lines.

Another mode of indentation exists that helps you to set up your indentation scheme. If you press ***C-c TAB***, the ada-mode will do the following:

- Reindent the current line, as *TAB* would do
- Temporarily move the cursor to a reference line, i.e the line that was used to calculate the current indentation
- Display at the bottom of the window the name of the variable that provided the offset for the indentation

The exact indentation of the current line is the same as the one for the reference line, plus an offset given by the variable.

Once you know the name of the variable, you can either modify it through the usual ⟨Ada->Customize⟩ menu, or by typing ⟨M-x customize-variable RET⟩ in the Emacs window, and then give the name of the variable.

TAB indent the current line or the current region.

M-C- indent lines in the current selected block.

`C-c` *TAB* indent the current line and prints the name of the variable used for indentation.

You can also want to reformat a set of files directly from the command line, without going into interactive mode. You should then use a command like:

```
emacs -batch -l ada-mode -f ada-batch-reformat file1 file2 ...
```

This will load the files, reformat them and save them with the same name. Note the order of the arguments on the command line, which is important. The files should appear last on the command line.

11 Formatting Parameter Lists

To help you correctly align fields in a subprogram parameter list, Emacs provides one function that will do most of the work for you. This function will align the declarations on the colon (':') separating argument names and argument types, plus align the 'in', 'out' and 'in out' keywords if required.

C-c C-f `ada-format-paramlist`
: Format the parameter list.

12 Automatic Casing

Casing of identifiers, attributes and keywords is automatically performed while typing when the variable `ada-auto-case` is set. Every time you press a word separator, the previous word is automatically cased.

You can customize the automatic casing differently for keywords, attributes and identifiers. The relevant variables are the following: `ada-case-keyword`, `ada-case-attribute` and `ada-case-identifier`.

All these variables can have one of the following values:

`downcase-word`
: The previous word will simply be in all lower cases. For instance `My_vARIable` is converted to `my_variable`.

`upcase-word`
: The previous word will be fully converted to upper cases. For instance `My_vARIable` is converted to `MY_VARIABLE`.

`ada-capitalize-word`
: All letters, except the first one of the word and every letter after the '_' character are lower cased. Other letters are upper cased. For instance `My_vARIable` is converted to `My_Variable`.

`ada-loose-case-word`
: No letters is modified in the previous word, except the ones after the '_' character that are upper cased. For instance `My_vARIable` is converted to `My_VARIable`.

`ada-no-auto-case`
: Do nothing. This can be used to disable some specific aspects of the auto-casing, for instance if you have some special rules for identifiers, not covered by the other functions above, but still want the keywords to be lower-cased.

These functions, although they will work in most cases, will not be accurate sometimes. The Ada mode allows you to define some exceptions, that will always be cased the same way.

The idea is to create a dictionary of exceptions, and store it in a file. This file should contain one identifier per line, with the casing you want to force. The default name for this file is '`~/.emacs_case_exceptions`'. You can of course change this name, through the variable `ada-case-exception-file`.

Note that each line in this file must start with the key word whose casing you want to specify. The rest of the line can be used for comments (explaining for instance what an abbreviation means, as recommended in the Ada 95 Quality and Style, paragrpah 3.1.4). Thus, a good example for this file could be:

```
DOD        Department of Defense
Text_IO
GNAT       The GNAT compiler from Ada Core Technologies
```

When working on project involving multiple programmers, we recommend that every member of the team sets this variable to the same value, which should point to a system-wide file that each of them can write. That way, you will ensure that the casing is consistent throughout your application(s).

There are two ways to add new items to this file: you can simply edit it as you would edit any text file, and add or suppress entries in this file. Remember that you should put one entity per line. The other, easier way, is to position the cursor over the word you want to add, in an Ada buffer. This word should have the casing you want. Then simply select the menu *Ada->Edit->Create Case Exception*, or the key *C-c C-y*. The word will automatically be added to the current list of exceptions and to the file.

It is sometimes useful to have multiple exception files around (for instance, one could be the standard Ada acronyms, the second some company specific exceptions, and the last one some project specific exceptions). If you set up the variable `ada-case-exception-file` as a list of files, each of them will be parsed and used in your emacs session.

