ECB version 2.33beta2 - User manual

ECB stands for "Emacs Code Browser". While (X)Emacs already has good **editing** support for many modes, its **browsing** support is somewhat lacking. That's where ECB comes in: it displays a number of informational windows that allow for easy source code navigation and overview.

The informational windows can contain:

- A directory tree,
- a list of source files in the current directory,
- a list of functions/classes/methods/... in the current file, (ECB uses the Semantic Bovinator, or Imenu, or etags, for getting this list so all languages supported by any of these tools are automatically supported by ECB too)
- a history of recently visited files,
- the Speedbar and
- output from compilation (the "*compilation*" window) and other modes like help, grep etc. or whatever a user defines to be displayed in this window.

As an added bonus, ECB makes sure to keep these informational windows visible, even when you use `C-x 1` and similar commands.

It goes without saying that you can configure the layout, ie which informational windows should be displayed where. ECB comes with a number of ready-made window layouts to choose from.

Please note: Experienced ECB users find a complete alphabetical list of all commands and user-options in ⟨undefined⟩ [Interactive ECB commands], page ⟨undefined⟩ and ⟨undefined⟩ [Customizable options], page ⟨undefined⟩.

The latest version of ECB can always be found at the URL
`http://ecb.sourceforge.net`.

To send bug reports, or participate in discussions about ECB, use the mailing list
`ecb-list@lists.sourceforge.net` via the URL
`http://lists.sourceforge.net/lists/listinfo/ecb-list`.

IMPORTANT: Cause of extra appearance of SPAM in the mailing-lists, SourceForge has changed its policy: Now it is only possible to post to the mailing-list for users who have subscribed this mailing-list. So please be aware you will not be able to send comments, bug reports and improvement suggestions before you have subscribed the ECB-mailing-list. See the section "Mailing-list" at the ECB-website at `http://ecb.sourceforge.net` how to do this.

1 Installation and first steps of ECB

This chapter describes how to install ECB and setup (X)Emacs correctly and what are the first steps after activation of ECB.

1.1 Installation of ECB

This section describes how to install ECB.

1.1.1 Installation of ECB for XEmacs users

Basic requirement: ECB requires a XEmacs-version >= 21!

For XEmacs-users it is strongly recommended to use the package-management-system of XEmacs for first-time downloading/installing ECB or for upgrading to a newer version of ECB. Here is a short guide (for details about the package-manager of XEmacs see the related info-manual):

Caution: If ECB is already installed and you just want upgrading to a newer version then it is recommended to deactivate ECB before proceeding with the steps below!

1. Choose a download-site

   This can be done via the menu "Tools –> Packages –> Add Download Site": Choose one of the listed sites. Or you can customize the option `package-get-remote` by hand and save it for future sessions.

2. Activate the packages list

   This can be done either by the menu "Tools –> Packages –> List and Install" or via the command `pui-list-packages`. After that a special packages-buffer is displayed where you can interactively install or upgrade packages. At the end of this buffer there is a short description how to use this buffer.

3. Install ECB and all required packages

   Mark the package named "ecb" for install. Do this also for the required packages "semantic", "eieio" and "speedbar". The package "mail-lib" is needed for easy submitting of problem-reports to the ECB-maintainers and the package "c-support" is needed for easy using hideshow within the Methods-buffer of ECB[1].

   After marking all needed packages for installation hit `x` to install them.

   If you have already installed ECB and you want just upgrading to the latest available version then proceed as described above - same if you want to upgrade one of the required packages.

4. Start ECB

   Now you can immediately start ECB via the command `ecb-activate`; there is no need to restart XEmacs! As an alternative you can first read the online-help via `ecb-show-help`.

If you do not like the package-manager of XEmacs but you want installing ECB "by hand" direct from the ECB-website then you have to follow the instructions for GNU Emacs, see ⟨undefined⟩ [GNU Emacs Installation], page ⟨undefined⟩.

[1] All required packages can simply autom. marked by hitting `r` in the packages buffer. But this installs a lot of packages more (e.g. the Newsreader Gnus) which are really not essential for ECB. Therefore it is recommended to mark the required packages by hand.

1.1.2 Installation of ECB for GNU Emacs users

Basic requirement: ECB requires an Emacs-version >= 21!

IMPORTANT: If you are a XEmacs-user please read ⟨undefined⟩ [XEmacs Installation], page ⟨undefined⟩ before proceeding with the following instructions!

Using the new cedet 1.0 suite: From beginning with version 2.01 ECB supports the next generation of the cedet-tools. But before the cedet 1.0 suite becomes stable this means that ECB runs correctly with loaded cedet 1.0 but the ECB-upgrading feature (see ⟨undefined⟩ [Downloading new versions], page ⟨undefined⟩) does not support autom. upgrading to latest available cedet versions. This will be first available after first stable release of the new cedet-library 1.0.

So, if the cedet 1.0 suite is loaded then the min- and max-version of semantic, eieio and speedbar (mentioned in the Requirements-section of the file '`README`') have no relevance! If the new cedet 1.0 suite should be used then just install and load cedet 1.0 like described in the cedet-installation-instructions and go one with step 3. But ensure you have loaded - as described in the cedet-'`INSTALL`'-file - the file '`/path/to/cedet/common/cedet.el`' because otherwise cedet is not properly installed and ECB can not savely recognize that the new cedet-suite is loaded and should be used.

1. Download and unpack the ECB archive (probably you have already done this :-)
2. Read the file '`README`' in the ECB-directory and install the required semantic-, eieio- and speedbar-version[2].

   Please note: ECB maybe requires a newer version of these libraries than shipped with (X)Emacs. You have to install exactly a version ECB requires and also to make sure that the correct version is loaded into (X)Emacs!

   But ECB performs two autom checks:

   - At load-time: It checks if the packages semantic, eieio and speedbar are at least installed so ECB can be loaded. If not it offers to download and install them.
   - At start-time: It checks if the correct versions of semantic, eieio and speedbar are installed and gives you proper feedback. See ⟨undefined⟩ [Download required packages], page ⟨undefined⟩.

   So if you are not sure if you have installed the required packages at all or if you have installed the correct versions of these packages then do not worry about this, just go on with the following installation steps: If ECB is missing something it will give you proper feedback and support not later than at load-time or start-time!
3. Add the new ECB-directory to your `load-path` variable.

   You **MUST** add the ECB-install-directory to the `load-path` either by changing the `load-path` variable directly in your '`.emacs`' or '`site-lisp/site-start.el`' or by working with a file '`subdirs.el`'[3].

   So for example the needed entry for your '`.emacs`'-file could be:

[2] The speedbar-version shipped with GNU Emacs <= 21.3 does not satisfy the requirements for this feature - download a newer one!

[3] This works at least for Emacs but XEmacs may have slightly different mechanisms; see the XEmacs documentation

```
(add-to-list 'load-path
             "/path/to/your/ecb/installation/directory")
```

ATTENTION: ECB is NOT properly installed if it's directory is not added to `load-path` and for example just loaded by

```
(load-file "/path/to/ecb/ecb.el")
```

Do not do this!

4. Load ECB by adding code to your '`.emacs`':

   If you want to load the complete ECB at (X)Emacs-loadtime (Advantage: All ECB-options available after loading ECB. Disadvantage: Increasing loadtime[4]):

   ```
   (require 'ecb)
   ```

   If you want to load the ECB first after starting it by `ecb-activate` (Advantage: Fast loading[5]. Disadvantage: ECB- and semantic-options first available after starting ECB):

   ```
   (require 'ecb-autoloads)
   ```

   This loads all available autoloads of ECB, e.g. `ecb-activate`, `ecb-minor-mode`, `ecb-byte-compile` and `ecb-show-help`.

   Regardless which method you prefer: In both cases the used statement must be placed **after** the statement of step 3!

5. Restart (X)Emacs.

ECB is now ready for use and can be activated by calling `M-x ecb-activate` or `ecb-minor-mode`. Now you can either starting using ECB or you can do these optional installation steps:

6. Reading the online help with `ecb-show-help`

   Maybe you are interested to read the online-help of ECB before first start.

7. Bytecompiling ECB with `ecb-byte-compile`

   This byte compiles ECB. You can safely ignore all messages if there are any. (You can also bytecompile ECB from the command-line either by using the '`Makefile`' or by using the batch-file '`make.bat`'; just read the comments in that file you choose.)

8. Installing the Info-help of ECB

   The ECB distribution contains a subdirectory '`info-help`' which contains the online-help of ECB in Info-format. You can install this online help so it's available in the Top-directory of Info. There are two ways to do this:

   – Use "install-info" (recommended):

     1. Copy the files of the subdirectory '`info-help`' into the info-directory of Emacs
     2. Install the file '`info-help/ecb.info`' with the command "install-info" (if available on your system) in the '`dir`'-file.

   The supplied '`Makefile`' offers a target `install-help` which does both of these steps. You have just to call `make install-help` with the correct EMACSINFOPATH set (see the comment in '`Makefile`'). Here is an example:

[4] Cause of full loading of ECB itself and also the packages semantic, eieio and speedbar regardless if ECB is started.

[5] ECB, semantic, eieio and speedbar are first loaded after starting ECB or with other words: ECB and semantic are not loaded if you do not use/need them

```
make EMACSINFOPATH=/path/to/emacs/info install-help
```

– Manual Installation:

  Copy the files of the subdirectory '`info-help`' into the info-directory of Emacs and modify the file '`dir`' manually.

But it doesn't matter if you do not execute this step (8.) because the online help of ECB is always available though, see `ecb-show-help` (see ⟨undefined⟩ [Interactive ECB commands], page ⟨undefined⟩).

1.2 How to set up Emacs for file parsing with ECB

Please note: Normally it should not necessary for you to bother with the following stuff unless you have problems getting ECB working correctly for you.

1.2.1 General hints for a correct setup

ECB is for browsing files and therefore you have to setup your Emacs-configuration properly so the file-parsing engines like semantic, imenu or etags can be activated automatically for parsing your Emacs-Lisp, C, C++ or Java buffers[6]. For this Emacs must activate the correct `major-mode` for the source-files and Emacs can only do this if the option `auto-mode-alist` is setup correctly. The correct major-modes and possible file-extensions[7] are:

Language	**Major-mode**	**Extension(s)**
Emacs Lisp	`emacs-lisp-mode`	.el
C	`c-mode`	.h, .c
C++	`c++-mode`	.h, .hxx, .hh, .HH, .cxx, .cpp, .cc, .CC
Java	`java-mode` or `jde-mode` (if you use JDEE)	.java

Example: If you want files with extension ".cpp" being c++-parsed by semantic and ECB, your `auto-mode-alist` must contain an entry like:

```
("\\.cpp\\'" . c++-mode)
```

After this ECB will correctly parse your ".cpp"-sources and display all the parsing information in the ECB-methods buffer.

1.2.2 Setting up semantic

To ensure ECB and semantic are working correctly for all by semantic supported languages you have to pay attention to the following aspects concerning your Emacs-setup:

1. Setting up semantic itself

   For all semantic-supported file-types parsing files is done completely by semantic. ECB just displays the parsing results. For all needs of ECB semantic is completely setup by

[6] semantic supports some more "languages" like Makefiles etc. but these are the most important ones.

[7] Especially for C++ and C you can use any extension you want but these are the most common ones!

ECB itself, i.e. ECB sets up semantic for you! You have only to add the installation directory of semantic to your `load-path` (in an appropriate way)!

Please note: If you setup semantic for yourself following the recommendations in the installation instructions of semantic then you have probably added code to your startup-file like:

```
(setq semantic-load-turn-everything-on t)
(require 'semantic-load)
```

Be aware that this also enables the minor-modes `semantic-show-dirty-mode` and `semantic-show-unmatched-syntax-mode` where the former one highlights all code which has to be reparsed with dark background (which results in large portions of dark background ;-) and the latter one underlines all syntax which can not be parsed. Especially the former one can be really annoying.

To switch off these modes you can add to your startup-file:

```
(global-semantic-show-dirty-mode -1)
(global-semantic-show-unmatched-syntax-mode -1)
```

2. Checking your hooks

   If you have already checked point (1.) and if you have still problems getting ECB/semantic working properly for your sources you should check the related major-mode hook. Every major-mode X has a hook with name "X-hook" which is evaluated after activating the major-mode (see above, 2.), e.g. the hook for the major-mode `c++-mode` is `c++-mode-hook`.

   Semantic adds automatically during load-time a special "semantic-setup" to these major-mode hooks[8] in form of a "setup-function". Example: For c and c++ modes semantic adds `semantic-default-c-setup` to `c-mode-hook` and `c++-mode-hook`.

   If your own Emacs-setup (e.g. in '`.emacs`' or '`site-lisp/site-start.el`') overwrites such a major-mode-hook then semantic can not be activated for this major-mode and in consequence ECB can not work properly too!

   Check if your Emacs-setup uses somewhere `setq` for adding code to a major-mode-hook. **IMPORTANT**: Use `add-hook` instead of `setq`[9]!

If your source-files are "running" with correct `major-mode` and correct major-mode hooks ECB and semantic will do what you expect them doing!

1.2.3 Setup for file types not supported by semantic

From version 1.94 on ECB supports also parsing and displaying file-contents for file-types not supported by semantic (i.e. there is no semantic-grammar available for such file-types).

Such non-semantic file-types can often be parsed by imenu and/or etags. Both of these parsing methods are now supported: ECB can display the results of imenu and/or etags in its Method-buffer. ECB uses for this speedbar-logic. Therefore the following speedbar options takes effect for this feature:

– `speedbar-dynamic-tags-function-list`

[8] Of course only for major-modes supported by semantic!

[9] `setq` replaces/overwrites the current value of a hook with the new value whereas `add-hook` **adds** the new value to the old-value of the hook!

- `speedbar-tag-split-minimum-length`
- `speedbar-tag-regroup-maximum-length`
- `speedbar-tag-hierarchy-method`

Normally there should be no need for you to bother with these options, because the default values are suitable for most situations! But if you are not satisfied with the parsing/display results then you can change some of these options.

1.3 First steps after activating ECB first time

This section of the ECB online-help is displayed automatically by ECB after activating ECB first time and describes what are the first basic steps:

1. Configure where ECB can find your sources:

   Call `M-x customize-option RET ecb-source-path RET`[10]. This lets you customize the option `ecb-source-path` with the customize-feature of Emacs. This opens a customize-buffer where you can insert all the directories where ECB can find your source-files. Save your changes with the button "Save for future sessions" and then throw away this customize-buffer either by killing it with `M-x kill-buffer` or clicking at the button "Finish".
2. Take a look at the most important options of ECB Call `M-x ecb-customize-most-important RET` and see a list of options which you should at least know that these options exist.
3. Read the online-help of ECB:

   The online-help of ECB is available via
   - calling `M-x ecb-show-help`,
   - pressing *`C-c . h`* or
   - using the menu "ECB".

   (The section you are currently reading is part of the online-help of ECB)

   The chapter "Tips and tricks" is also very interesting!
4. Start working with ECB.

[10] This means first hitting the keys *`M`* (Meta- or Alt-key) and *`x`* simultaneously, inserting "customize-option" in the minibuffer, hitting RETURN, inserting "ecb-source-path" in the minibuffer and finally hitting RETURN again

2 Overview

ECB is a global minor-mode which offers a couple of *ECB-windows* for browsing your sources comfortable with the mouse and the keyboard. These "special" windows are also called *interactors* in this manual.

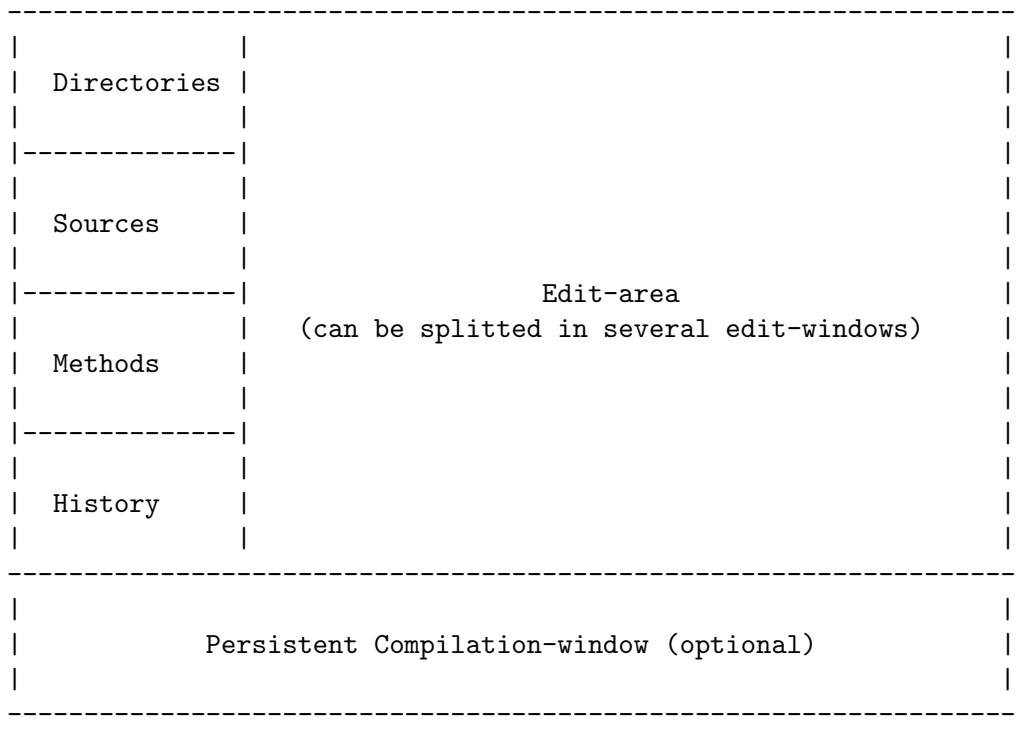
ECB offers some basic interactors to browse your sources:

- ECB-Directories for browsing directories
- ECB-Sources for browsing source-files and a file-history
- ECB-Methods for browsing the contents of a source

See ⟨undefined⟩ [Basic interactors], page ⟨undefined⟩ for a detailled description what these basic interactors offer. See ⟨undefined⟩ [ECB-interactors], page ⟨undefined⟩ for a general introduction in the interactor-concept of ECB.

In addition to these "special" ECB-windows you have always an *edit-area* where you can edit your source-files. The edit-area can be divided into several *edit-windows* - as many as you need (see ⟨undefined⟩ [The edit-area], page ⟨undefined⟩). And at the bottom of the ECB-frame a persistent *compilation-window* (also called *compile-window*) can be displayed (optional), where all the output of Emacs-compilation (compile, grep etc.) is shown (see ⟨undefined⟩ [Temp- and compile-buffers], page ⟨undefined⟩).

The following "screenshot" illustrates the typical layout of the ECB-frame[1]:



[1] This is only one example of the layouts ECB offers, see ⟨undefined⟩ [Changing the ECB-layout], page ⟨undefined⟩

3 How to use this manual

IMPORTANT: You should have read the chapter ⟨undefined⟩ [Overview], page ⟨undefined⟩ before going on reading the current chapter.

This chapter describes shortly the contents of the following chapters of this manual so maybe you can find faster what you are searching for.

All interactors of ECB (see ⟨undefined⟩ [ECB-interactors], page ⟨undefined⟩)
: Gives an introduction into the concept of interactors of ECB, what they are, which different types exist, how they look, how they can be used and last but not least what they do in detail, i.e. explains every interactor of ECB in detail.

Activation and Deactivation (see ⟨undefined⟩ [Activation and Deactivation], page ⟨undefined⟩)
: Describes how to activate and deactivate ECB and which different types of activation are possible.

Usage of ECB (see ⟨undefined⟩ [Usage of ECB], page ⟨undefined⟩)
: Describes in detail how to use ECB with mouse and keyboard, explains the concept of the edit-area and the persistent compile-window, describe how to change the window-layout and hide and show special windows, which stealthy background-tasks ECB performs and so on...

Customizing ECB (see ⟨undefined⟩ [Customizing], page ⟨undefined⟩)
: Gives an overview of the most important options of ECB and tell you something about do's and don'ts concerning customization of ECB. Lists all options of ECB and describe exactly the purpose of them.

Submitting a problem report (see ⟨undefined⟩ [Submitting problem report], page ⟨undefined⟩)
: Instructions what to do if you encounters problems with ECB.

Upgrading and downloading ECB (see ⟨undefined⟩ [Upgrading], page ⟨undefined⟩)
: Gives an introduction into the automatic option-upgrading-feature of ECB and you can download and install a newer ECB-version from within ECB.

Tips and tricks (see ⟨undefined⟩ [Tips and tricks], page ⟨undefined⟩)
: How to deal with special situations and circumstances, so e.g. working with big and small screens, working with large directories, using the builtin version-control-support of ECB, parsing non-semantic-supported sources like perl, using hide-show, working best with eshell and JDEE and some window-managers of Emacs (like escreen or winring) and many more...

Entry points for Elisp programmers (see ⟨undefined⟩ [Elisp programming], page ⟨undefined⟩)
: All informations needed by programmers when using some ECB-concepts from within other elisp-libraries. This includes a full description how to use the ECB-independent library tree-buffer.el. Lists also all available hooks and describes with full working example how to program own layouts.

Conflicts and bugs of ECB (see ⟨undefined⟩ [Conflicts and bugs], page ⟨undefined⟩)
: Lists all currently known bugs and problems and gives well working work-arounds.

4 All interactors of ECB

ECB displays a number of informational windows that allow for easy source code navigation and overview. These informational windows are called *interactors*. Each interactor is displayed in its own special window/buffer which is dedicated and read-only.

There are some "basic" interactors (e.g. for browsing directories and sources) and some "add-on" interactors for special purposes like displaying the definition of the current symbol under point. This chapter describes all interactors of ECB in detail.

4.1 The basic interactors of ECB

ECB offers basic interactors for browsing directory-structures, files of a directory and contents of source-files (e.g. methods and variables). These basic interactors are build from a special class of interactors, called *tree-buffer*. See ⟨undefined⟩ [Tree-buffer basics], page ⟨undefined⟩ for more details about the functionality of tree-buffers. See ⟨undefined⟩ [Tree-buffer styles], page ⟨undefined⟩ to get an impression about the look&feel of these tree-buffers.

In the following subsections these basic interactors of ECB will be explained in detail.

4.1.1 General introduction into tree-buffers

ECB displays most of its informations (e.g. about directory-structures or file-contents) in so called *tree-buffers* which means the display of such a tree-buffer is structured in a tree consisting of *tree-nodes*. Every line in a tree-buffer displays exactly one tree-node. Each node can have any arbitrary number of *children*-nodes. If a tree-node has no children then it is called a *leaf*.

Each tree-buffer of ECB is displayed in an own special ECB-window/buffer which is read-only ie. not editable.

The difference between a natural tree like a fir and an ECB-tree is that the root(-node) of a tree-buffer is not visible but only its children. In the example below the nodes parent-node-1 and parent-node-2 are the children of the invisible root-node.

If a tree-node contains at least one child it is displayed with a special expand/collapse-symbol (see the example below). This symbol allows expanding (rsp. collapsing) the tree-node wheras expanding means to display the children-nodes and collapsing means to hide the childrens of a tree-node.

Here is an example of a tree-buffer:

```
[+] parent-node-1  -------.
[-] parent-node-2  -------|
    [-] expanded  --------|
        leaf-node-1  -----|
        leaf-node-2  -----|-----[tree-nodes]
        leaf-node-3  -----|
        leaf-node-4  -----|
    [+] collapsed  -------
     |
     `-----------------[expand/collapse-symbol]
```

In most cases an action is triggered when clicking with the mouse onto a tree-node[1] (e.g. clicking onto “leaf-node-1” or “parent-node-1” in the example above). Which actions depends on the type of the tree-buffer. For example clicking on a tree-node in the ECB-sources-buffer (which is the name of a source-file) opens the relelated file in the edit-area of ECB (see ⟨undefined⟩ [ECB Sources-buffer], page ⟨undefined⟩) whereas clicking onto a node in the ECB-methods-buffer (which is the name of a tag in the current source-file displayed in the edit-area) “jumps” to the location of this tag in the source-buffer in the edit-area (see ⟨undefined⟩ [ECB Methods-buffer], page ⟨undefined⟩).

Almost every interactor of ECB offers a special popup-menu when clicking with the right mouse-button (of course also possible via keyboard, see ⟨undefined⟩ [Using the keyboard], page ⟨undefined⟩) onto a tree-node (e.g. some senseful actions possible for directory-nodes like grepping this directory or performing version-control actions for this directory or something else).

See ⟨undefined⟩ [ECB Directories-buffer], page ⟨undefined⟩, ⟨undefined⟩ [ECB Sources-buffer], page ⟨undefined⟩, ⟨undefined⟩ [ECB Methods-buffer], page ⟨undefined⟩ and ⟨undefined⟩ [Add-on interactors], page ⟨undefined⟩ for a detailed description which actions are triggered and which popup-menus are offered in all the interactors of ECB.

4.1.2 Displaying the trees with different styles

ECB offers three different styles for the tree-buffers in the ECB-windows. Two of the styles are ascii-based and one style uses images for drawing the tree-structure.

4.1.2.1 Basic knowledge about the styles

There are nine image-names which define the control- and guide-symbols to draw the tree. Here is the list of the allowed image-names and the related corresponding ascii-symbols:

- open (“[-]”): The control-symbol displayed for an opened tree-node which has several subnodes. Clicking onto this control closes the node.
- close (“[+]”): The control-symbol displayed for a closed tree-node, i.e. an expandable node with subnodes but all subnodes are hidden. Clicking onto this control opened the node and displays its subnodes - if there are any. If it has no subnodes the empty-symbol will be displayed.
- empty (“[x]”): The symbol displayed for an empty node. An empty node is a node which could have subnodes but has currently none.
- leaf (“*”): The symbol displayed for a node which can not have any subnodes so it is a “leaf” in the tree.
- guide (“ |”): The symbol used for drawing vertical “guide-lines” for opened nodes. See the example below.
- no-guide (“ ”): Sometimes invisible guide-lines are needed to draw the tree.
- end-guide (“ `”): The symbol used for the guide-line of the last subnode of an opened node.
- handle (“-”): The symbol displayed before every subnode. Each handle is connected to a guide-line - either a normal guide or an end-guide.

[1] Of course using the keyboard is also possible, see ⟨undefined⟩ [Using the keyboard], page ⟨undefined⟩.

– no-handle (“ ”): An invisible handle.

A tree will be build-up with these elements like follows:

```
[-] node-with-subnodes      (open)
 |-[+] not-empty-subnode1   (guide+handle+close)
 |-[x] empty-subnode        (guide+handle+empty)
 `-[-] not-empty-subnode2   (end-guide+handle+open)
    |-* leaf-1              (no-guide+no-handle+guide+handle+leaf)
    `-* leaf-2              (no-guide+no-handle+end-guide+handle+leaf)
```

4.1.2.2 How to customize the ascii-styles

The ECB-option `ecb-tree-buffer-style` offers two different styles completely drawn with ascii-controls and -guides.

Ascii-style with guide-lines (value `ascii-guides`)[2]:

```
[-] ECB
 |  [+] code-save
 `- [-] ecb-images
     |  [-] directories
     |   |  [-] height-15
     |   |   |  * close.xpm
     |   |   |  * empty.xpm
     |   |   |  * leaf.xpm
     |   |   `- * open.xpm
     |   |  [+] height-17
     |   |  [+] height-19
     |   `- [+] height-21
     |  [x] history
     |  [x] methods
     `- [x] sources
```

Ascii-style without guide-lines (value `ascii-no-guides`) - this is the style used by ECB <= 1.96:

[2] For a better look&feel of such a tree-buffer ECB displays only the last subnode of an opened node with a handle!

```
[-] ECB
    [+] code-save
    [-] ecb-images
        [-] directories
            [-] height-15
                * close.xpm
                * empty.xpm
                * leaf.xpm
                * open.xpm
            [+] height-17
            [+] height-19
            [+] height-21
        [x] history
        [x] methods
        [x] sources
```

The tree-layout of both ascii-styles can be affected with the options `ecb-tree-indent` and `ecb-tree-expand-symbol-before` (the examples above have set 4 for the former and true for the latter one). For the guide-style the face and color of the guide- and handle-symbols can be customized with the option `ecb-tree-guide-line-face` (default is the equal-named face).

4.1.2.3 Which images are used for the tree

Depending on the value of `ecb-tree-buffer-style` and the image-support of (X)Emacs, the tree-buffer try to use images instead of strings to draw a nice-looking tree.

If images can and should be used then the option `ecb-tree-image-icons-directories` tells ECB where to search for suitable image-icons for each of the nine image-names (see above). An image is used for displaying a control with name "XXX" if one of the directories of `ecb-tree-image-icons-directories` contains an image-file with basename "ecb-XXX" and an extension which is supported by (X)Emacs. Currently supported extensions are ".xpm", ".png", ".gif", ".jpeg", ."jpg" and ".xbm".

Example: To display the control with name "open" with a suitable image then one of the directories of `ecb-tree-image-icons-directories` must contain a file with name "ecb-open.xpm" or "ecb-open.png" etc. See the description of this option to get all important details how and in which sequence ECB searches the directories of `ecb-tree-image-icons-directories`.

ECB comes with predefined default-images usable for every tree-buffer and special images for the Directories- and the Methods-tree-buffer. They are defined in several different heights - so for the most senseful font-heights of a tree-buffer a fitting image-size should be available. The shipped images reside either in the subdirectory `"ecb-images"` of the ECB-installation or - if ECB is installed as regular XEmacs-package - in the ECB-etc data-directory (the directory returned by evaluating (locate-data-directory `"ecb"`). If you do not want to change the images then you normally have nothing to do because the default value of `ecb-tree-image-icons-directories` points already to the correct image-directories.

A special remark for XEmacs:

At least XEmacs 21.14 (but probably previous versions too) has a bug in its display-engine which prevents adjacent images to be displayed correctly. The effect is, that in a

row of two or more adjacent images (e.g. end-guide+handle+open - see the tree-example above) always all images are masked by the last one, means only the last one is visible. If at least one normal character (e.g. a space) is placed between two images then the images are displayed correctly. Therefore ECB has implemented the following work-around to get best possible results with XEmacs: open-, close-, empty-, leaf-, guide-, end-guide- and no-guide-images are displayed with images and the handle- and the no-handle-images are displayed with the corresponding ascii-symbols (which is "-" rsp. " "). The face (the color) of the handle-symbol is customizable via the option `ecb-tree-guide-line-face`.

This bug is already reported to the XEmacs-team. If your XEmacs has fixed this bug then add the following to your '`.emacs`'-file (or whereever your emacs-setup is located):

```
(setq tree-buffer-enable-xemacs-image-bug-hack nil)
```

Then ECB uses images without any special work-around with XEmacs too. Just try it - if the tree-buffers look ugly then the XEmacs-bug is probably not fixed correctly.

4.1.2.4 Special images for the Methods-buffer

ECB can display all the semantic-tags in the Method-buffer with special icons for methods, variables and classes - each of them with a different icon dependend of the protection of the tag. This feature can be disabled/enabled via the option `ecb-display-image-icons-for-semantic-tags`. All the special images are located in that directory where the option `ecb-tree-image-icons-directories` point to for methods.

4.1.3 ECB Directories-interactor

The ECB directories interactor is for browsing directories. The direct children of the invisible root-node are called *source-path* and can be defined with the option `ecb-source-path`. Each source-path is the starting-node of the complete directory-structure below this path and can be browsed with the directories-interactor.

When a sources interactor is contained in the current layout then per default only directories and subdirectories are displayed in the directories tree-buffer (the source-files are displayed in the sources tree-buffer - see ⟨undefined⟩ [ECB Sources-buffer], page ⟨undefined⟩) but this can be changed with the option `ecb-show-sources-in-directories-buffer`.

4.1.3.1 Usage of the directories interactor

- Select directories (and - if enabled - source files) in the *ECB-Directories* buffer by clicking a mouse button on the directory name or by hitting RETURN when the cursor is placed on the item line, see ⟨undefined⟩ [Usage of ECB], page ⟨undefined⟩.

  IMPORTANT: If you use the POWER-click (i.e. hold down the SHIFT-key while clicking with the primary mouse button (see ⟨undefined⟩ [Using the mouse], page ⟨undefined⟩) or RETURN (see ⟨undefined⟩ [Using the keyboard], page ⟨undefined⟩)) on a directory node in the this buffer then the directory-contents-cache for this directory will be refreshed and actualized.

- Directory names with a "[+]" symbol after (or before) them can be expanded/collapsed by clicking on the symbol, pressing the TAB key (see ⟨undefined⟩ [Using the keyboard], page ⟨undefined⟩) when the cursor is placed on the package line or clicking a mouse button on the item, see ⟨undefined⟩ [Using the mouse], page ⟨undefined⟩.

- Right clicking on an item will open a popup menu where different operations on the item under the mouse cursor can be performed. This popup-menu offers operations for version-control, dired, grep, some file-operations like creating a directory and commands to make a directory a source-path in the sense of `ecb-source-path`.
- Pressing F2 will open the ECB customization group (see ⟨undefined⟩ [Customizing], page ⟨undefined⟩) in the edit window. F3 shows the online help in the edit-window. Pressing F4 in the ECB-directories buffer will offer adding a new source-path.

When source-files-nodes are displayed in the directories-buffer (see `ecb-show-sources-in-directories-buffer`) then for these nodes all descriptions of section ⟨undefined⟩ [ECB Sources-buffer], page ⟨undefined⟩ are valid.

4.1.3.2 Activating/Displaying the directories interactor

Either use one of the predefined layouts which contain the directories interactor (see ⟨undefined⟩ [Changing the ECB-layout], page ⟨undefined⟩) (e.g. via `C-c . l c`) or create a new ecb-layout via the command `ecb-create-new-layout` and add a buffer of type "directories" into this new layout (see ⟨undefined⟩ [Creating a new ECB-layout], page ⟨undefined⟩).

4.1.3.3 Customizing the directories interactor

See ⟨undefined⟩ [ecb-directories], page ⟨undefined⟩ for a list of all options currently available for customizing this interactors to your needs.

4.1.4 ECB Sources- and history-interactor

ECB offers two interactors for displaying source-file-names: The sources- and the history-interactor. The former one displays all source-file names of the currently selected directory of the directories-interactor (see ⟨undefined⟩ [ECB Directories-buffer], page ⟨undefined⟩) whereas the latter one displays the names of all currently loaded source-files regardless in which directory they reside so it works as a "history" of source-files.

Both the sources- and the history-tree-buffer are "flat" tree-buffers means all nodes are direct children of the invisible root-node and can not be expanded.

4.1.4.1 Usage of the sources/history interactor

- Source files can be selected by clicking with the primary mouse button (see ⟨undefined⟩ [Using the mouse], page ⟨undefined⟩) or hitting RETURN (see ⟨undefined⟩ [Using the keyboard], page ⟨undefined⟩) on the source row in the *ECB-Sources* or *ECB-History* windows. The buffer of the selected source-file will be displayed in an edit-window - which one depends on the setting in `ecb-mouse-click-destination`.

  IMPORTANT: If you use the POWER-click (i.e. hold down the SHIFT-key while clicking with the primary mouse button (see ⟨undefined⟩ [Using the mouse], page ⟨undefined⟩) or RETURN (see ⟨undefined⟩ [Using the keyboard], page ⟨undefined⟩)) on a source row in the ECB-Sources or ECB-History windows then the source will not be displayed in an edit-window but it will be scanned in the background and all its contents (e.g. methods and variables) are listed in the *ECB Methods* window (see ⟨undefined⟩ [ECB Methods-buffer], page ⟨undefined⟩. So you can get an overlook over the source without changing the buffer in the edit-window.

- Clicking on the source file with the secondary mouse button or C-RETURN (see ⟨undefined⟩ [Usage of ECB], page ⟨undefined⟩) will open the source file in another edit window - which one depends on the setting in `ecb-mouse-click-destination`.
- Right clicking on a source file (mouse-button 3) will open a popup menu where different operation on the item under the mouse cursor can be performed. This popup-menu offers operations for version-control, dired, grep, filtering the file-names and some file-operations like deleting the related file from disk.

4.1.4.2 Activating/Displaying the sources/history interactor

Either use one of the predefined layouts which contain the sources (rsp. history) interactor (see ⟨undefined⟩ [Changing the ECB-layout], page ⟨undefined⟩) (e.g. via *C-c . l c*) or create a new ecb-layout via the command `ecb-create-new-layout` and add a buffer of type "sources" (rsp. "history") into this new layout (see ⟨undefined⟩ [Creating a new ECB-layout], page ⟨undefined⟩).

4.1.4.3 Customizing the sources/history interactor

See ⟨undefined⟩ [ecb-sources], page ⟨undefined⟩ and ⟨undefined⟩ [ecb-history], page ⟨undefined⟩ for a list of all options currently available for customizing these interactors to your needs.

4.1.5 The ECB Methods interactor

The *ECB-Methods* interactor contains all parsed and recognized tags of the current source-buffer. It is called "Method-buffer" because ECB is mostly designed for browsing sourcecode files and for programming-languages these tags are often methods (and variables etc.) To simplify explanations we talk in the following only about methods and variables - but in general the method-buffer can contain any kind of tags (e.g. sections and subsections for texinfo buffers).

Per default the content of the methods-interactor is automatically synchronized and updated with current point of the current source-buffer in the edit-area (see `ecb-window-sync` and ⟨undefined⟩ [ECB-window synchronizing], page ⟨undefined⟩).

4.1.5.1 Usage of the methods interactor

- When a method/variable is selected with the primary mouse button (see ⟨undefined⟩ [Using the mouse], page ⟨undefined⟩) or RETURN (see ⟨undefined⟩ [Using the keyboard], page ⟨undefined⟩) the buffer in the edit-window (which one depends on the setting in `ecb-mouse-click-destination`) will jump to the method/variable.

  IMPORTANT: If you use the POWER-click (i.e. hold down the SHIFT-key while clicking with the primary mouse button (see ⟨undefined⟩ [Using the mouse], page ⟨undefined⟩) or RETURN (see ⟨undefined⟩ [Using the keyboard], page ⟨undefined⟩)) on a node in this buffer then the edit-buffer will be narrowed to the selected tag (see also option `ecb-tag-visit-post-actions`). But this works only for sources parsed by semantic, not by imenu or etags!
- Clicking on a method/variable with the secondary mouse button or C-RETURN (see ⟨undefined⟩ [Usage of ECB], page ⟨undefined⟩) will jump to the method in another edit window - which one depends on the setting in `ecb-mouse-click-destination`.

- Right clicking on a method/variable will open a popup menu where different operation on the item under the mouse cursor can be performed. The popup-menu offers commands for filtering the displayed tree-nodes, hiding/narrowing the related tags in the source-buffer and expanding/collapsing one/all tree-nodes according to a certain expansion-level.

4.1.5.2 Activating/Displaying the methods interactor

Either use one of the predefined layouts which contain the methods interactor (see ⟨undefined⟩ [Changing the ECB-layout], page ⟨undefined⟩) (e.g. via `C-c . l c`) or create a new ecb-layout via the command `ecb-create-new-layout` and add a buffer of type "methods" into this new layout (see ⟨undefined⟩ [Creating a new ECB-layout], page ⟨undefined⟩).

4.1.5.3 Customizing the methods interactor

See ⟨undefined⟩ [ecb-methods], page ⟨undefined⟩ for a list of all options currently available for customizing this interactor to your needs.

4.2 Add-on interactors of ECB

This chapter gives detailled informations about available add-on interactors. This includes basic desciptions what they do as well as how to use them.

4.2.1 Displaying the current semantic context

The cedet-suite contains the *semantic analyzer* which is a library tool that performs context analysis and can derive useful information. See the related node in the info-manual of cedet/semantic for more detailed informations about this tool.

The analyzer output can be used through a special ECB-interactor. This interface lists details about the analysis, such as the current function, local arguments and variables, details on the prefix (the symbol the cursor is on), and a list of all possible completions (see `semantic-analyze-possible-completions` for more details about completions available via the semantic-analyser).

Per default the content of the analyser-interactor is automatically synchronized and updated with current point of the current source-buffer in the edit-area (see `ecb-window-sync` and ⟨undefined⟩ [ECB-window synchronizing], page ⟨undefined⟩).

The analyser-interactor is of type tree-buffer. See ⟨undefined⟩ [Tree-buffer basics], page ⟨undefined⟩ for basic informations how to use such a tree-buffer.

4.2.1.1 Usage of the analyser-interactor

- When a node-name in the "Context"-bucket is selected with the primary mouse button (see ⟨undefined⟩ [Using the mouse], page ⟨undefined⟩) or RETURN (see ⟨undefined⟩ [Using the keyboard], page ⟨undefined⟩) the buffer in the edit-window (which one depends on the setting in `ecb-mouse-click-destination`) will jump to the related entry. For strongly typed languages, this means you will jump to the definition of the variable, slot, or type definition.
- Clicking on a node-name in the "Context"-bucket with the secondary mouse button or C-RETURN (see ⟨undefined⟩ [Usage of ECB], page ⟨undefined⟩) will jump to the

related entry in another edit window - which one depends on the setting in `ecb-mouse-click-destination`.

- If you click on a node-name in the "Completions"-bucket, then the text that was recently analyzed will be replaced with the name of the tag that was clicked on in the analyser-interactor.
- Right clicking with the mouse (or with the keyboard, see ⟨undefined⟩ [Using popup-menus], page ⟨undefined⟩) onto a tree-node opens a popup-menu which allows to display additional (if available) informations to the current node.

4.2.1.2 Interactive commands of the analyser-interactor

ECB offers the following commands for the analyser-interactor:

- `ecb-analyse-buffer-sync`
- `ecb-goto-window-analyse`
- `ecb-maximize-window-analyse`

See ⟨undefined⟩ [Interactive ECB commands], page ⟨undefined⟩ for details about these commands. But you should not have any need to call `ecb-analyse-buffer-sync` directly because ECB automatically syncronizes the analyser-interactor with current active edit-buffer.

4.2.1.3 Activating/Displaying the analyser-interactor

Either use one of the predefined layouts "left-analyse" or "leftright-analyse" (e.g. via *C-c . l c*) or create a new ecb-layout via the command `ecb-create-new-layout` and add a buffer of type "other" and name "analyse" into this new layout (see ⟨undefined⟩ [Creating a new ECB-layout], page ⟨undefined⟩).

4.2.1.4 Customizing the analyser interactor

See ⟨undefined⟩ [ecb-analyse], page ⟨undefined⟩ for a list of all options currently available for customizing this interactor to your needs.

4.2.2 Displaying the definition of the current symbol under point

5 Activation and Deactivation

This chapter describes all aspects of activating and deactivating ECB.

IMPORTANT: Regardless of the activation-type (standard or automatic) the activation-process of ECB is always completely failure-save. This means any error during any step of the activation-process forces a complete cleanup (e.g. removing hooks, disabling advices etc.) of all settings ECB did (e.g. adding hooks, activating advices etc.) until the failure. Therefore if ECB-activation stops cause of a failure then you can be sure that your Emacs has the same state as before the ECB-activation-start!