However, when you save a new exception through the menu, as described above, the new exception will be added to the first file in the list only. You can not automatically add an exception to one of the other files, although you can of course edit the files by hand at any time.

Automatic casing can be performed on port or whole buffer using:

C-c C-b Adjust case in the whole buffer.

C-c C-y Create a new entry in the exception dictionary, with the word under the cursor

C-c C-t Rereads the exception dictionary from the file `ada-case-exception-file`.

13 Statement Templates

NOTE: This features are not available on VMS for Emacs 19.28. The functions used here do not exist on Emacs 19.28.

Templates exist for most Ada statements. They can be inserted in the buffer using the following commands:

`C-c t b` exception Block

`C-c t c` case.

`C-c t d` declare Block.

`C-c t e` else.

`C-c t f` for Loop.

`C-c t h` Header.

`C-c t i` if.

`C-c t k` package Body.

`C-c t l` loop.

`C-c t t` task Body.

`C-c t w` while Loop.

`C-c t u` use.

`C-c t x` exit.

`C-c t C-a` array.

`C-c t C-e` elsif.

`C-c t C-f` function Spec.

`C-c t C-k` package Spec.

`C-c t C-p` procedure Spec.

`C-c t C-r` record.

`C-c t C-s` subtype.

`C-c t C-t` task Spec.

`C-c t C-u` with.

`C-c t C-v` private.

`C-c t C-w` when.

`C-c t C-x` exception.

`C-c t C-y` type.

14 Comment Handling

By default, comment lines get indented like Ada code. There are a few additional functions to handle comments:

M-; Start a comment in default column.

M-j Continue comment on next line.

C-c ; comment-region
: Comment the selected region (add – at the beginning of lines).

C-c : Uncomment the selected region

M-q autofill the current comment.

15 Compiling Executing

Ada mode provides a much complete environment for compiling, debugging and running an application within Emacs.

All the commands used by Emacs to manipulate your application can be customized in the project file. Some default values are provided, but these will likely not be good enough for a big or even medium-sized project. See the section on the project file for an explanation on how to set up the commands to use.

One of the variables you can set in your project file, `cross_prefix`, indicates whether you are using a cross-compilation environment, and if yes for which target. The default command used for compilation will add this `cross_prefix` in front of the name: `gcc` will become `cross_prefix-gcc`, `gnatmake` will become `cross_prefix-gnatmake`, ...

This will also modify the way your application is run and debugged, although this is not implemented at the moment.

Here are the commands for building and using an Ada application

- Compiling the current source This command is issued when issuing the `compile` command from the Ada menu. It compiles unconditionally the current source using the `comp_cmd` variable of the project file. Compilation options can be customized with the variable `comp_opt` of the project file.

  Emacs will display a new buffer that contains the result of the compilation. Each line associated with an error will become active: you can simply click on it with the middle button of the mouse, or move the cursor on it and press *Return*. Emacs will then display the relevant source file and put the cursor on the line and column the error was found at.

  You can also simply press the *C-x `* key and Emacs will jump to the first error. If you press that key again, it will move you to the second error, and so on.

  Some error messages might also include references to some files. These references are also clickable in the same way.

- (Re)building the whole application This command is issued when you select the `build` command from the Ada menu. It compiles all obsolete units of the current application using the `make_cmd` variable of the project file. Compilation options can be customized with the variable `comp_opt` of the project file, binder options with `bind_opt` and linker options with `link_opt`. The main unit of the application may be specified with `main`.

  The compilation buffer is also active in the same way it was for the above command.

- Running the application This command is issued when you select the `run` command from the Ada menu. It executes the current application in an emacs buffer. Arguments can be passed through before executing. The execution buffer allows for interactive input/output.

  This command is not yet available in a cross-compilation toolchain. Emacs would first need to log on the target before running the application. This will be implemented in a future release of Gnat.

16 Debugging your application

You can set up in the project file a command to use to debug your application. Emacs is compatible with a lot of debuggers, and provide an easy interface to them.

This selection will focus on the gdb debugger, and two of the graphical interfaces that exist for it.

In all cases, the main window in Emacs will be split in two: in the upper buffer, the source code will appear, whereas the debugger input/output window is displayed at the bottom. You can enter the debugger commands as usual in the command window. Every time a new source file is selected by the debugger (for instance as a result of a `frame` command), the appropriate source file is displayed in the upper buffer.