5.1 Standard activation and deactivation

Call *M-x ecb-activate* and *M-x ecb-deactivate* to activate or deactivate ECB[1]. If you want ECB started in a new frame see the option `ecb-new-ecb-frame` (default is nil). `ecb-activate` always raises and selects the ECB-frame even if ECB is already started.

Because ECB is a global minor-mode it can also be (de)activated/toggled by *M-x ecb-minor-mode*.

The following sequence of hooks is evaluated during activation of ECB:

1. `ecb-before-activate-hook`
2. <All actions for activation but no layout drawing>
3. `ecb-activate-before-layout-draw-hook`
4. `ecb-redraw-layout-before-hook`
5. <Drawing the layout>
6. `ecb-redraw-layout-after-hook`
7. `ecb-activate-hook`

The following sequence of hooks is evaluated during deactivation of ECB:

1. `ecb-before-deactivate-hook`
2. <All actions for deactivation of ECB>
3. `ecb-deactivate-hook`

5.2 Automatic activation and deactivation

There are two ways for auto. (de)activation ECB after Emacs-start and for different window-configurations.

`ecb-auto-activate`
: This option is for auto. activating ECB after Emacs-startup. Set this to not nil and ECB will automatically be started after Emacs comes up.

`window-manager support`
: The window-manager support of ECB deactivates ECB when going to another window-configuration and reactivates ECB when coming back to the ECB-window-configuration. For all details about this see ⟨undefined⟩ [Window-managers and ECB], page ⟨undefined⟩.

[1] Activation is also possible via the menu "Tools –> Start Code Browser (ECB)".

6 Usage of ECB

This chapter describes in a detailed manner all aspects of using the Emacs Code Browser.

6.1 Working with the mouse in the ECB-windows

Normally you get best usage if you use ECB with a mouse. ECB distinguishes between a *primary* and a *secondary* mouse-button.

With the option `ecb-primary-secondary-mouse-buttons` the following combinations of primary and secondary mouse-buttons are possible:

- primary: mouse-2, secondary: C-mouse-2[1]. This is the default.
- primary: mouse-1, secondary: C-mouse-1
- primary: mouse-1, secondary: mouse-2

If you change this during ECB is activated you must deactivate and activate ECB again to take effect.

6.1.1 The primary mouse-button

A click with the primary button causes the main effect in each ECB-buffer:

- ECB Directories: Expanding/collapsing nodes and displaying files in the ECB-Sources buffer.
- ECB sources/history: Opening the file in that edit-window specified by the option `ecb-mouse-click-destination`.
- ECB Methods: Jumping to the method in that edit-window specified by the option `ecb-mouse-click-destination`.

6.1.2 The POWER- or SHIFT-click

A click with the primary mouse-button while the SHIFT-key is pressed is called the POWER-click and does the following (depending on the ECB-buffer where the POWER-click occurs):

- ECB Directory: Refreshing the directory-contents-cache (see `ecb-cache-directory-contents`).
- ECB sources/history: Only displaying the source-contents in the method-buffer but not displaying the source-file in an edit-window. This means it opens the clicked source only in the background and shows all its methods/variables in ECB-Methods; the buffer of the edit-window is not changed! This is very useful to get only an overlook for a certain source.
- ECB Methods: Narrowing to the clicked method/variable/ect... (see `ecb-tag-visit-post-actions`). But this works only for sources parsed by semantic, not by imenu or etags!

Per default the complete node-name of an item in a tree-buffer is displayed in the echo-area if the mouse moves over it, regardless if the related window is the active one or not. You get the same effect always after a POWER-click. In general: Via `ecb-show-node-info-in-minibuffer` you can specify in a detailed manner for every ECB tree-buffer when and which node-info should be displayed in the minibuffer.

[1] means mouse-2 while CTRL-key is pressed.

6.1.3 The secondary mouse-button

The secondary mouse-button is for opening (jumping to) the file in another edit-window (see the documentation of the option `ecb-mouse-click-destination`).

6.1.4 The right mouse-button

In each ECB-buffer mouse-3 (= right button) opens a special context popup-menu for the clicked item where you can choose several senseful actions.

With the options `ecb-directories-menu-user-extension`, `ecb-sources-menu-user-extension`, `ecb-methods-menu-user-extension` and `ecb-history-menu-user-extension` you can add statically new commands to the popup-menus. See the docstring of `ecb-directories-menu-user-extension` for more details.

With the options `ecb-directories-menu-user-extension-function`, `ecb-sources-menu-user-extension-function`, `ecb-methods-menu-user-extension-function` and `ecb-history-menu-user-extension-function` you can add new commands to the popup-menus in a dynamic manner. See the docstring of `ecb-directories-menu-user-extension-function` for more details.

With the options `ecb-directories-menu-sorter`, `ecb-sources-menu-sorter`, `ecb-methods-menu-sorter` and `ecb-history-menu-sorter` you can even re-arrange all the entries of the popup-menus.

6.1.5 Horizontal scrolling with the mouse

In each tree-buffer of ECB you can easily scroll left and right with the mouse if the option `ecb-tree-easy-hor-scroll` is not `nil`.

The reason for this is: XEmacs has horizontal scroll-bars so invisible parts beyond the right window-border of a tree-buffer can always made visible very easy.

GNU Emacs does not have hor. scroll-bars so especially with the mouse it is quite impossible to scroll smoothly right and left. The functions `scroll-left` and `scroll-right` can be annoying and are also not bound to mouse-buttons.

ECB offers three ways for smoothly hor. scrolling with GNU Emacs if `ecb-tree-easy-hor-scroll` is a positive integer-value S:

- In all ECB-tree-buffers the keys *`M-mouse-1`* and *`M-mouse-3`* are bound to scrolling left rsp. right with scroll-step S.
- Clicking with mouse-1 or mouse-2 onto the edge of the modeline has the same effect, i.e. if you click with mouse-1 onto the left \(resp right) edge of the modeline you will scroll left \(resp. right) with scroll-step S.
- Additionally *`C-M-mouse-1`* and `C-M-mouse-3` are bound to scrolling left rsp. right with scroll-step `window-width` - 2.

This is NOT done for XEmacs cause of its horizontal scrollbars. If you want scrolling left and right with the mouse in XEmacs then activate the horizontal scrollbars.

6.2 Working with the keyboard in the ECB-windows

ECB offers the `ecb-mode-map` which binds the most important functions of ECB to keys so you can easily use ECB in every window[2] without a mouse.

[2] Regardless if a tree-window or an edit-window

IMPORTANT: Do not modify `ecb-mode-map` directly! The option `ecb-key-map` defines all ECB keybindings, including a common prefix-key (This is by default `C-c .`). If there are conflicts with other minor-modes or packages you can define very easy another prefix. Please read carefully the description of `ecb-key-map` (see ⟨undefined⟩ [ecb-general], page ⟨undefined⟩).!

The following sections describe special topics about using the keyboard in the ECB-tree-buffers:

6.2.1 Navigation and Selection in a tree-buffer

In the ECB-buffers RETURN and TAB are the most important keys:

- RETURN does the same as the primary button and C-RETURN does the same as the secondary button. S-RETURN is the same as the SHIFT-click or POWER-click. The terms "primary", "secondary", "SHIFT-" and "POWER-click" are explained in ⟨undefined⟩ [Using the mouse], page ⟨undefined⟩. See also the option `ecb-tree-do-not-leave-window-after-select` and the function `ecb-toggle-do-not-leave-window-after-select` which is bound to `C-t` in each tree-buffer of ECB!

  For all RETURN (and S-RETURN and C-RETURN) hits the position of the point within the current node-line is irrelevant! This is for conveniance.
- TAB always expands or collapses expandable nodes.

The RETURN and TAB keys can not be (re)defined with `ecb-key-map`!

If you set `ecb-tree-navigation-by-arrow` to not nil then the arrow keys work in the ECB tree-window in the following smart way:

- Left-arrow: If node is expanded then it will be collapsed otherwise (i.e. current node is either not expandable or not expanded) point jumps to the next "higher" node in the hierarchical tree (higher means the next higher level or - if no higher level available - the next higher node on the same level).
- Right-arrow: If node is expandable but not expanded then it will be expanded. Otherwise (i.e. current node is either not expandable or already expanded) point jumps to the next following node (which is the first subnode in case of an already expanded node or simply the next node in the following line).
- Up- and down-arrow: Point jumps to the first character of the previous (up) rsp. next node (down). "First" character means either the first character of the expand-symbol (in case `ecb-tree-expand-symbol-before` is not nil) or of the displayed node-name. Or with other words: The first non-indentation and non-guide-line (see `ecb-tree-buffer-style`) character of a node (see ⟨undefined⟩ [Tree-buffer styles], page ⟨undefined⟩).

6.2.2 Incremental search for a node in current tree-buffer

Each display-able key (e.g. all keys normally bound to `self-insert-command`) is appended to the current search-pattern. The tree-buffer tries to jump to the first node which matching the current search-pattern either as substring or as prefix (see below). If no match is found then nothing is done. There are some special keys:

- *`backspace`* and *`delete`*: Delete the last character from the search-pattern.
- *`home`*: Delete the complete search-pattern

- *end*: Expand either to a complete node if current search-pattern is already unique or expands to the greatest common substring or prefix of the nodes. If there are at least two nodes with the same greatest common-prefix than every hit of *end* jumps to the next node with this common prefix.

For better overlooking the current search-pattern is shown in the echo area. After selecting a node with RET the search-pattern is cleared out. With `ecb-tree-incremental-search` you can specify if the current search-pattern must be a real prefix of the node (default) or if any substring is matched.

For faster and easier finding the right node in a ecb-window the incremental search ignores the following non interesting stuff:

- any leading spaces
- expand/collapse-buttons: [+] rsp. [-]
- protection-signs (+, -, #) in the method-window if uml-notation is used
- variables types or return-types in front of variable- or method-names.
- const specifier for variables

This means: Just type in the prefix (rsp. a substring) of a class-, variable-, method-, directory- or filename and ECB will bring you as fast as possible to the node you want. Incremental node-search uses the value of `case-fold-search`.

Tip: The `ecb-minor-mode` offers you in the `ecb-mode-map` (customizable via `ecb-key-map`) some keys for selecting every window of the ecb-frame. This makes window-selection a childs play. For example you can jump into the method-window by hitting *C-c . gm*.

6.2.3 Adding personal keybindings for the tree-buffers

There are five hooks which can be used for adding personal keybindings to the ECB tree-buffers:

- `ecb-common-tree-buffer-after-create-hook`
- `ecb-directories-buffer-after-create-hook`
- `ecb-sources-buffer-after-create-hook`
- `ecb-methods-buffer-after-create-hook`
- `ecb-history-buffer-after-create-hook`

These hooks are called directly after tree-buffer creation so they can be used to add personal local keybindings[3] either to all tree-buffers (`ecb-common-tree-buffer-after-create-hook`) or just to a certain tree-buffer. Here is a list which keys MUST NOT be rebound:

- *RET* and all combinations with *Shift* and *Ctrl*: Used for selecting nodes in all tree-buffers.
- *TAB*: Used for expanding/collapsing nodes in all tree-buffers.
- *C-t*: Used for toggling "do not leave window after selection" in all tree-buffers, see command `ecb-toggle-do-not-leave-window-after-select`.

[3] To be more general: They can be used to run any arbitrary lisp code after a tree-buffer creation!

- All self-inserting characters: Used for easy and fast navigation in all tree-buffers, See ⟨undefined⟩ [Incremental search], page ⟨undefined⟩.
- *F2*, *F3*, *F4*: Used for customizing ECB, adding source-path in the directories buffer.

The following example binds *C-a* to some useful code in ALL tree-buffers and *C-d* to even more useful code ONLY in the directories buffer:

```
(add-hook 'ecb-common-tree-buffer-after-create-hook
          (lambda ()
            (local-set-key (kbd "C-a")
                           (lambda ()
                             (interactive)
                             (message "ECB is great!")))))

(add-hook 'ecb-directories-buffer-after-create-hook
          (lambda ()
            (local-set-key (kbd "C-d")
                           (lambda ()
                             (interactive)
                             (message "ECB is wonderful!")))))
```

6.2.4 Using the popup-menu of a tree-buffer from keyboard.

A popup-menu of a tree-buffer can be activated from keyboard with the command `tree-buffer-show-menu-keyboard` which is bound to *M-m* in every tree-buffer. How the popup-menu is displayed depends if this command is called with a prefix-argument or not:

If called without a prefix-arg then the popup-menu is displayed graphically as if it would be activated via mouse (for GNU Emacs this works perfectly but for XEmacs there is a bug which results in a wrong menu-position - but the menu itself works also with XEmacs).

If called with a prefix-arg (*C-u M-m*) then the popup-menu is displayed with the tmm-mechanism (like the Emacs-[menu-bar] is displayed when 'tmm-menubar' is called). This tmm-display is only available for GNU Emacs.

6.3 Working with the edit-window(s) of the edit-area

ECB offers you all what you need to work with the edit-area as if the edit-windows of the edit-area would be the only windows of the ECB-frame.

ECB advices the following user-commands so they work best with ECB:

- `balance-windows`
- `delete-other-windows`
- `delete-window`
- `delete-windows-on`
- `display-buffer`
- `shrink-window-if-larger-than-buffer`
- `split-window`
- `split-window-horizontally`
- `split-window-vertically`

– `switch-to-buffer`
– `switch-to-buffer-other-window`
– `other-window`
– `other-window-for-scrolling`

The behavior of the adviced functions is (slightly simplified):

- All these adviced functions behaves exactly like their corresponding original functions but they always act as if the edit-window(s) of ECB would be the only window(s) of the ECB-frame. So the edit-window(s) of ECB seems to be a normal Emacs-frame to the user. This means that you can split and delete edit-windows without any restriction - ECB ensures that neither the special ECB-windows nor the compile-window will be damaged.
- If there is a persistent compile-window (see ⟨undefined⟩ [Temp- and compile-buffers], page ⟨undefined⟩) then all compilation-buffers in the sense of `ecb-compilation-buffer-p` will be displayed in the compile-window.
- If called in another frame than the ECB-frame these functions behave exactly like the not adviced original versions!

Please note: All these advices are only enabled when ECB is enabled.

Another interesting option in the context of the edit-window and these adviced functions is `ecb-layout-always-operate-in-edit-window`!

6.3.1 Documentation of the adviced window functions

This section describes for every adviced window function (s.a.) how it differs from the original version. Only the differences are mentioned, so if you want the full documentation of such a function call `describe-function` or `C-h f`.

`other-window` *ARG* **&optional** *ALL-FRAMES* [Command]
Around-advice `ecb`: The ECB-version of `other-window`. Works exactly like the original function with the following ECB-adjustment: The behavior depends on `ecb-other-window-behavior`.

`delete-window` **&optional** *WINDOW* [Command]
Around-advice `ecb`: The ECB-version of `delete-window`. Works exactly like the original function with the following ECB-adjustment:

If optional argument WINDOW is nil (i.e. probably called interactively): If called in a splitted edit-window then it works like as if all the edit-windows would be the only windows in the frame. This means the current edit-window which contains the point will be destroyed and its place will be occupied from another one. If called in an unsplitted edit-window then nothing is done. If called in the compile-window of ECB then the compile-window will be hidden (like with `ecb-toggle-compile-window`). If called in an ecb-window of the current ECB-layout there are two alternatives:

- If the function is contained in `ecb-layout-always-operate-in-edit-window` the right edit-window is selected (depends on the value of the option `ecb-mouse-click-destination`) and does then its job.
- Otherwise the behavior depends on the value of the option `ecb-advice-window-functions-signal-error`.

If optional argument WINDOW is a living window (i.e. called from program): If WINDOW is an edit-window then this window is deleted, if WINDOW is the compile-window then it will be hidden and otherwise the behavior depends on `ecb-advice-window-functions-signal-error`.

`delete-other-windows` **&optional** *WINDOW* [Command]

Around-advice `ecb`: The ECB-version of `delete-other-windows`. Works exactly like the original function with the following ECB-adjustment:

If optional argument WINDOW is nil (i.e. probably called interactively): If called in a splitted edit-window then it works like as if all the edit-windows would be the only windows in the frame. This means all other edit-windows besides the current edit-window which contains the point will be destroyed and the current edit-window fills the whole edit-area. Neither the special ecb-windows nor the compile-window will be destroyed!

- If called in an unsplitted edit-window then either the compile-window will be hidden (if there is one) otherwise nothing is done.
- If called in one of the ecb-windows then the current one is maximized, i.e. the other ecb-windows (not the edit-windows!) are deleted.
- If called in the compile window there are two alternatives:
  - If the function is contained in `ecb-layout-always-operate-in-edit-window` the right edit-window is selected (depends on the value of the option `ecb-mouse-click-destination`) and then it does its job.
  - Otherwise the behavior depends on the value of the option `ecb-advice-window-functions-signal-error`.

If optional argument WINDOW is a living window (i.e. called from program): If WINDOW is an edit-window then this window is maximized (i.e. the other edit-window is deleted) if there are more at least 2 edit-windows otherwise the compile-window is deleted (if there is one). If WINDOW is an ecb-window then only the other ecb-windows are deleted and in all other cases the behavior depends on `ecb-advice-window-functions-signal-error`.

`delete-windows-on` *BUFFER* **&optional** *FRAME* [Command]

Around-advice `ecb`: The ECB-version of `delete-windows-on`. Works exactly like the original function with the following ECB-adjustment:

An error is reported if *BUFFER* is an ECB-tree-buffer. These windows are not allowed to be deleted.

`split-window` **&optional** *WINDOW SIZE HORFLAG* [Command]

Around-advice `ecb`: The ECB-version of `split-window`. Works exactly like the original function with the following ECB-adjustment:

If called for a not-edit-window in the `ecb-frame` it stops with an error if `split-window` is not contained in the option `ecb-layout-always-operate-in-edit-window`! Besides this (e.g. called for a window in another frame than the `ecb-frame`) it behaves like the original version.

`split-window-horizontally` [Command]
Around-advice `ecb`: The ECB-version of `split-window-horizontally`. Works exactly like the original function with the following ECB-adjustment:

If called in any other window of the current ECB-layout it stops with an error if this `split-window-horizontally` is not contained in the option `ecb-layout-always-operate-in-edit-window`!

`split-window-vertically` [Command]
Around-advice `ecb`: The ECB-version of `split-window-vertically`. Works exactly like the original function with the following ECB-adjustment:

If called in any other window of the current ECB-layout it stops with an error if this `split-window-vertically` is not contained in the option `ecb-layout-always-operate-in-edit-window`!

`display-buffer` *BUFFER* **&optional** *NOT-THIS-WINDOW FRAME* [Command]
Around-advice `ecb`: Makes this function compatible with ECB if called in or for the ecb-frame. It displays all buffers which are "compilation-buffers" in the sense of `ecb-compilation-buffer-p` in the compile-window of ECB. If the compile-window is temporally hidden then it will be displayed first.

If there is no compile-window (`ecb-compile-window-height` is nil) then it splits the edit-window if unsplitted and displays BUFFER in another edit-window but only if `pop-up-windows` is not nil (otherwise the edit-window will not be splitted).

All buffers which are not "compilation-buffers" in the sense of `ecb-compilation-buffer-p` will be displayed in one of the edit-area and `display-buffer` behaves as if the edit-windows would be the only windows in the frame.

If BUFFER is contained in `special-display-buffer-names` or matches `special-display-regexps` then `special-display-function` will be called (if not nil). But this behavior depends on the value of the option `ecb-ignore-special-display`. The values of `same-window-buffer-names` and `same-window-regexps` are supported as well.

See the option `ecb-ignore-display-buffer-function`!

If called for other frames it works like the original version.

`switch-to-buffer` *BUFFER* **&optional** *NORECORD* [Command]
Around-advice `ecb`: The ECB-version of `switch-to-buffer`. Works exactly like the original but with the following enhancements for ECB:

"compilation-buffers" in the sense of `ecb-compilation-buffer-p` will be displayed always in the compile-window of ECB (if `ecb-compile-window-height` is not nil) - if the compile-window is temporally hidden then it will be displayed first. If you do not want this you have to modify the options `ecb-compilation-buffer-names`, `ecb-compilation-major-modes` or `ecb-compilation-predicates`.

If called for non "compilation-buffers" (s.a.) from outside the edit-area of ECB it behaves as if called from an edit-window if `switch-to-buffer` is contained in the option `ecb-layout-always-operate-in-edit-window`. Otherwise an error is reported.

`switch-to-buffer-other-window` *BUFFER* **&optional** *FRAME* [Command]
Around-advice `ecb`: The ECB-version of `switch-to-buffer-other-window`. Works exactly like the original but with some adaptions for ECB so this function works in a "natural" way:

If called in any special ecb-window of the current ECB-layout then it goes always to an edit-window (which one depends on the setting in `ecb-mouse-click-destination`) and then goes on as if called from this edit-window.

If a compile-window is used (i.e. `ecb-compile-window-height` is not nil) then "compilation-buffers" in the sense of `ecb-compilation-buffer-p` are always displayed in the compile-window. If the compile-window is temporally hidden then it will be displayed first. If no compile-window is used it behaves like the original.

If called from within the compile-window then "compilation-buffers" will be displayed still there and all other buffers are displayed in one of the edit-windows - if the destination-buffer is already displayed in one of the edit-windows then this one is used otherwise it behaves like the original.

If called within an edit-window it behaves like the original function except for compilation-buffers (if a compile-window is used, see above).

`other-window-for-scrolling` [Function]
Around-advice `ecb`: This function determines the window which is scrolled if any of the "other-window-scrolling-functions" is called (e.g. `scroll-other-window`):

If the option `ecb-scroll-other-window-scrolls-compile-window` is not nil (maybe set by `ecb-toggle-scroll-other-window-scrolls-compile`) and a compile-window is visible then always the current buffer in the compile-window is scrolled!

Otherwise it depends completely on the setting in `ecb-other-window-behavior`.

`balance-windows` [Command]
Around-advice `ecb`: When called in the `ecb-frame` then only the edit-windows are balanced.

6.4 Temp- and compile-buffers display in ECB

If you call any help in Emacs, e.g. by calling `describe-function`, or if you do a completion in the minibuffer, then Emacs displays the result-buffer in another window. This behavior you have also in ECB.

6.4.1 Standard Emacs behavior

If the edit-area is already splitted into at least two edit-windows then the temp-buffer is displayed in another edit-window otherwise the edit-area will be splitted first into two edit-windows, one above the other. The variables `temp-buffer-max-height` and `temp-buffer-resize-mode` (for GNU Emacs) and `temp-buffer-shrink-to-fit` (for XEmacs) work also correctly with ECB.

Same for all compilation output-buffers (e.g. after a `compile` or `grep`) and the variable `compilation-window-height`.

This is default behavior of ECB. But there is also another way to display such buffers: Using a persistent extra window at the bottom of the ECB-frame:

6.4.2 Using a persistent compile window

With the option `ecb-compile-window-height` you can define if the ECB layout should contain per default a compile-window at the bottom (just specify the number of lines which should be used for the compile-window at the bottom of the frame). If "yes" ECB displays all buffers for which the function `ecb-compilation-buffer-p` returns not nil (e.g. all output of compilation-mode (compile, grep etc.) or all temp-buffers like *Help*-buffers) in this special window.

In general: With the options `ecb-compilation-buffer-names`, `ecb-compilation-major-modes` and `ecb-compilation-predicates` you can define which buffers should be displayed in the compile-window of ECB (for example if you call `switch-to-buffer` or `display-buffer` or if you run `compile` or if you display *Help*-buffers). Per default these are all temp-buffers like *Help*-buffers, all compile- and grep buffers, *Occur*-buffers etc. See the default values of these options.

With the command `ecb-toggle-compile-window` (bound to `C-c . \`) you can toggle the visibility of the compile-window (see ⟨undefined⟩ [Interactive ECB commands], page ⟨undefined⟩).

There are some more useful options and commands related to the compile-window of ECB (to see all options for the compile-window see the customization group ⟨undefined⟩ [ecb-compilation], page ⟨undefined⟩):

- With the option `ecb-compile-window-temporally-enlarge` you can allow Emacs to enlarge temporally the ECB-compile-window in some situations. Please read the comment of this option. See also the description of the command `ecb-toggle-compile-window-height`.
- With the option `ecb-enlarged-compilation-window-max-height` you specify how `ecb-toggle-compile-window-height` should enlarge the compile-window.
- With the command `ecb-cycle-through-compilation-buffers` (see ⟨undefined⟩ [Interactive ECB commands], page ⟨undefined⟩) you can cycle through all current open compilation-buffers (in the sense of `ecb-compilation-buffer-p`) very fast.

ECB offers the same compile-window functionality regardless if the ECB-window are hidden or not. The state of the compile-window will be preserved when toggling the ecb-windows or when maximizing one ecb-windows! So you have the advantage of one special window for all help-, grep or compile-output (see above) also when the ecb-windows are hidden - a window which will not be deleted if you call `delete-other-windows` (bound to `C-x 1`) for one of the edit-windows. In general: All features of the compile-window work with hidden ecb-windows exactly as when the ecb-windows are visible.

6.4.3 What to do if there are problems with the compile-window

Normally displaying temp- and compilation-buffers (or more general: displaying buffer for which `ecb-compilation-buffer-p` is not nil) should work reliable. But if there are problems which you can not handle with the options `ecb-compilation-buffer-names`, `ecb-compilation-major-modes` or `ecb-compilation-predicates` then please go on like follows:

1. Set the option `ecb-layout-debug-mode` to not nil.

2. Reproduce the wrong behavior exactly by repeating all the operations which lead to the problem. If possible then restart Emacs before reproducing the problem so you can begin from the beginning!
3. Now send immediately a bug report with `ecb-submit-problem-report`.
4. Set `ecb-layout-debug-mode` back to nil if you do not want further debugging output in the *Messages* buffer"

6.4.4 Handling special-display-buffers

Emacs offers three options for a special-display-handling of certain buffers: `special-display-function`, `special-display-buffer-names` and `special-display-regexps` (see the Emacs manual for a description of these options). ECB offers an option `ecb-ignore-special-display` for specification in which situations ECB should take account for the values of these special-display-options.

Default-behavior of ECB is to ignore these special-display-options when a persistent compile-window is active (i.e. `ecb-compile-window-height` is not nil). But with `ecb-ignore-special-display` you can tell ECB, that either always the special-display-options should be ignored as long as ECB is active or that they should be never igored regardless if a persistent compile-window is set or not. In the latter case using `display-buffer` or `pop-to-buffer` takes always account for the values of these options - like the original behavior of Emacs.

6.5 How the "other window" is determined by ECB

Normally all windows in an Emacs-frame are arranged in a cyclic order and window-selecting-commands like `other-window` or window-scrolling-commands like `scroll-other-window` choose simply the next[4] window after the current window as "other window".

6.5.1 "Other window"-basics in ECB

With a typical window-layout of ECB such a cyclic order of **all** windows in the ECB-frame does not make sense because it would be not very intuitive and against that what the user wants to "say" when calling `other-window` or `scroll-other-window`.

Therefore ECB divides the whole set of windows of the ECB-frame in several subsets:

- The edit-windows of the edit-area
- The special tree-windows for browsing-tasks
- The compile-window at the bottom (if there is one)
- The minibuffer-window of the ECB-frame (if active)

Each of these subsets will be treated as a cyclic ordered subset, i.e. all windows in each of these subsets are ordered as the function `walk-windows` would visit the windows when the windows of a subset would be the only windows of a frame[5].

[4] `other-window` allows to select ARG'th different window, i.e. the window ARG steps away from current window in the cyclic order of the windows

[5] `other-window` uses the same window-ordering as `walk-windows`

6.5.2 Builtin "other window" behaviors of ECB

ECB now offers to specify the behavior of commands like `other-window` or `scroll-other-window` within the ECB-frame. This can be done with the option `ecb-other-window-behavior`. This option offers several builtin behaviors:

- All windows of the ECB-frame are considered

  ECB will cycle through all windows of the ECB-frame or scroll simply the next window in the ECB-frame, means it behaves like the original `other-window` rsp. the original `other-window-for-scrolling`.
- Only the windows of the edit-area are considered

  ECB will only cycle through the edit-windows of ECB or only scroll another edit-window. If the selected window is not an edit-window then all windows are taken into account.
- The edit-windows and the compile-window are considered

  Like above but the compile-window will be added to the subset of the edit-windows.
- Behave as smart and intuitive as possible

  This is the default behavior of ECB. ECB tries to choose the `other-window`-destination or the "other window" to scroll in a smart and intuitive way: If point is in one of the edit-windows and if the edit-area is splitted then always the "next" edit-window is choosen (whereas the next edit-window of the last edit-window is the first edit-window)- if the edit-area is unsplitted then the compile-window is used if there is one. In the context of an `other-window`-call the *ARG* of `other-window` will be taken into account.

  If one of the special ecb-windows is selected then always the "next" ecb-window is choosen (whereas the next ecb-window of the last ecb-window is the first ecb-window). In the context of an `other-window`-call the *ARG* of `other-window` will be taken into account. If there is only one ecb-window then ECB considers also the edit-windows.

  If the compile-window is selected then always the last edit-window which had the point will be used unless `other-window` has been called with a prefix-argument unequal 1.

Regardless of the different behaviors above ECB handles the situation of an active minibuffer during a call to `other-window` or `scroll-other-window` like follows:

If the minibuffer-window is selected then ECB always chooses the window `minibuffer-scroll-window` points to (when this variable is set, otherwise the compile-window or the last selected edit-window is choosen) when the called command is called to choose the 1. next window (always true for scrolling another window or true when `other-window` called without prefix-arg or with prefix-arg equal 1). Otherwise the window ARG steps away is choosen (in case of `other-window`).

If there is an active minibuffer but the minibuffer-window is not selected then `other-window` and `scroll-other-window` behave like the original version.

6.5.3 User-defined "other window" behavior

In addition to the builtin "other window" behaviors ECB offers a user to completely define for himself how ECB should choose another window for scrolling it or selecting it. This can be done with the option `ecb-other-window-behavior` too because this option can also have a function-symbol as value:

Such a function gets seven arguments:

1. A canonical list of all currently visible windows of the `ecb-frame`
2. A canonical list of all currently visible edit-windows
3. A canonical list of all currently visible ecb-windows
4. The window-object of the compile-window if there is any.
5. The minibuffer-window of the ECB-frame if there is an active minibuffer.
6. The result of the function `ecb-where-is-point` - see the documentation of this function for details.
7. An integer which indicates how many steps away from the current selected window the "other-window" is. Is nil when this function is called in another context than for `other-window`.

The function has to return a window-object which is then used as "other window" for the command `other-window` or for scrolling another window (e.g. with `scroll-other-window`). Such a function has to handle properly all situation for itself.

Here is an example for such a function:

```
(defun ecb-get-other-window-smart (win-list
                                   edit-win-list
                                   ecb-win-list
                                   comp-win
                                   minibuf-win
                                   point-loc
                                   nth-window)
  "Implements the smart-setting of `ecb-other-window-behavior'."
  (if minibuf-win
      ;; if we have an active mini-buffer we delegate this to
      ;; `ecb-get-other-window-minibuf-active'
      (ecb-get-other-window-minibuf-active win-list
                                           edit-win-list
                                           ecb-win-list
                                           comp-win
                                           minibuf-win
                                           point-loc
                                           nth-window)
    ;; here we have no active minibuffer!
    (let ((nth-win (or nth-window 1)))
      (cond ((equal point-loc 'ecb)
             (ecb-next-listelem ecb-win-list (selected-window) nth-win))
            ((equal point-loc 'compile)
             (if (= nth-win 1)
                 (or (and ecb-last-edit-window-with-point
                          (window-live-p ecb-last-edit-window-with-point)
                          ecb-last-edit-window-with-point)
                     (car edit-win-list))
               (ecb-next-listelem (append edit-win-list
                                          (list (selected-window)))
                                  (selected-window)
                                  nth-win)))
            (t ;; must be an edit-window
             (ecb-next-listelem (append edit-win-list
                                        (if (and comp-win
                                                 (= (length edit-win-list)
                                                    1))
                                            (list comp-win)))
                                (selected-window)
                                nth-win))))))
```

This example implements the builtin smart behavior described above.

6.6 Using and customizing the ECB-Methods buffer

ECB is mostly designed to display parsing information for files supported by semantic. But beginning with version 1.94 it also supports other parsing engines like imenu and etags, so

also files not supported by semantic but by imenu/etags can be displayed in the Method-buffer of ECB. Therefore we have to introduce some terminology:

- *semantic-sources*: These are file-types for which a semantic grammar is available, so the files are parse-able by semantic. These sources are supported best by ECB and most of the following options and descriptions are related to these file-types. Examples are programming-sources like C++, C, Java, Emacs-Lisp and Texinfo-file and some more.
- *non-semantic-sources*: For these files there is no semantic-grammar available so they can not be parsed by semantic. Examples are Perl-, LaTeX- and TeX-files. But for many of these files imenu and/or etags parsers exist. ECB supports now parsing and displaying these file-types too and it uses for this some speedbar-logic.

This chapter describes how to use and customize the Methods-buffer of ECB.

6.6.1 Possible actions after visiting a tag

You visit a tag by clicking with either the primary oder secondary mouse-button (or by hitting *RET* or *C-RET* if using the keyboard) onto a node in the Methods-tree-buffer of ECB. This simply selects the "right" edit-window (depends if clicked with the primary or secondary button, in how many windows the edit-area is splitted and the value of `ecb-mouse-click-destination`) and puts the point onto the first line of the clicked tag.

But you can define if after this "basic" tag-visit-action more additional actions should be performed by ECB. You can either use some of the predefined actions (e.g. highlighting the header-line of the tag) or define own actions. You can set different actions for different major-modes. All this is done via the option `ecb-tag-visit-post-actions`.

The following actions are currently predefined:

- `ecb-tag-visit-highlight-tag-header`
- `ecb-tag-visit-smart-tag-start`
- `ecb-tag-visit-recenter`
- `ecb-tag-visit-recenter-top`
- `ecb-tag-visit-goto-doc-start`
- `ecb-tag-visit-narrow-tag`

See the documentation of these function for details what they do.

Per default ECB performs the actions `ecb-tag-visit-smart-tag-start` and `ecb-tag-visit-highlight-tag-header` for all major-modes.

6.6.2 Explicit and automatic expanding of the ECB-methods-buffer

6.6.2.1 Explicit expanding all nodes to a certain expansion level

With the command `ecb-expand-methods-nodes` (bound to *C-c . x*) you can get a fast overlook of the contents of the source-buffer, because this command allows precisely expanding all tags with a certain indentation-level. So you can either expand no tags (or with other words collapse all tags) or expand all tags so see the contents of a buffer at one glance. Or you can expand exactly that tags of a certain indentation level.

Which node-types are expanded (rsp. collapsed) by this command depends for semantic-sources on the options `ecb-methods-nodes-expand-spec` and `ecb-methods-nodes-collapse-spec`! For non-semantic-sources always all node-types are expanded/collapsed, i.e. the two options above takes no effect for these files.

6.6.2.2 Explicit expanding of the current node to a certain level

With the popup-menu of the methods-buffer an even more precise expansion is possible because it allows not only expanding all tags (see above) but offers in addition expanding only the current-node (for which the menu was activated) to an exact level of expansion:

All menu-entries are label with an expansion-"level" whereas level specifies precisely which level of nodes should be expanded. level means the indentation-level of the NODE itself and its (recursive) subnodes relative to the NODE itself.

So a level value X means that all (sub)nodes with an indentation-level <= X relative to NODE are expanded and all other are collapsed.

Examples:

- Expand this node to level 0: Expand only the NODE itself because it is the only node which has indentation 0 to itself. All deeper indented nodes will be collapsed. This is also the important difference between using this menu compared to clicking onto the expand-symbol of the node: The latter one expands the NODE to that expansion-state it has before the last collapsing (so when deeper nodes has been expanded they will be expanded now to). The former one expands exactly(!) to level 0, means expand only the node itself and collapse all(!) its subnodes recursively(!).
- Expand this node to level 1: Expand the NODE itself and all of its direct subnodes - because only the direct subnodes of NODE have indentation-level 1 relativ to NODE. All deeper nodes will be collapsed.
- Collapse this node completely: Collapses the current node recursively, means collapse not only the node itself but also its subnodes, the subnodes of the subnodes and so on! This is very differnt from clicking onto the collapse symbol because this action only collapses the node itself but preserves the expansion-state of all its subnodes!

Expanding the current node with the popup-menu ignores the settings in the options `ecb-methods-nodes-expand-spec` and `ecb-methods-nodes-collapse-spec`!

6.6.2.3 Automatic expansion ot tags after buffer-parsing

With the option `ecb-show-tags` you tell ECB how to display tags of a certain tag-class (see ⟨undefined⟩ [Customizing the display], page ⟨undefined⟩). Among other things you can tell ECB that a certain tag-class should be combined under an expanded or collapsed bucket-node. But such a setting defines the expansion-state of such a bucket-node only direct afterwards the buffer has been (re)parsed, which can occur after opening a file, after autom. reparsing the buffer via semantic or after manually rebuilding the methods-buffer of ECB.

The tag-class `type` (classes, interfaces, enumerations etc.) is divided into several subtypes by semantic. The subtypes are strings which names exactly if the tag with tag-class `type` is a class, an interface, an enumeration, a typedef etc. With the option `ecb-type-tag-expansion` you can tell ECB if these type-tags should be autom. expanded after (reparsing)

a buffer (see above) or if they should be displayed collapsed - so all its members (methods, variables etc.) are hidden.

6.6.2.4 Automatic expanding the ECB-methods-buffer for current tag

If the option `ecb-highlight-tag-with-point` is switched on, then always that node in the method-buffer is highlighted which belongs to the current semantic-tag under point in the current active edit-window. But if this node is invisible (probably because its parent node is collapsed) then no node is highlighted if the auto. expanding feature is switched off.

You can either switch on this feature with the option `ecb-auto-expand-tag-tree` or even easier with the command `ecb-toggle-auto-expand-tag-tree`.

There is another option `ecb-expand-methods-switch-off-auto-expand` which makes both explicit and auto. expanding best working together. See the documentation of this option to get the details.

The autom. expanding feature is only available for semantic-sources!

Previous versions of ECB have always fully expanded the whole tree in the Methods-buffer if the current tag in the source-buffer was not visible in the current tree - e.g. because the variables-bucket was collapsed or the containing type of a tag (e.g. the class of a method) was collapsed. So in most cases much more was expanded as needed to make the current tag visible.

The mechanism of ECB 2.22 and higher only expands the needed parts of the tree-buffer to make the related node visible: First we try to highlight the current tag with current expansion-state of the Methods-buffer. If the node is not visible so the tag can not be highlighted then we go upstairs the ladder of type-tags the current tag belongs to (e.g. we expand successive the nodes of the whole class-hierachy of the current method-tag until the related node becomes visible). If there is no containing type for the current tag then the node of the tag is probably contained in a toplevel-bucket which is currently collapsed; in this case we expand only this bucket-node and try highlighting again. Only if this has still no success then we expand the full tree-buffer and try to highlight the current tag.

There is another option `ecb-auto-expand-tag-tree-collapse-other`: If this option is set then auto. expanding the tag-tree collapses all not related nodes. This means that all nodes which have no relevance for the currently highlighted node will be collapsed, because they are not necessary to make the highlighted node visible. This feature is switched off by default because if always collapses the complete Methods-tree before the following highlighting of the node for the current tag expands the needed parts of the tree-buffer.

6.6.3 Customizing the display of the Methods-buffer

The ECB-Methods buffer is probably the most important browsing window offered by ECB. It displays all parsing informations of the current source-buffer (the buffer displayed in the current active edit-window).

Normally ECB gets all informations displayed in this Methods-buffer from the semantic-library - at least for semantic-sources. This library parses auto. the current source-buffer in the edit-window of ECB and returns all information in form of *tags* to ECB which displays them in a browse-able form in its Method-buffer. See ⟨undefined⟩ [ECB Methods-buffer], page ⟨undefined⟩ for information how to use the Methods-buffer.

There are several options to customize which tags ECB should display in general, if the tags should be collapsed or expanded, how to fontify them (i.e. syntax-highlighting) and something more.

`ecb-show-tags`
: With the option `ecb-show-tags` you specify how ECB should display the tags returned by the semantic parser. Semantic divides its tags in several so called *tag classes*. A tag-class is always a symbol and can be for example `type` (tags which represent a class[6], a interface, an enumeration etc.), `function` (tags which represent function or methods), `variable` (variables and attributes), `include` (import-statements) etc. There is no predefined superset of allowed tag-class-symbols because each language-parser can define its own tag-classes. But to get an overview of the most common tag-classes see the default value of the option `ecb-show-tags`.

  With the option `ecb-show-tags` you can now specify how ECB should display tags of a certain tag-class in a certain major-mode. You can tell ECB that all tags of a tag-class `X` should be displayed in an expanded bucket and all tags of a tag-class `Y` should be displayed in a collapsed bucket and all tags of a tag-class `Z` should be displayed flattened (means not contained in a expandable/collapsable bucket-node). These settings can be made separately for each major-mode but you can also define a default-display which takes effect when for a major-mode no special setting can be found in `ecb-show-tags`.

  For every tag-class you can tell ECB how the tags should be sorted.

`ecb-font-lock-tags`
`ecb-type-tag-display`
: How to fontify the tags in the Method-buffer

`ecb-tag-display-function`
: ECB and semantic offer several predefined functions for displaying the tags. Here you can customize, what informations tags should contain (only the method-name or the whole signature or something else) and what notation should be used, e.g. UML or not.

These are the most important options for this topic but it is recommended to have a look into the customize-group `ecb-methods` (see ⟨undefined⟩ [ecb-methods], page ⟨undefined⟩) and check all the options offered there!

All these options are only relevant for semantic-sources and take no effect for non-semantic-sources!

6.6.4 Rebuilding the Methods-buffer

In almost all cases there is **NO** need to manually rebuild the method-buffer, because it is always done automatically if necessary; the mechanism depends on the sources:

[6] Do not confuse the term "class" in the context of a tag, which means the class of the tag and which is a semantic-term and a "class" in the context of an object oriented language like Java or C++! Normally the surrounding context sould be sufficient to understand which type of "class" is meant whenever the term "class" is used in this manual.

- semantic-sources: The command `global-semantic-auto-parse-mode` switches on autom. reparsing of semantic-sources.
- non-semantic-sources (imenu supported): You can switch on autom. rescanning/reparsing with the option `imenu-auto-rescan`. But nevertheless you have to manually rebuild the Method-buffer (with the autom. updated imenu-tags) via the command `ecb-rebuild-methods-buffer` (bound to *C-c . r*).
- non-semantic-sources (etags supported): For these sources there is no built-in auto-rescan mechanism, because etags is an external tool it can only operate on the saved file-contents. So rescanning the buffer contents would need to save the buffer before. Therefore there is no built-in auto-rescan mechanism because this would always result in saving the buffer and running an external tool. But of course you can program such a an etags-auto-rescan mechanism for yourself!