The source window is interactive: you can click on an identifier with the right mouse button, and print its value in the debugger window. You can also set a breakpoint simply by right-clicking on a line.

You can easily use Emacs as the source window when you are using a graphical interface for the debugger. The interesting thing is that, whereas you still have the graphical nifties, you can also you the cross-references features that the ada-mode provides to look at the definition for the identifiers,...

Here is how you can set up gdbtk and ddd for use with Emacs (These are the commands you should setup in the project file):

- gdbtk should be used with the switch –emacs_gdbtk. It provides a nice backtrace window, as well as a tasks window. You can click interactively on both of them, and Emacs will display the source file on the correct line.
- ddd (Data Display Debugger) should be used with the switches –tty and -fullname. Whenever you print a variable from Emacs, it will be displayed graphically in the data window.

17 Using non-standard file names

By default, Emacs is configured to use the GNAT style file names, where file names are the package names, and the extension for spec and bodies are respectively .ads and .adb.

If you want to use other types of file names, you will need to modify your .emacs configuration file.

Adding new possible extensions is easy. Since the ada-mode needs to know how to go from the body to the spec (and back), you always have to specify both. A function is provided with the ada-mode to add new extensions.

For instance, if your files are called <unit>_s.ada and <unit>_b.ada respectively for spec and bodies, you need to add the following to your '.emacs' :

```
(ada-add-extensions "_s.ada" "_b.ada")
```

Note that it is possible to redefine the extension, even if they already exist, as in:

```
(ada-add-extensions ".ads" "_b.ada")
(ada-add-extensions ".ads" ".body")
```

This simply means that whenever the ada-mode will look for the body for a file whose extension is '.ads', it will take the first available file that ends with either '.adb' (standard), '_b.ada' or '.body'.

If the filename is not the unit name, then things are a little more complicated. You then need to rewrite the function ada-make-filename-from-adaname (see the file 'ada-mode.el' for an example).

18 Working Remotely

When you work on project that involve a lot of programmers, it is generally the case that you will edit the files on your own machine, but you want to compile, run and debug your application in another buffer.

Fortunately, here too Emacs provides a very convenient way to do this.

18.1 Remote editing

First of all, the files do not need to be on your machine. Emacs can edit any remote file, by doing transparent FTP sessions between your machine and the remote machine that stores your files. This is a special Emacs mode, called `ange-ftp`. To use it, you just have to use a slightly different syntax when you open a file.

```
For instance, if you want to  open the file /work/foo.adb on the machine
aleph.gnu.org, where you log in as qwe, you would simply do this:

⟨C-x C-f⟩ /qwe@aleph.gnu.org:/work/foo.adb ⟨Return⟩

i.e put your name, the name of the machine and the name of the file.
```

The first time, Emacs will ask you for a password that it will remember until you close the current Emacs. Even if the ftp session times out, you won't need to reenter your password.

Every time you save the file, Emacs will upload it to the remote machine transparently. No file is modified on the local machine.

18.2 Remote compiling

If the machine you want to compile on is not the one your Emacs is running on, you can set the variable `remote_machine` in the project file for your application.

This will force Emacs to issue a rsh command for the compilation, instead of running it on the local machine. Unfortunately, this won't work on Windows workstations, since this protocol is not supported.

```
If your remote_machine is aleph.gnu.org and the standard
compilation command is cd /work/ && gnatmake foo, then Emacs will
actually issue the command rsh aleph.gnu.org 'cd /work/ &&
gnatmake foo'.
```

The advantage of using the `remote_machine` variable is that it is easier to change that machine without having to modify the compilation command.

Note that if you need to set up some environment variables before the compilation, you need to insert a call to the appropriate initialization script in the compilation command, for instance:

```
build_cmd= initialization_script ; cd /work/ && gnatmake foo
```

18.3 Remote running and debugging

This feature is not completely implemented yet.

However, most of the time, you will be able to run your application remotely simply by replacing it with a 'rsh' call on Unix.

```
For instance, if your command was '${main}', you could replace it with
'rsh aleph.gnu.org ${main}'.
```

However, this would not fully work for instance on vxworks, where rsh is not supported.

Table of Contents