Besides for etags-supported non-semantic-sources there exist a few rare scenarios also for the other sources where a complete manual rebuild can be necessary. Here is one example:

Depending on the semantic-version: If an Elisp-file is parsed which contains a defun X in the middle where the closing) is missing, then semantic parses only until this defun X is reached and you will get an incomplete ECB-method buffer. In such a case you must complete the defun X and then completely reparse the Elisp-file and rebuild the ECB method buffer!

A complete manually rebuild is done by `ecb-rebuild-methods-buffer`. For etags-parsed non-semantic-sources this causes an automatic saving of the source-buffer because otherwise etags would not operate with the latest contents!

6.7 Applying filters to the special ECB-tree-buffers

To get a fast and good overlook of the contents of a tree-buffer ECB offers a filter-mechanism for the Directories-, Sources-, the History- and the Methods-buffer.

6.7.1 Applying filters to the Directories-buffer

With the option `ecb-excluded-directories-regexps` certain directories can be excluded from being displayed in the directories-browser of ECB. This can be done by defining regular expressions. If the name of a directory matches at least one of the regexps of this option the directory is not displayed in the directories-tree-buffer.

The option `ecb-sources-exclude-cvsignore` allows to exclude source-files from the sources-tree-buffer if their name is listed in a so called '`.cvsignore`'-file. This option is a list of regular expressions and if a directory-name matches at least one of these regexps then all files listed in the '`.cvsignore`'-file of this directory are not displayed in the sources-tree-buffer.

6.7.2 Applying filters to the Sources-buffer

6.7.2.1 Interactive Sources-filters

The command `ecb-sources-filter` allows to filter these tree-buffer either by a regular expression or by an extension (e.g. java, cc, el for java-, c++- rsp elisp-sources). This functionality is also available via the popup-menu of the Sources-tree-buffer.

The currently applied filter is indicated in the modeline of the related tree-buffer. Applying a new filter replaces an eventually already applied filter.

6.7.2.2 Default Sources-filters

The option `ecb-source-file-regexps` allow to specify which source-files should be displayed and which not. This is done on a directory-basis, ie. for each directory-regexp the files to display can be specified. See the documentation of this option for all details.

In addition to this option you should also know the option `ecb-sources-exclude-cvsignore` (see ⟨undefined⟩ [Filtering Directories], page ⟨undefined⟩).

6.7.3 Applying filters to the History-buffer

6.7.3.1 Interactive History-filters

The command `ecb-history-filter` allows to filter these tree-buffer either by a regular expression or by an extension (e.g. java, cc, el for java-, c++- rsp elisp-sources). This functionality is also available via the popup-menu of the History-tree-buffer.

The currently applied filter is indicated in the modeline of the related tree-buffer. Applying a new filter replaces an eventually already applied filter.

6.7.3.2 Default History-filters

The option `ecb-history-exclude-file-regexps` allows to exclude source-files from being historized (ie. displayed in the History-buffer). Despite the fact that the History-buffer already excludes all non-file-buffers there can be still buffers which name you do not wat to be displayed in the history. These are file-buffer like '`TAGS`' or '`semantic.cache`' which store meta-informations used by Emacs and its tools (etags, semantic etc.). These files can be excluded via `ecb-history-exclude-file-regexps`.

6.7.4 Applying filters to the Methods-buffer

The commands `ecb-methods-filter`, `ecb-methods-filter-regexp`, `ecb-methods-filter-protection`, `ecb-methods-filter-tagclass`, `ecb-methods-filter-function`, `ecb-methods-filter-delete-last`, `ecb-methods-filter-nofilter` allows to filter the tags/nodes of the Methods-buffer by several criterias. As for the Sources- and the History-buffer the same functionality is also available via the popup-menu of the Methods-buffer.

6.7.4.1 Possible filter-criterias

- Filter by protection: Just insert the protection you want the Methods-buffer being filtered: private, protected or public! For sources not supported by semantic the protection filter will not be offered because these informations are not available for such sources.
- Filter by regexp: Insert the filter as regular expression.
- Filter by tag-class: You can filter by tag-classes. The popup-menu contains mode-dependend tag-filter entries and the command `ecb-methods-filter` offers only the tag-classes of the current mode. This means for sources not supported by semantic the tag-class filter will not be offered. And for semantic-supported sources exactly

these tag-classes are offered the semantic-parser for the current major-mode offers. For example texi-sources can only be filtered by the tag-classes “Definitions” and “Sections” and java-sources can be filtered by “Methods”, “Variables”, “Classes” etc. In general the semantic-variable `semantic-symbol->name-assoc-list` is used to get the right tag-classes.

- Filter by a filter-function: Such a function gets two arguments: a tag and the source-buffer of this tag. If the tag should be displayed (i.e. not being filtered out) then the function has to return not nil otherwise nil.
- No special filter: This means to display all tags specified with the option `ecb-show-tokens`. If currently some of the above filters are applied they will be all removed.
- Delete the last added: This removes only the topmost filter-layer, means that filter added last.

Be aware that the tag-list specified by the option `ecb-show-tags` is the basis of all filters, i.e. tags which are excluded by that option will never be shown regardless of the filter type here!

All tags which match the applied filter(s) will be displayed in the Methods-buffer. Such a filter is only applied to the current source-buffer, i.e. each source-buffer can have its own tag-filters.

These tag-filters can also applied to sources which are not supported by the semantic-parser but “only” by imenu or etags. But because for these sources not all information are avaiable the protection- and tag-class filter are not offered with such “non-semantic”-sources. See ⟨undefined⟩ [Non-semantic sources], page ⟨undefined⟩ for further details about working with source-files not supported by the semantic-parser.

6.7.4.2 Inverse Filters

But if `ecb-methods-filter` is called with a prefix-argument then an inverse filter is applied to the Methods-buffer, i.e. all tags which do **NOT** match the choosen filter will be displayed in the Methods-buffer!

6.7.4.3 Layered filters

Per default the choosen filter will be applied on top of already existing filters. This means that filters applied before are combined with the new filter. This behavior can changed via the option `ecb-methods-filter-replace-existing`.

6.7.4.4 Display of currently applied filters

The current active filter will be displayed in the modeline of the Methods-buffer [regexp, prot (= protection), tag-class, function (= filter-function)]. If an inverse filter has been applied then this is signalized by a preceding caret `^`. If currently more than 1 filter is applied then always the top-most filter is displayed in the modeline but the fact of more than 1 filter is visualized by the number of the filters - included in parens. You can see all currently applied filters by moving the mouse over the filter-string in modeline of the Methods-buffer: They will displayed as help-echo.

6.7.4.5 Default filters for certain files.

The new option `ecb-default-tag-filter` allow to define default tag-filters for certain files which are applied automatically after loading such a file into a buffer. The possible filters are the same as offered by the command `ecb-methods-filter` and they are applied in the same manner - the only difference is they are applied automatically. The files can be specified on a combined major-mode- and filename-regexp-basis.

Usage-example: This can be used to display in outline-mode files (e.g. '`NEWS`') only the level-1-headings by defining a filter regexp "`^\* .*`".

6.8 Changing, customizing, redrawing and creating layouts

The term *ECB-layout* means in which windows the ECB-frame is divided. This chapter describes all aspects concerning this layout, especially changing, customizing, redrawing and also creating new layouts.

6.8.1 Changing and customizing the ECB-layout

ECB offers several predefined layouts with different sets and also different locations of ECB-windows. See below the "ascii-screenshot" of all currently built-in layouts[7].

In addition to these predefined layouts you can either interactively create new layouts "by example" (see ⟨undefined⟩ [Creating a new ECB-layout], page ⟨undefined⟩) or program new layouts with the macro `ecb-layout-define` (see ⟨undefined⟩ [The layout-engine], page ⟨undefined⟩). The former method is the recommended one!

There are two ways to interactively change the layout:

- Changing permanently: Customize the option `ecb-layout-name` and store it for future Emacs sessions.
- Switching between several layouts at runtime: If you want to switch fast between a certain sequence of layouts see the option `ecb-toggle-layout-sequence` and the command `ecb-toggle-layout` (see ⟨undefined⟩ [Simulating speedbar], page ⟨undefined⟩). For just selecting another layout during current Emacs-session use the command `ecb-change-layout`.

With the option `ecb-show-sources-in-directories-buffer` you can define if sources are displayed in the directory-window of a layout (see ⟨undefined⟩ [ECB Directories-buffer], page ⟨undefined⟩).

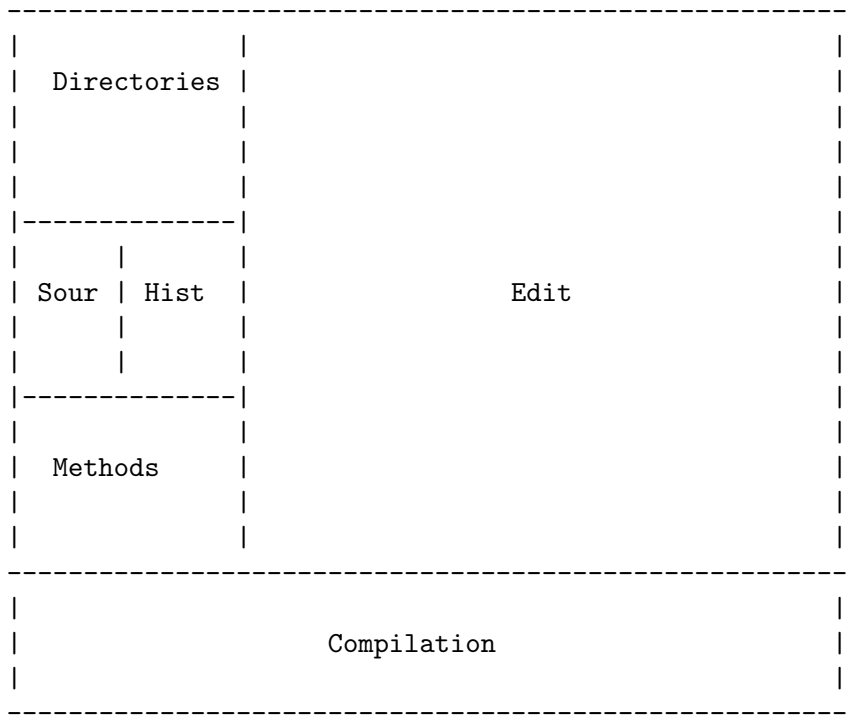
In addition to the general layout you can specify if the layout should contain a persistent compilation-window at the bottom of the frame, see `ecb-compile-window-height` (see ⟨undefined⟩ [Temp- and compile-buffers], page ⟨undefined⟩).

Maybe you want also change the look&feel of the tree-buffers. Then you can change the general style of the tree-buffers with the option `ecb-tree-buffer-style` and the location of the collapse- and expand-symbols and the indentation of sub-nodes in a tree. See `ecb-tree-indent` and `ecb-tree-expand-symbol-before`. More details about the different tree-buffer-styles are described in ⟨undefined⟩ [Tree-buffer styles], page ⟨undefined⟩.

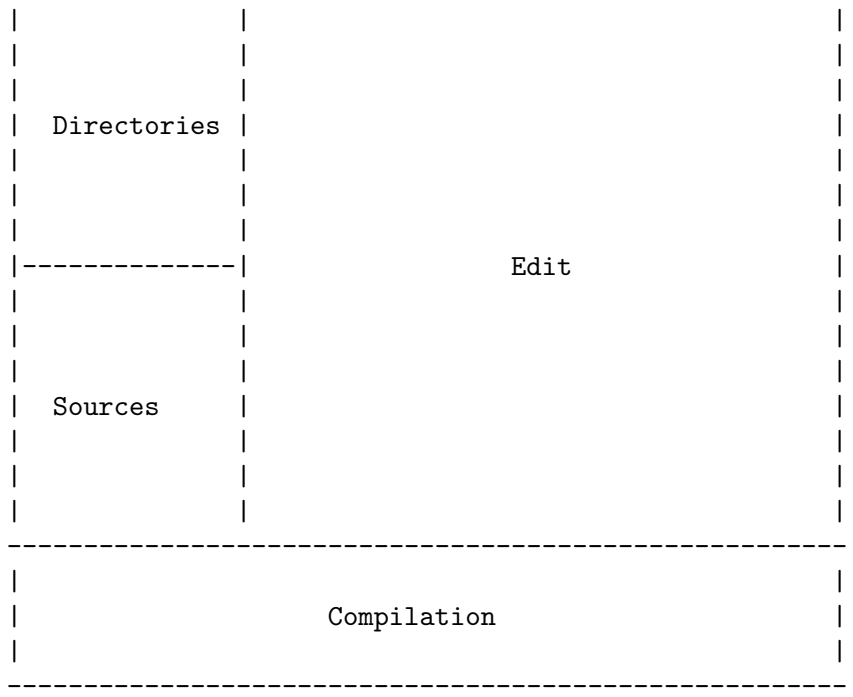
Here are all currently available layouts (for creating own new layouts see ⟨undefined⟩ [Creating a new ECB-layout], page ⟨undefined⟩); just customize the option `ecb-layout-name` to the related name:

[7] The command `ecb-show-layout-help'` shows the outline-picture for all built-in layouts.

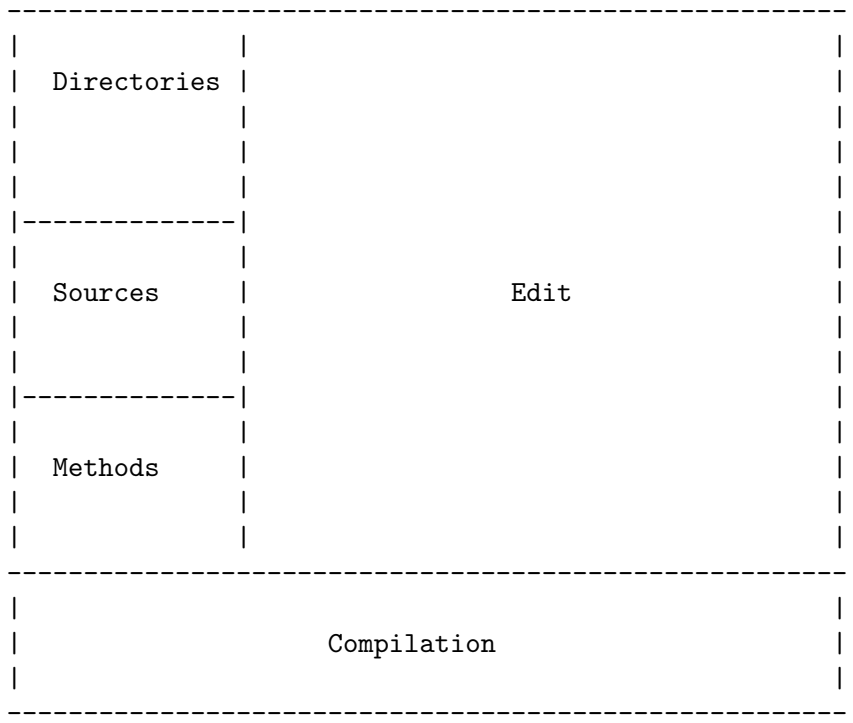
Layout “left1”



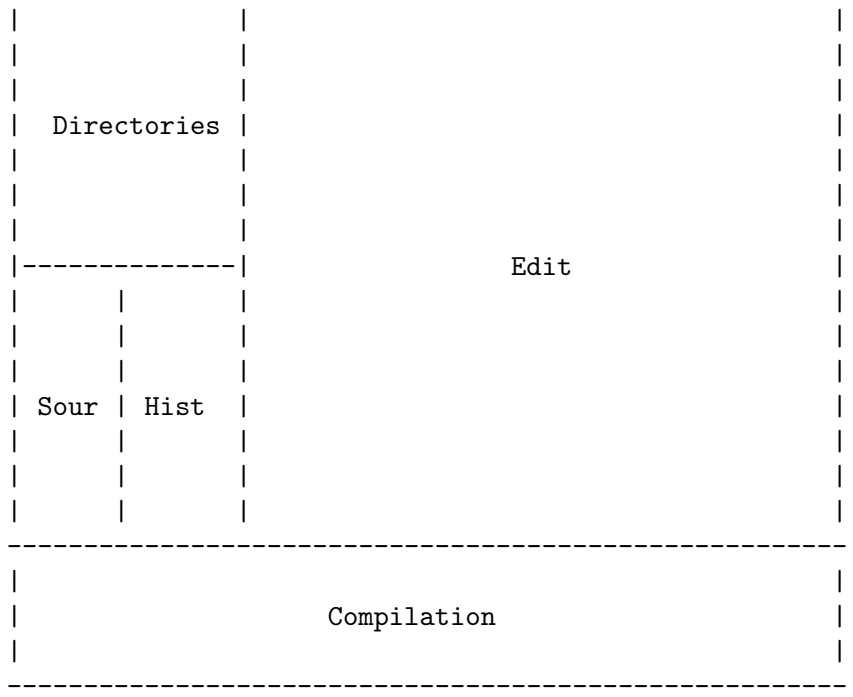
Layout “left2”



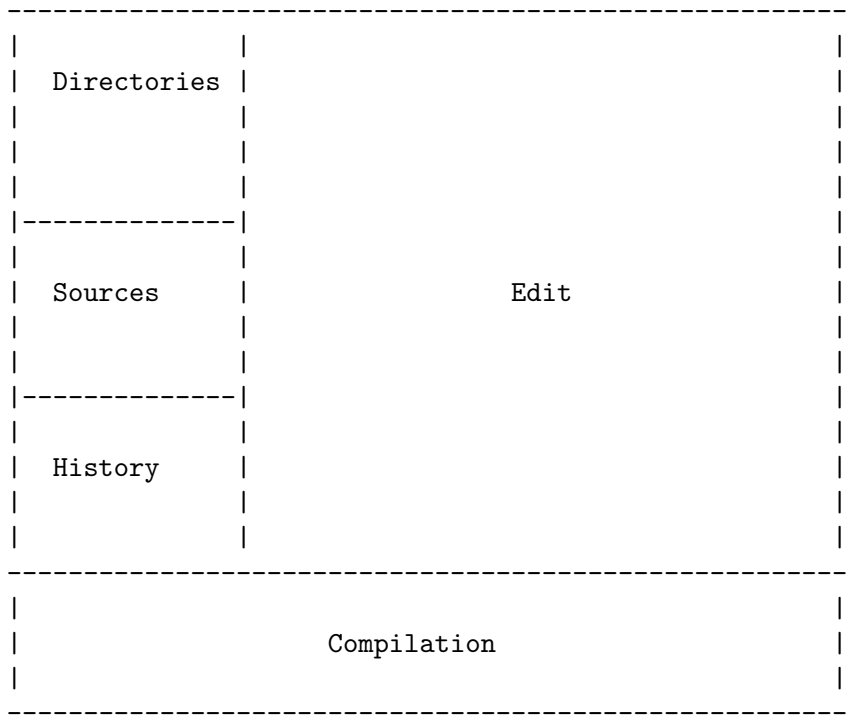
Layout “left3”



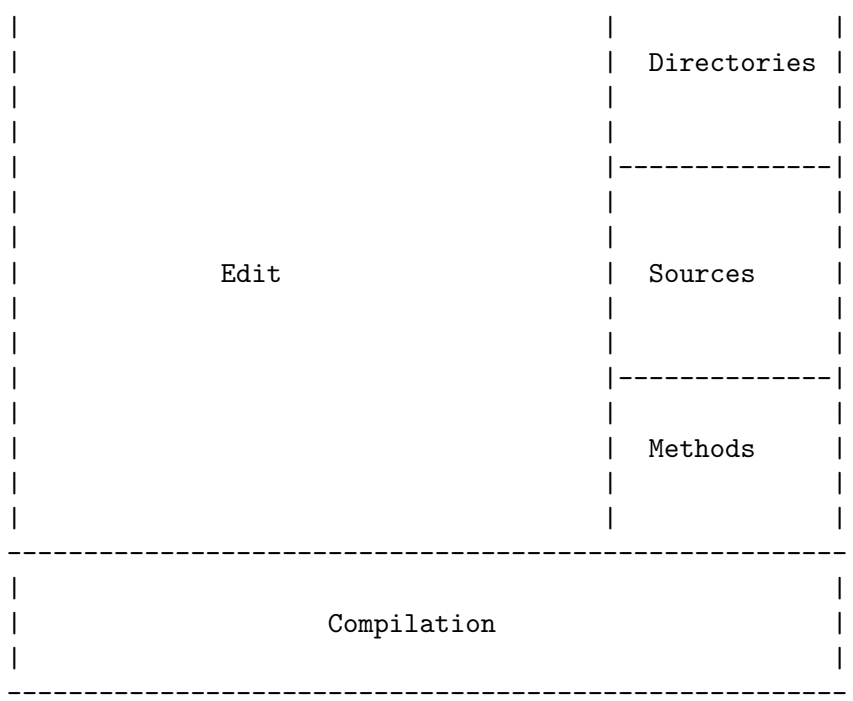
Layout “left4”



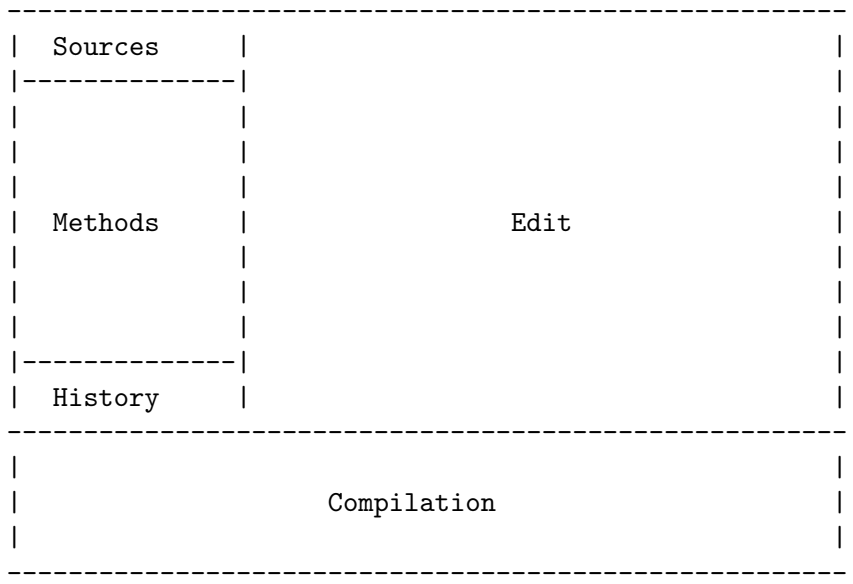
Layout “left5”



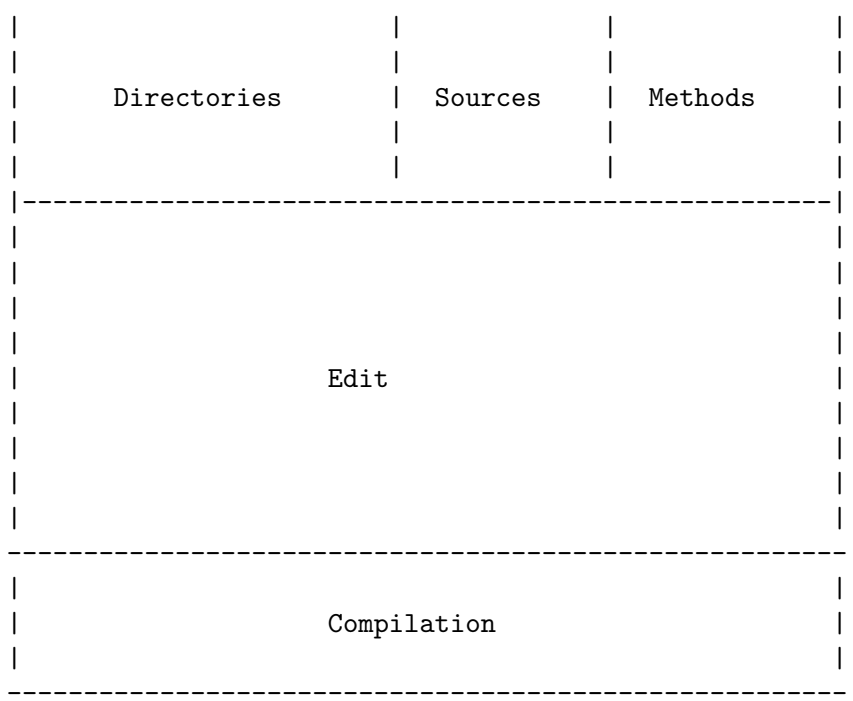
Layout “right1”



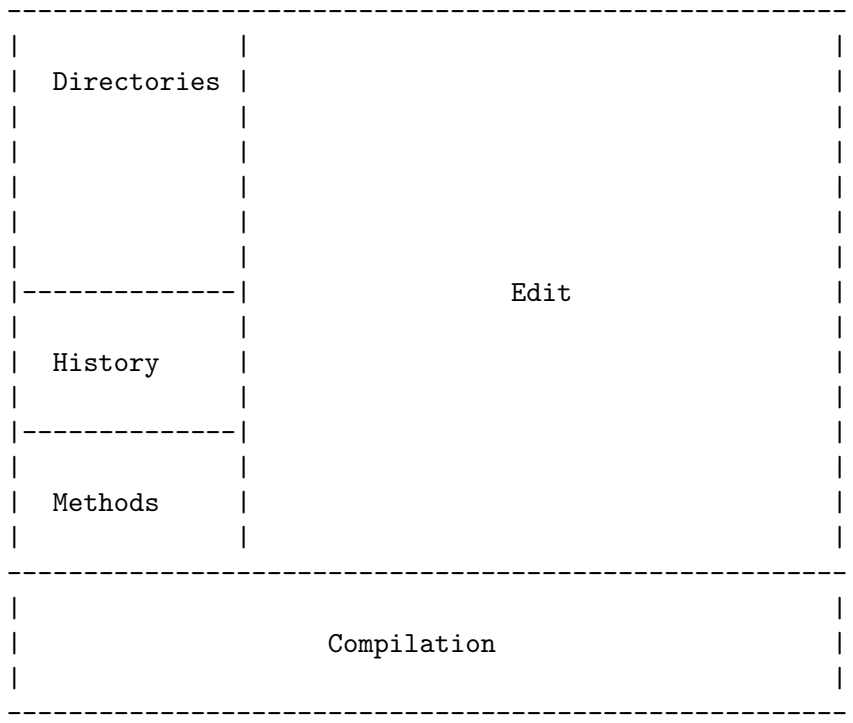
Layout “left6”



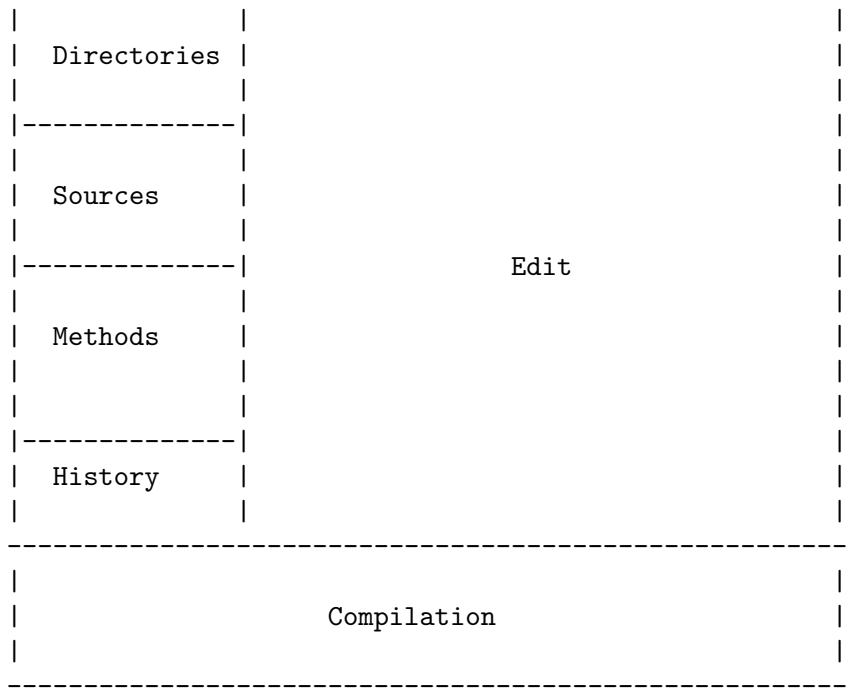
Layout “top1”



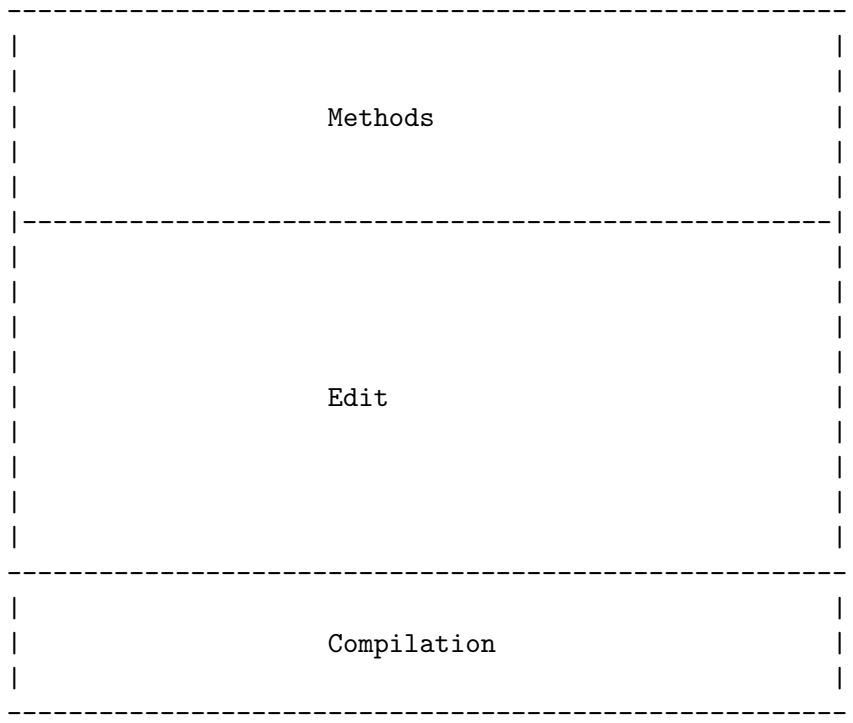
Layout “left7”



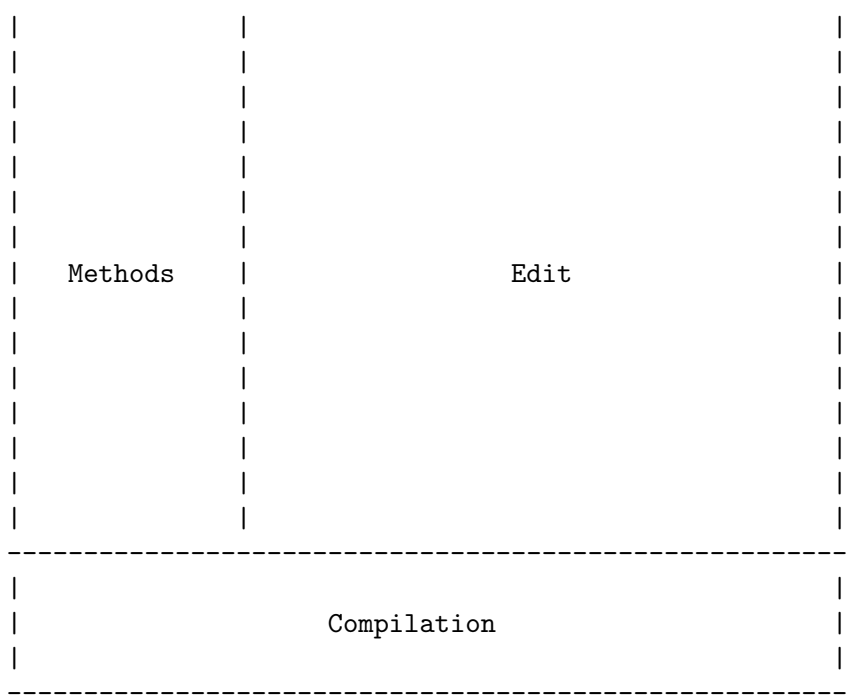
Layout “left8”



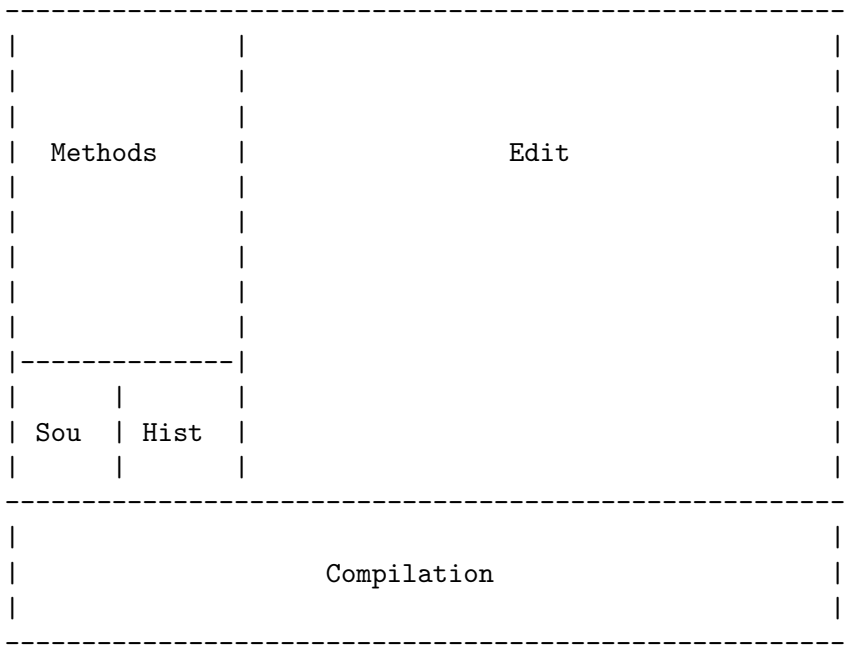
Layout “top2”



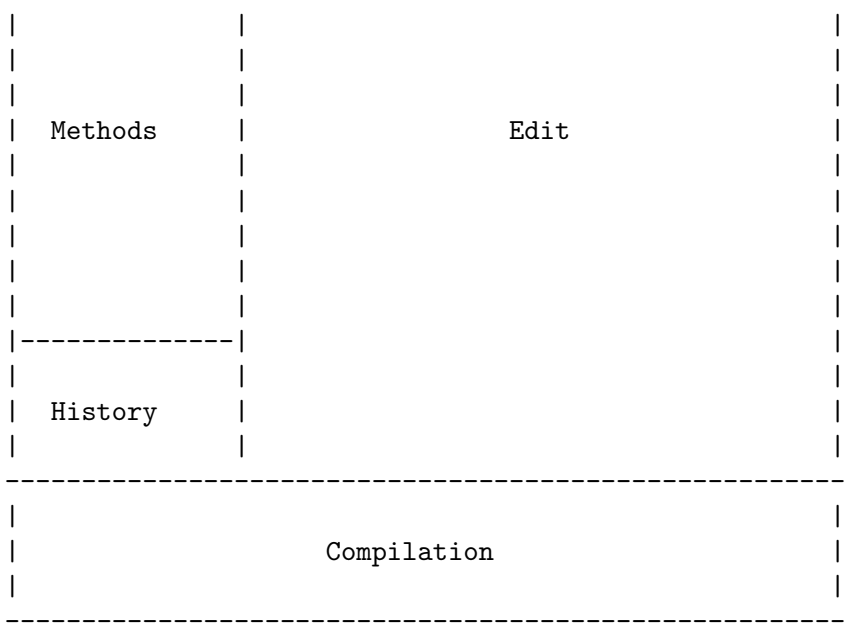
Layout “left9”



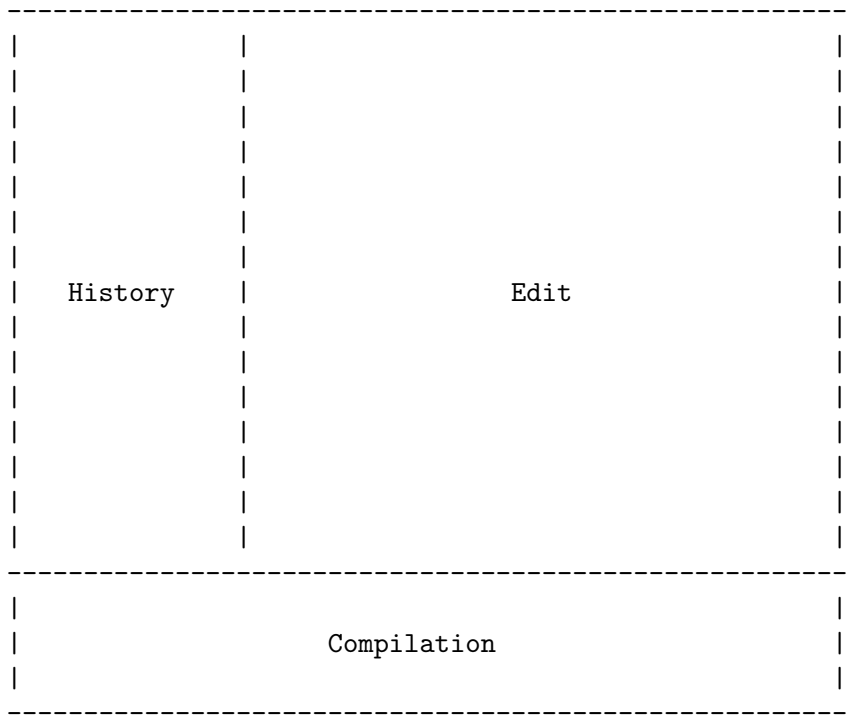
Layout “left10”



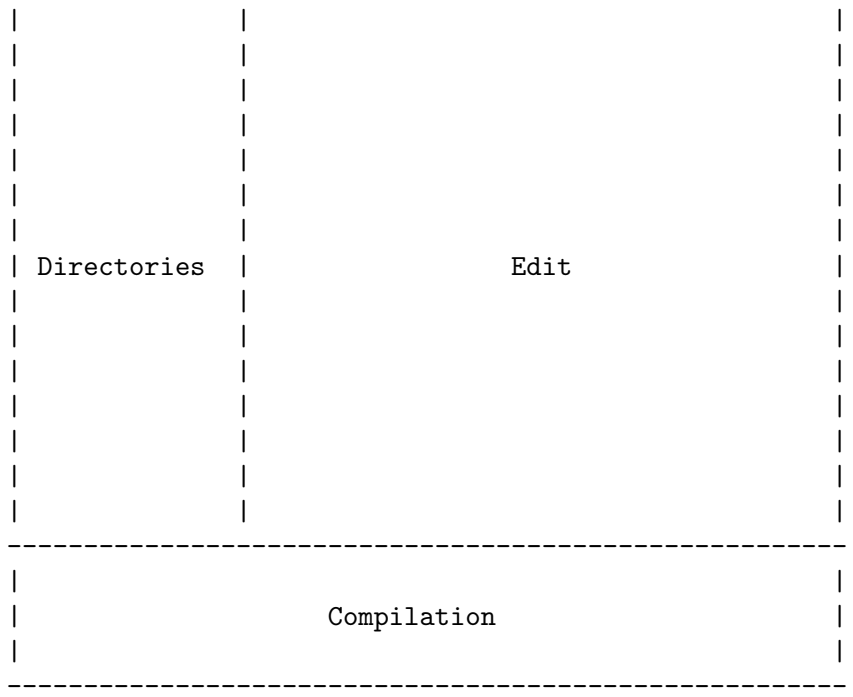
Layout “left11”



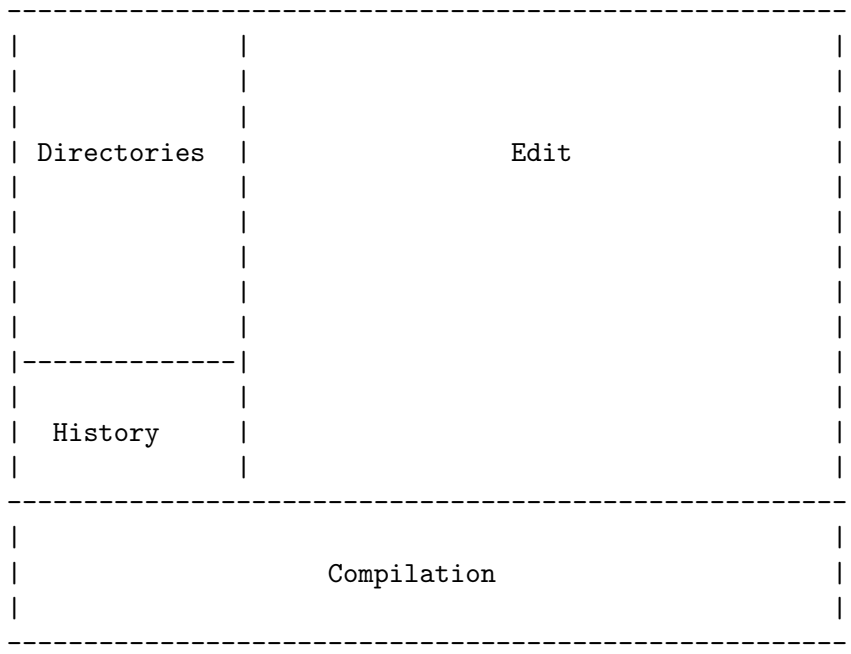
Layout “left12”



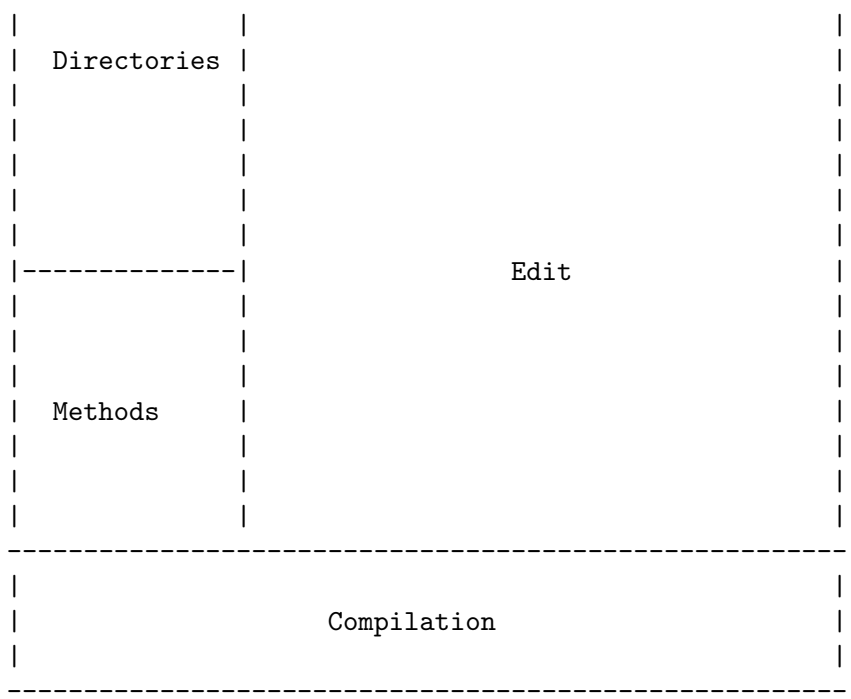
Layout “left13”



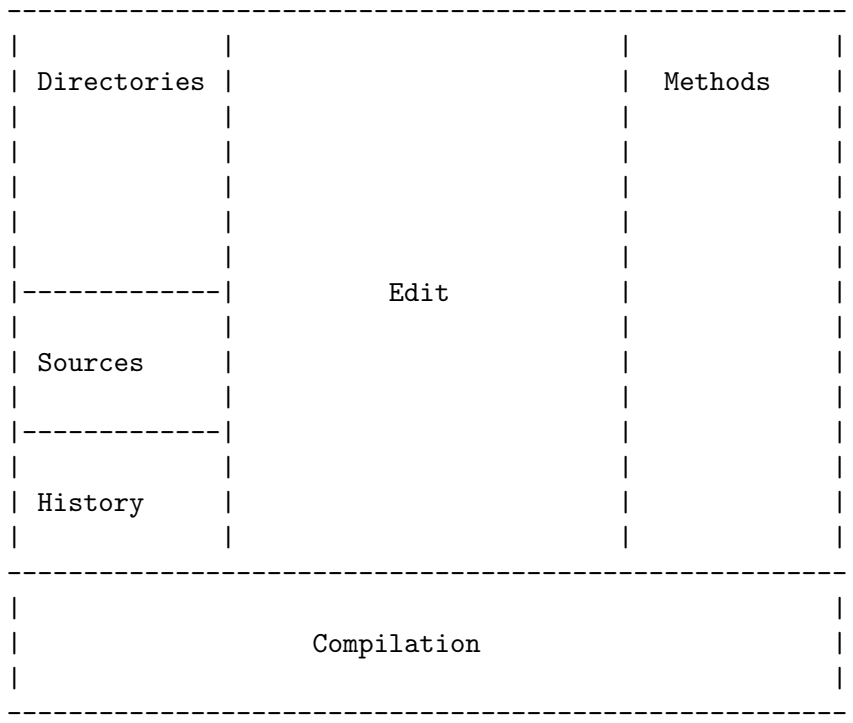
Layout “left14”



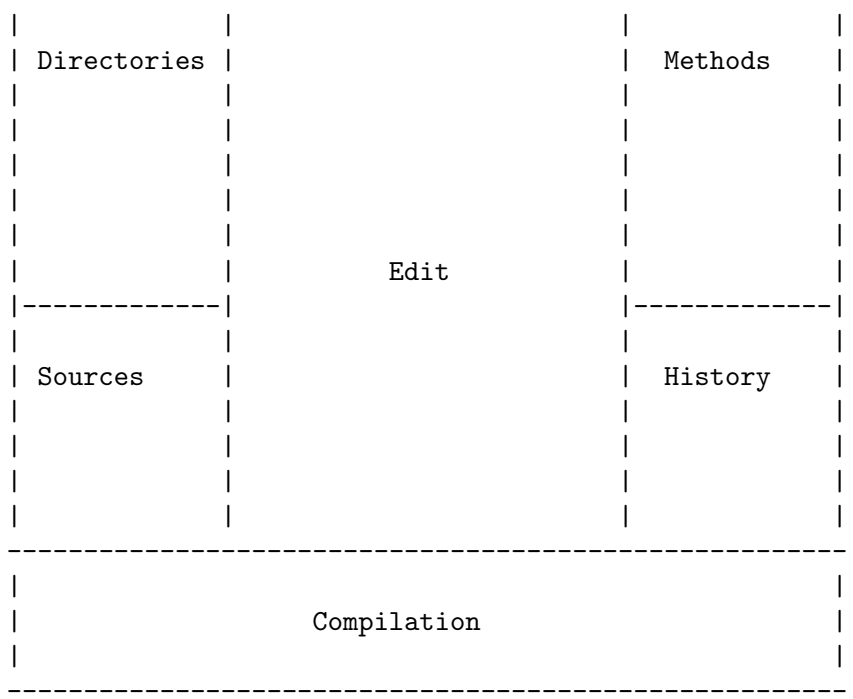
Layout “left15”



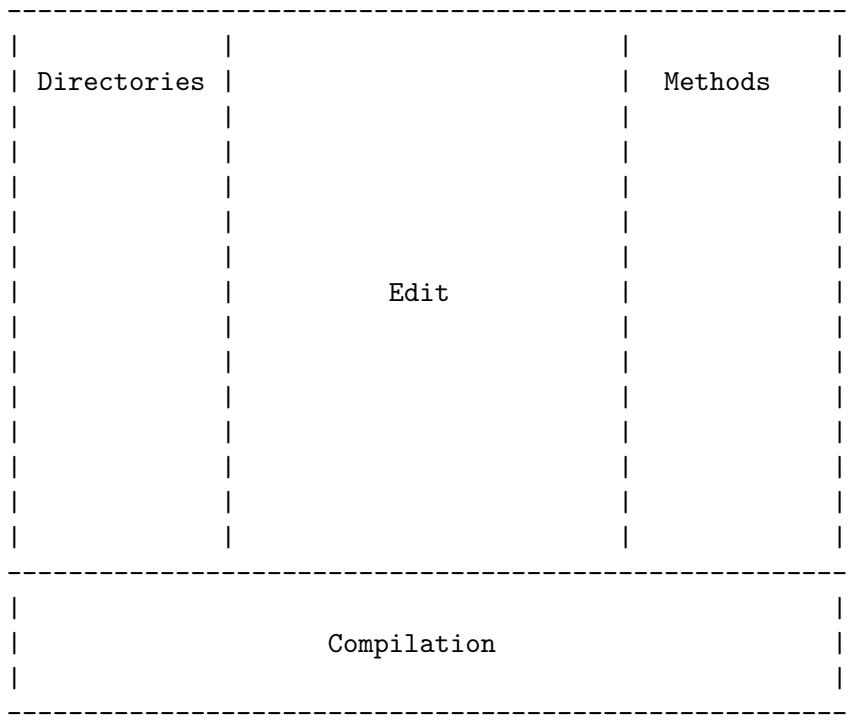
Layout “leftright1”



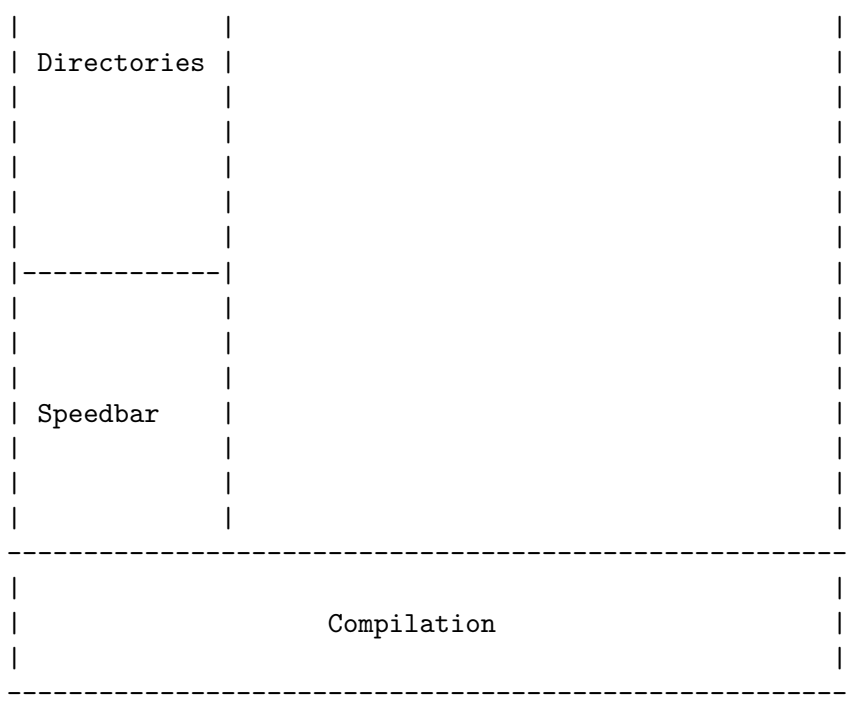
Layout “leftright2”



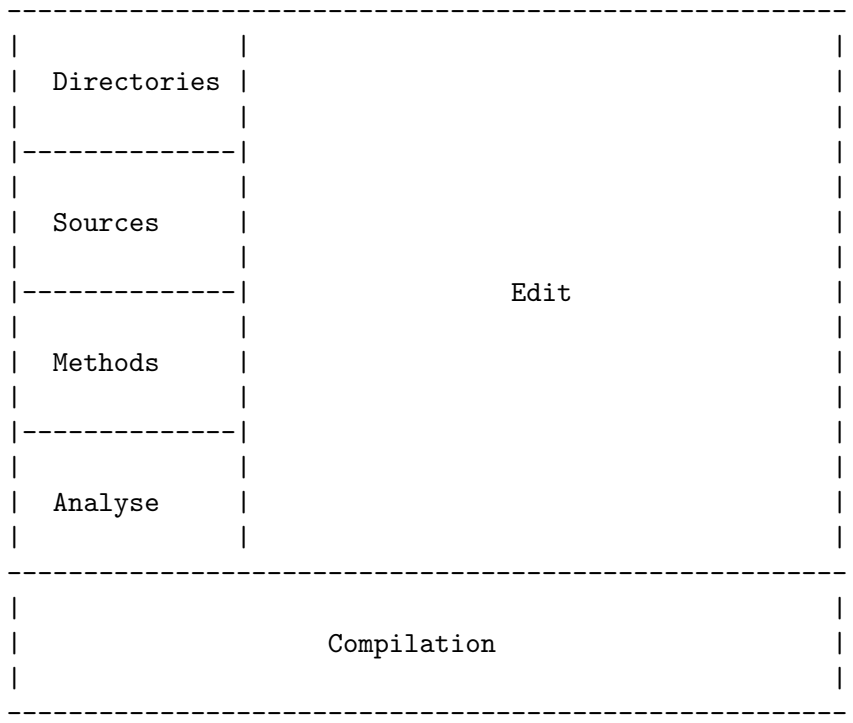
Layout “leftright3”



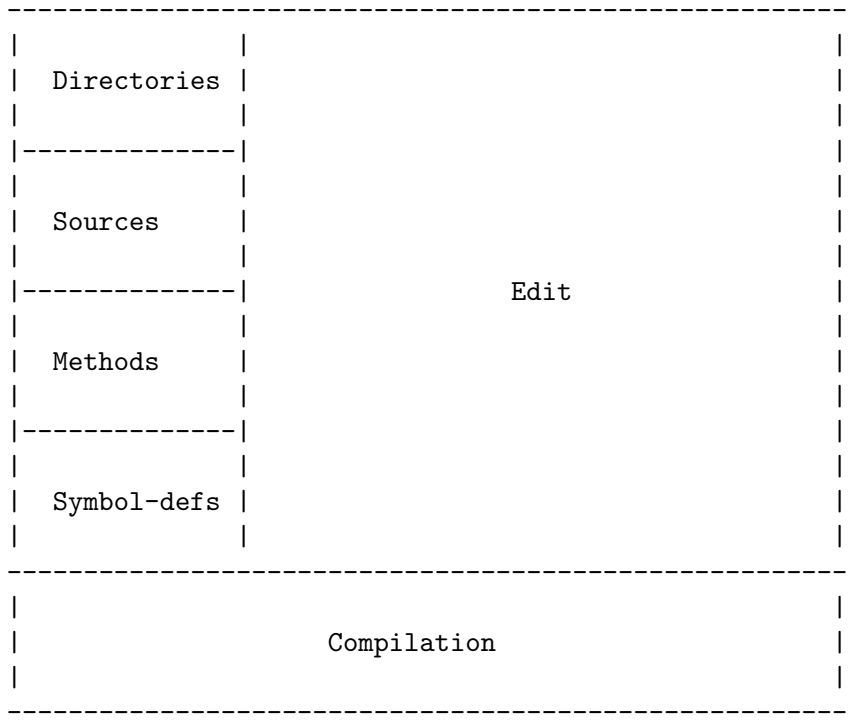
Layout “left-dir-plus-speedbar”



Layout “left-analyse”



Layout “left-symboldef”



6.8.2 Redrawing the ECB-layout

If you have unintentionally destroyed the ECB-layout, you can always restore the layout with calling `ecb-redraw-layout`. This is even true, if you get messages like "wrong-type-argument window-live-p #<window 222>".

If the variable `ecb-redraw-layout-quickly` is not nil then the redraw is done by the `ecb-redraw-layout-quickly` function, otherwise by `ecb-redraw-layout-full`. But it's strongly recommended to use the quick redraw only if you have really slow machines where a full redraw takes several seconds because the quick redraw is not really safe and may have some drawbacks! On normal machines the full redraw should be done in << 1s!

Please read the documentation of the command `ecb-redraw-layout`!

See also the hooks `ecb-redraw-layout-after-hook` and `ecb-redraw-layout-before-hook`!

6.8.3 Changing the sizes of the special ECB-windows

The standard width and height of the special ECB-windows is defined with the options `ecb-windows-width` and `ecb-windows-height`. But changing these options always influences all layouts which is not always desired.

ECB offers to re-adjust the width and height of the ECB-windows (e.g. by dragging the windows-borders via the mouse) and then saving exactly these current window-sizes for the current layout so after activating this layout all windows have autom. the stored sizes.

This is done via the option `ecb-layout-window-sizes` and the commands `ecb-store-window-sizes`, `ecb-restore-window-sizes` and `ecb-restore-default-window-sizes`.

Here is an example how to resize and store the sizes of the ECB-windows of layout "left1":

1. Switch to layout "left1" via `ecb-change-layout` (*C-c . lc*)
2. Resize the ECB-windows by dragging the window-borders with the mouse
3. Call `ecb-store-window-sizes`

After this layout "left1" will be always drawn with the new sizes until you call `ecb-restore-default-window-sizes` during layout "left1" is active.

Please note: `ecb-store-window-sizes` stores the width and height of the windows per default as fractions of the width (rsp. height) of the ECB-frame, so the stored sizes are always correct regardless of the current frame-size! But if called with a prefix argument then fixed sizes are stored.

6.8.4 Fixing the sizes of the special ECB-windows

GNU Emacs 21 introduced a new feature which can fix the sizes of a window displaying a certain buffer even after resizing the frame. This new feature is driven by the new buffer-local variable `window-size-fixed`[8].

ECB offers an option `ecb-fix-window-size` for fixing the sizes of the special ECB-windows/buffers even after frame-resizing. The fix type (valid values are `nil`, `t`, `width` and `height`) can either be set on a layout-basis (means a different value for each layout) or one

[8] Currently XEmacs does not support this feature therefore `ecb-fix-window-size` has no effect with XEmacs

value can be set for all layouts. In the latter case there is an additional value `auto` which choose autom. the senseful fix-type depending on the current layout-type: For top-layouts the fix-type `height` and for all other layout-types the fix-type `width`.

Probably the most senseful value is `auto` for all layouts because it makes less sense to fix the height of the ecb-windows in a left-, right- or leftright-layout. Same for fixing the width in a top-layout.

Note: With Emacs < 22 there seems to be no distinction between `width`, `height` and `t`. Therefore this option takes no effect (means all ecb-windows have always unfixed sizes) with Emacs < 22 if `ecb-compile-window-height` is not `nil`.

6.8.5 Interactively creating new layouts

If you want to create your own ECB-layout then you can do this very easy "by example" with the command `ecb-create-new-layout`. This command creates a new empty frame and offers a small set of keys to create the new layout by splitting windows. `ecb-create-new-layout` and this couple of keys are your guide during the layout-creation-process[9].

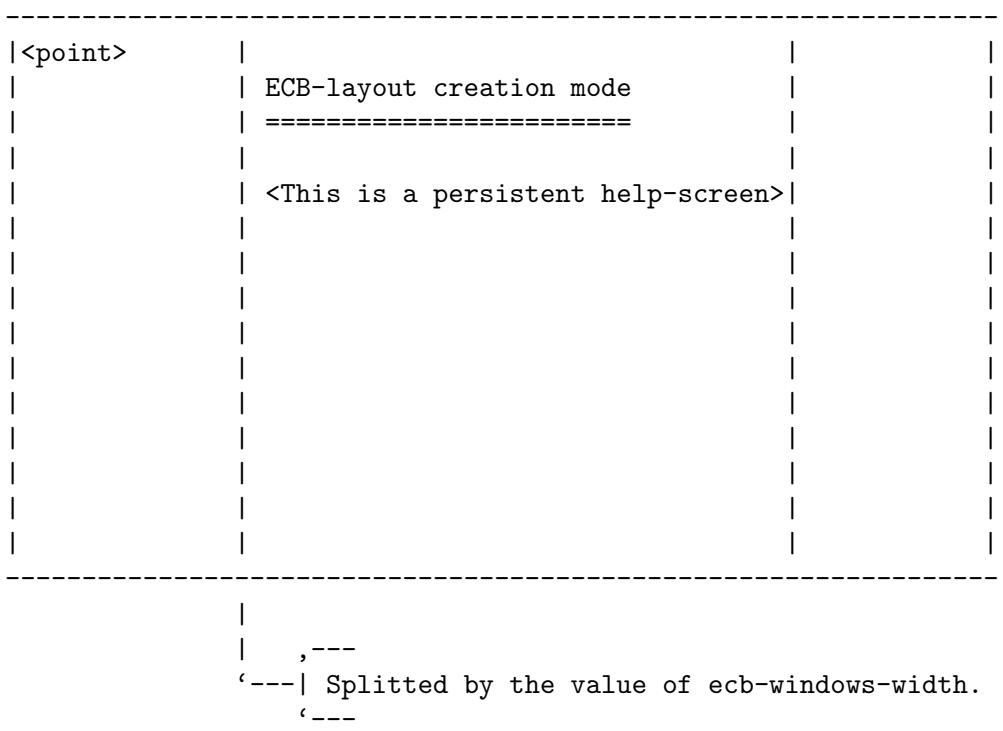
After calling `ecb-create-new-layout` you will be asked which type of layout you want to create: "left", "right", "top" or "left-right". Here you specify where the ECB-tree-windows/buffers should be located in the ECB-frame:

- left: All ECB-tree-windows are located on the left side
- right: All ECB-tree-windows are located on the right side
- top: All ECB-tree-windows are located on the top side
- left-right: All ECB-tree-windows are located on the left and right side

Depending on the type you choose the window is splitted by the values of the options `ecb-windows-width` (types "left", "right" and "left-right") or `ecb-windows-height` (type "top").

Afterwards you will see a frame like follows (here the layout-type is "left-right"):

[9] During the creation process you will be asked in the minibuffer for several options; here you can use TAB-completion and an "empty" RET chooses always the first option!



The big window (here the middle window) will be the edit-area of the new layout and can not be selected, deleted or splitted during the creation process. It displays the help-screen for the layout-creation mode. Here all the available commands are displayed.

The small window(s) (here the left and right windows) can be splitted by you wherever you want (`C-s`). The left one contains the point. You must give every ECB-tree-window you create a type (`C-t`) which can be either

- One of the built-in types

  This can be either "directories", "sources", "methods", "history" or "speedbar".

- Any user-defined type:

  In this case you insert "other" after hitting `C-t` and you will then be asked for the name of the user-defined type. You can insert any arbitrary type name X. But to get this layout working you have to define a function with name `ecb-set-X-buffer` whereas X is the name of the user-defined type you have specified during layout-creation.

  This function `ecb-set-X-buffer` must be defined with the macro `defecb-window-dedicator` and has to switch to the buffer you want to display in this window. This macro automatically ensures that this window is dedicated to that buffer.

  Here is an example: Suppose you have inserted as type name "example" then you have to define and load a function `ecb-set-example-buffer` which could be defined like follows:

```
(defecb-window-dedicator ecb-set-example-buffer " *ECB example-buf*"
  (switch-to-buffer (get-buffer-create " *ECB example-buf*")))
```

  If you forget to define such a function for the user-defined type then nevertheless ECB will draw this layout but it will use the default-function `ecb-set-default-ecb-buffer` instead.

If you are satisfied with your new layout just hit `C-q`. You will be asked for a new layout-name (TAB-completion is offered to get a list of all names already in use) and after inserting a new(!) name the new layout is saved in the file defined by the option `ecb-create-layout-file`. The new layout is now available via the option `ecb-layout-name`.

There is no need for you to load the file `ecb-create-layout-file` manually into your Emacs because it's automatically loaded by ECB!

Please note: During the layout-creation process only the commands displayed in the help-screen are available. ALL other commands are temporally disabled (even the mouse-commands).

For programming new layouts with emacs-lisp see ⟨undefined⟩ [The layout-engine], page ⟨undefined⟩.

With the command `ecb-delete-new-layout` you can delete previously created layouts (TAB-completion is offered for all names of user created layouts).

6.9 Hiding/Showing the ECB windows

With `ecb-toggle-ecb-windows`, `ecb-hide-ecb-windows` and `ecb-show-ecb-windows` you can hide/show all the ECB windows without changing the activation state of ECB and also without deactivating the advices for `delete-other-windows` and/or `delete-window`. This is most useful if you use a layout like "top2" (see ⟨undefined⟩ [Tips and tricks], page ⟨undefined⟩) or if you want to have maximum space for editing and you don't need the browsing windows all the time.

The following sequence of hooks is evaluated during showing again the hidden ECB-windows:

1. `ecb-show-ecb-windows-before-hook`
2. `ecb-redraw-layout-before-hook`
3. <Redrawing the layout to show the hidden ECB-windows>
4. `ecb-redraw-layout-after-hook`
5. `ecb-show-ecb-windows-after-hook`

The following sequence of hooks is evaluated during hiding the ECB-windows:

1. `ecb-hide-ecb-windows-before-hook`
2. `ecb-redraw-layout-before-hook`
3. <Hiding the ECB-windows>
4. `ecb-redraw-layout-after-hook`
5. `ecb-hide-ecb-windows-after-hook`

If the special ECB-windows are hidden (e.g. by 'ecb-toggle-ecb-windows') all adviced functions behave as their originals. So the frame can be used as if ECB would not be active but ECB IS still active in the "background" and all ECB-commands and all ECB-keybindings can be used. Of course some of them doesn't make much sense but nevertheless they can be called. Toggling the visibility of the ECB-windows preserves the splitting-state of the edit-area: If you hide the ECB-windows then the frame will be divided in the same window-layout the edit-area had before the hiding and if you show the ECB-windows

again the edit-area will be divided into all the edit-windows the ECB-frame had before the showing.

Therefore it should be enough to hide the ECB-windows to run other Emacs-applications which have their own window-layout-managing. There should be no conflicts. But nevertheless the most recommended method for running ECB and other applications (e.g. xrefactory, Gnus etc.) in the same frame is to use a window-manager like winring.el or escreen.el (see ⟨undefined⟩ [Window-managers and ECB], page ⟨undefined⟩).

6.10 Maximizing the ECB windows

6.10.1 How to maximize and minimize special ecb-tree-windows

To get a better overlook about the contents of a certain ECB-window every ECB-window can be "maximized", means all other ECB-windows are deleted so only the edit-window(s) and this maximized ECB-window are visible (and maybe a compile-window if active). There are several ways to do this:

- Via the node-popup-menus of the ECB-windows
- Via the main "ECB"-menu and here "Display window maximized"
- Via calling the adviced version of `delete-other-windows`[10] (bound to *`C-x 1`*) in one of the ECB windows.
- Via one of the commands `ecb-maximize-window-directories`, `ecb-maximize-window-sources`, `ecb-maximize-window-methods`, `ecb-maximize-window-history` or `ecb-maximize-window-speedbar` or the bound short-cuts for those commands.
- Via the new command `ecb-cycle-maximized-ecb-buffers` which cycles through all ecb-buffers of current layout by maximizing exactly one of the ecb-windows after every cycle-step.
- Via the option `ecb-maximize-ecb-window-after-selection` and then just by selecting an ECB-window. "Deselecting" an ECB-window brings back all ECB-windows of current layout.
- Via the default modeline-mechanisms for deleting other windows. GNU Emacs binds *`mouse-2`* in its modeline to `delete-other-window`. XEmacs binds a popup-menu with some window commands to *`button-3`* in its modeline.

  ECB combines the best of both worlds by supporting both of these mechanisms for both Xemacs and Emacs: ECB binds a toggle-command to *`mouse-2`* in the modeline of each tree-buffer which maximizes the current tree-window if all ECB-windows are visible and displays all ECB-windows if current tree-window is maximized. In addition ECB binds a popup-menu to *`mouse-3`* which offers exactly 2 commands: Maximizing current tree-window and displaying all ECB-windows.

"Minimizing" such a maximized ECB-window, i.e. bringing back to its original size and displays all ecb-windows of current layout, can simply be done by redrawing the layout via the command `ecb-redraw-layout` (bound to *`C-c . lr`*).

[10] This command is adviced per default, see ⟨undefined⟩ [The edit-area], page ⟨undefined⟩.

6.10.2 Selecting a node in a maximized ecb-tree-window

When you select a node (either via mouse or RET) in a maximized tree-window the default behavior of ECB is the following:

Maximized directories-window: When selecting a directory then first automatically the maximized directories-window will be "minimized" (i.e. all ecb-windows of current layout are displayed) if the current layout contains a sources-buffer and no sources are shown in the directories-window - see `ecb-show-sources-in-directories-buffer`. So the source-files can be displayed in the sources-window.

Maximized sources- or history-window: When selecting a source-file in one of these buffers then first automatically the maximized window will be "minimized" (i.e. all ecb-windows of current layout are displayed) if the current layout contains a methods-buffer. So the tag-contents of the selected source-file can be displayed in the methods-window.

For a even smarter behavior ECB offers the option `ecb-maximize-next-after-maximized-select` which automatically maximizes the next logical tree-window after a node selection. The definition of "next logical is": Directories –> sources, sources/history –> methods. But if the current maximized tree-buffer is also contained in the option `ecb-tree-do-not-leave-window-after-select` (see also the tree-buffer-command `ecb-toggle-do-not-leave-window-after-select` which is bound to C-T in each tree-buffer) then ECB does *not* maximize the next logical tree-window but point stays in the currently maximized tree-buffer so for example the user can select more than one source-file from the sources-buffer.

6.11 Back- and forward navigation like a browser

With ECB you can "browse" in your source-files like with a web-browser. This means ECB stores the current buffer- and window-position relative to the current tag[11] in the edit-window after

- selecting a tag in the ECB-methods buffer or
- selecting a source-file in the ECB-sources/history-buffer.

ECB offers two commands `ecb-nav-goto-next` (*C-c . n*) and `ecb-nav-goto-previous` (*C-c . p*) to go forward and backward within this navigation history-list. These commands are also available via the menu "ECB –> Navigate".

Aside normal "location-browsing" this is useful for example in a scenario where the buffer is narrowed to a tag (see `ecb-tag-visit-post-actions`):

1. You edit a function
2. Goto another function above the current in the same file
3. Add a few lines
4. Call ecb-nav-goto-previous

   Now you will edit at the same place in the function.

[11] e.g. a method, a variable or any other semantic tag

6.12 Synchronization of the ECB-windows

Per default ECB synchronizes automatically the contents of the ECB-windows/tree-buffers with the current active edit-window (rsp. the current buffer of the edit window):

- ECB-directories:

  This windows is synchronized to display the directory where the source-file which is displayed in the current active edit-window is located. If the source-path (i.e. an element of the option `ecb-source-path`) containing this directory is not expanded it will be auto. expanded according to the value of the option `ecb-auto-expand-directory-tree` (see ⟨undefined⟩ [ecb-directories], page ⟨undefined⟩).

- ECB-sources:

  The ECB-sources-buffer contains after synchronizing all the sources of the directory of the "current" source-file displayed in the edit-window. The entry of the "current" source-file is highlighted.

- ECB-methods:

  Contains after synchronizing all the tags of the buffer in the current selected edit-window, i.e. all methods, variables etc... depending on the major-mode.

- ECB-history:

  Highlights the entry of the buffer displayed in the current active edit-window if this buffer is a source-file.

This feature can be customized with the option `ecb-window-sync`:

If active then the synchronization takes place whenever a buffer changes in an edit window or if another edit-window with another buffer will be selected, if deactivated then never. But you can also set this option to a list of major-modes and then the sync. will only be done if the major-mode of the current buffer does NOT belong to this list.

But in every case the synchronization only takes place if the major-mode of the current-buffer in the current selected edit-window has a relation to files or directories. Examples for the former one are all programming-language-modes like `c++-mode` or `java-mode`, `Info-mode` too, an example for the latter one is `dired-mode`. For all major-modes related to non-file/directory-buffers like `help-mode`, `customize-mode` and others a synchronization will never be done!

It's recommended to exclude at least `Info-mode` because it makes no sense to synchronize the ECB-windows after calling the Info help. Per default also `dired-mode` is excluded but it can also making sense to synchronize the ECB-directories/sources windows with the current directory of the dired-buffer in the edit-window.

If you often need to toggle between autom. synchronization on and off then customizing the option `ecb-window-sync` is inefficient and therefore ECB offers the command `ecb-toggle-window-sync`.

Please note: With the command `ecb-window-sync` you can do a manual synchronization if the automatic one is switched off or if you just want to do this!

6.13 Stealthy background-tasks of ECB

ECB performs some tasks stealthily in the background and also interruptable by the user because these tasks can be time-consuming and could otherwise block ECB. Currently the following tasks are performed stealthily and in the background by ECB:

Prescann directories for emptyness
: Prescann directories and display them as empty or not-empty in the directories-buffer. See the documentation of the option `ecb-prescan-directories-for-emptyness` for a description.

File is read only
: Check if sourcefile-items of the directories- or sources-buffer are read-only or not. See documentation of the option `ecb-sources-perform-read-only-check`.

Version-control-state
: Checks the version-control-state of files in directories which are managed by a VC-backend. See the option `ecb-vc-enable-support`.

All of these tasks (e.g. checking if a directory is empty or not) perform a certain action for all directories or sources displayed in the current visible tree-buffers of ECB. Normally there should be no annoying delay for the user because each of these tasks will be only performed when Emacs is idle and will be interrupted immediatelly when a user hits a key or clicks the mouse but especially for remote-directories one single action (e.g. checking if a certain directory is empty or checking the VC-state of a sourcefile in such a remote directory) can be very time-consuming and such a single action is not interruptable (an interrupt can only occur between the single-actions for two directories or sources) For a further discussion how to deal best with remote directories see ⟨undefined⟩ [Remote directories], page ⟨undefined⟩.!

ECB offers for all stealthy tasks three steps of activation:

- `t`: Switch on this feature.
- `unless-remote`: Switch on this feature but not for remote directories. The term "remote" means here directories which are used via tramp, ange-ftp or efs. So mounted directories are counted not as remote directories here even if such a directory is maybe hosted on a remote machine. But normally only directories in a LAN are mounted so there should be no performance-problems with such mounted directories.
- `nil`: Switch off this feature completely.

In combination with the option `ecb-stealthy-tasks-delay` these three choices already allow adapting the stealthy tasks to most needs. But to offer finest granularity for which directories a certain stealthy task should be switched on and for which not ECB offers for every stealthy task an additional option which allows a finer adjustment:

- Prescanning directories for emptyness: `ecb-prescan-directories-exclude-regexps`.
- Checking the read-only-state of a sourcefile: `ecb-read-only-check-exclude-regexps`
- Checking the VC-state of sourcefiles: `ecb-vc-directory-exclude-regexps`

These options take only effect when the related task is not completely switched off. If this is the case they allow excluding certain directories (or the sources of directories) from being processed by a certain stealthy task.

6.14 Interactive ECB commands

ECB offers a lot of interactive commands. Some of these commands prompt the user in the minibuffer if called with a prefix argument.

Example: If `ecb-clear-history` is called with a prefix argument then you will be prompted in the minibuffer with:

```
Clear from history: [all, not-existing-buffers, existing-buffers]
```

You can choose one of the options enclosed in brackets with TAB-completion; hitting RET direct after the prompt chooses auto. the first offered option (in the example above "all").

Please note: The following interactive commands of ECB are listed without the prefix "ecb-" (e.g. the command `ecb-activate` is listed with name "activate"). This has been done for a better readable command index. See ⟨undefined⟩ [Command Index], page ⟨undefined⟩.

`activate` [Command]
: Activates ECB and creates the special buffers for the choosen layout. For the layout see `ecb-layout-name`. This function always raises the ECB-frame if called from another frame. This is the same as calling `ecb-minor-mode` with a positive argument.

`add-all-buffers-to-history` [Command]
: Add all current file-buffers to the history-buffer of ECB. Dependend on the value of `ecb-history-sort-method` afterwards the history is sorted either by name or by extension. If `ecb-history-sort-method` is nil the most recently used buffers are on the top of the history and the seldom used buffers at the bottom.

`analyse-buffer-sync` [Command]
: Synchronize the analyse buffer with the current buffer and point. This means in fact display the current analysis for current point.

`change-layout` **&optional** *preselect-type* [Command]
: Select a layout-name from all current available layouts (TAB-completion is offered) and change the layout to the selected layout-name. If optional argument PRESELECT-TYPE is not nil then you can preselect a layout-type \(TAB-completion is offered too) and then you will be asked only for layouts of that preselected type. Note: This function works by changing the option `ecb-layout-name` but only for current Emacs-session.

`clear-history` [Command]
: Clears the history-buffer.

`customize` [Command]
: Open a customize-buffer for all customize-groups of ECB.

`customize-most-important` [Command]
: Open a customize-buffer for the most important options of ECB.

`create-new-layout` [Command]
: Start process for interactively creating a new ECB-layout (see ⟨undefined⟩ [Creating a new ECB-layout], page ⟨undefined⟩).

`cycle-maximized-ecb-buffers` [Command]
Cycles through all ecb-buffers of current layout by maximizing exactly one of the ecb-windows after every cycle-step.

`cycle-through-compilation-buffers` **&optional** *choose-buffer* [Command]
Cycle through all compilation buffers currently open and display them within the compilation window `ecb-compile-window`. If the currently opened buffer within the compilation window is not a compilation buffer, we jump to the first compilation buffer. If not we try to loop through all compilation buffers. If we hit the end we go back to the beginning.

If *CHOOSE-BUFFER* is not `nil` then the user will be prompted for the compilation-buffer to switch to.

`deactivate` [Command]
Deactivates the ECB and kills all ECB buffers and windows.

`delete-new-layout` [Command]
Select a layout-name for a layout created by `ecb-create-new-layout` and delete this layout. This means the layout-definition is removed from the file `ecb-create-layout-file` and the layout-function and associated aliases are unbound.

`display-news-for-upgrade` **&optional** *FULL-NEWS* [Command]
Display the most important NEWS after an ECB-upgrade. If you call this function but no ECB-upgrade has been performed before starting ECB then nothing is display unless *FULL-NEWS* is not nil.

If *FULL-NEWS* is not nil then the NEWS-file is displayed in another window.

`display-upgraded-options` [Command]
Display a information-buffer which options have been upgraded or reset. Offers two buttons where the user can decide if the upgraded options should also being saved by ECB for future settings or if the buffer should be killed.

If saving is possible this command display where the options would be saved. It is that file Emacs uses to save customize-settings. This file is "computed" from the settings in `custom-file` and `user-init-file` (see the documentation of these variables).

ECB automatically makes a backup-file of that file which will be modified by storing the upgraded rsp. renamed ECB-options. This backup file gets a unique name by adding a suffix ".before_ecb_<version>" to the name of the modified file. If such a file already exists ECB adds a unique number to the end of the filename to make the filename unique. This is a safety mechanism if something fails during storing the upgraded options, so you never lose the contents of your customization-file!

`download-ecb` [Command]
Download ECB from the ECB-website and install it. For this the option `ecb-download-url` must be set correct, whereas the default value of this option should always be correct.

If `ecb-download-package-version-type` is set to -1 (means asking for a version) then you will be ask in the minibuffer for the version to download. Otherwise ECB

downloads autom. the latest version available for the type specified in `ecb-download-package-version-type`. If no newer version than the current one is available no download will be done.

For details about downloading and what requirements must be satisfied see function `ecb-package-download` and option `ecb-download-package-version-type`!

After successful downloading the new ECB will be installed in a subdirectory of `ecb-download-install-parent-dir`. After adding this subdirectory to `load-path` and restarting Emacs the new ECB version can be activated by `ecb-activate`.

If current running ECB is installed as regular XEmacs-package and not with the archive available at the ECB website then this function asks for proceeding!

`download-semantic` [Command]

Download semantic from the semantic-website and install it. For this the variable `ecb-cedet-url` must be set correct, whereas the default value of this variable should always be correct.

If `ecb-download-package-version-type` is set to -1 (means asking for a version) then you will be ask in the minibuffer for the version to download. Otherwise ECB downloads autom. the latest version available for the type specified in `ecb-download-package-version-type`. If no newer version than the current one is available no download will be done.

For details about downloading and what requirements must be satisfied see function `ecb-package-download` and option `ecb-download-package-version-type`!

After successful downloading the new semantic will be installed in a subdirectory of `ecb-download-install-parent-dir`. After adding this new subdirectory to `load-path` and restarting Emacs the new semantic version is loaded and is used after next start of ECB.

If current running semantic is installed as regular XEmacs-package and not with the archive available at the semantic website then this function asks for proceeding!

`expand-methods-nodes` **&optional** *force-all* [Command]

Set the expand level of the nodes in the ECB-methods-buffer.

This command asks in the minibuffer for an indentation level LEVEL. With this LEVEL you can precisely specify which level of nodes should be expanded. LEVEL means the indentation-level of the nodes.

A LEVEL value X means that all nodes with an indentation-level <= X are expanded and all other are collapsed. A negative LEVEL value means all visible nodes are collapsed.

Nodes which are not indented have indentation-level 0!

Which node-types are expanded (rsp. collapsed) by this command depends on the options `ecb-methods-nodes-expand-spec` and `ecb-methods-nodes-collapse-spec`! With optional argument *FORCE-ALL* all tags will be expanded/collapsed regardless of the values of these options.

Examples:

- LEVEL = 0 expands only nodes which have no indentation itself.

- LEVEL = 2 expands nodes which are either not indented or indented indented once or twice
- LEVEL ~ 10 should normally expand all nodes expect there are nodes which are indented deeper than 10.

Note 1: This command switches off auto. expanding of the method-buffer if `ecb-expand-methods-switch-off-auto-expand` is not nil. But it can be switched on again quickly with `ecb-toggle-auto-expand-tag-tree` or *`[C-c . a]`*.

Note 2: All this is only valid for file-types parsed by semantic. For other file types which are parsed by imenu or etags (see `ecb-process-non-semantic-files`) *FORCE-ALL* is always true!

`dump-semantic-toplevel` [Command]
Dump the current semantic-tags in special buffer and display them.

`eshell-current-buffer-sync` [Command]
Synchronize the eshell with the directory of current source-buffer. This is only done if the eshell is currently visible in the compile-window of ECB and if either this function is called interactively or `ecb-eshell-synchronize` is not nil.

`eshell-recenter` [Command]
Recenter the eshell window so that the prompt is at the buffer-end.

`expand-directory-nodes` [Command]
Set the expand level of the nodes in the ECB-directories-buffer. For argument LEVEL see `ecb-expand-methods-nodes`.

Be aware that for deep structured paths and a lot of source-paths this command can last a long time - depending on machine- and disk-performance.

`goto-window-analyse` [Command]
Make the ECB-analyse window the current window.

`goto-window-compilation` [Command]
Goto the ecb compilation window `ecb-compile-window`.

`goto-window-directories` [Command]
Make the ECB-directories window the current window. If `ecb-use-speedbar-instead-native-tree-buffer` is `dir` then goto to the speedbar-window.

`goto-window-edit1` [Command]
Make the (first) edit-window window the current window.

`goto-window-edit2` [Command]
Make the second edit-window (if available) window the current window.

`goto-window-edit-last` [Command]
Make the last selected edit-window window the current window. This is the same as if `ecb-mouse-click-destination` is set to `last-point`.

`goto-window-history` [Command]
Make the ECB-history window the current window.

`goto-window-methods` [Command]
Make the ECB-methods window the current window. If `ecb-use-speedbar-instead-native-tree-buffer` is `method` then goto to the speedbar-window.

`goto-window-sources` [Command]
Make the ECB-sources window the current window. If `ecb-use-speedbar-instead-native-tree-buffer` is `source` then goto to the speedbar-window.

`history-filter` [Command]
Apply a filter to the history-buffer to reduce the number of entries. So you get a better overlooking. There are three choices:

- Filter by extension: Just insert the extension you want the History-buffer being filtered. Insert the extension without leading dot!
- Filter by regexp: Insert the filter as regular expression.
- No filter: This means to display an entry for all currently living file-buffers.

`jde-display-class-at-point` [Command]
Display in the ECB-methods-buffer the contents (methods, attributes etc...) of the class which contains the definition of the "thing" under point (this can be a variable-name, class-name, method-name, attribute-name). This function needs the same requirements to work as the method-completion feature of JDEE (see `jde-complete`)!. The source-file is searched first in `jde-sourcepath`, then in `jde-global-classpath`, then in *$CLASSPATH*, then in current-directory.

Works only for classes where the source-code (i.e. the *.java-file) is available.

`maximize-window-analyse` [Command]
Maximize the ECB-analyse-window. I.e. delete all other ECB-windows, so only one ECB-window and the edit-window(s) are visible (and maybe a compile-window). Works also if the ECB-analyse-window is not visible in current layout.

`maximize-window-directories` [Command]
Maximize the ECB-directories-window, i.e. delete all other ECB-windows, so only one ECB-window and the edit-window(s) are visible (and maybe a compile-window). Works also if the ECB-directories-window is not visible in current layout.

`maximize-window-sources` [Command]
Maximize the ECB-sources-window, i.e. delete all other ECB-windows, so only one ECB-window and the edit-window(s) are visible (and maybe a compile-window). Works also if the ECB-sources-window is not visible in current layout.

`maximize-window-methods` [Command]
Maximize the ECB-methods-window, i.e. delete all other ECB-windows, so only one ECB-window and the edit-window(s) are visible (and maybe a compile-window). Works also if the ECB-methods-window is not visible in current layout.

`maximize-window-history` [Command]
Maximize the ECB-history-window, i.e. delete all other ECB-windows, so only one ECB-window and the edit-window(s) are visible (and maybe a compile-window). Works also if the ECB-history-window is not visible in current layout.

`maximize-window-speedbar` [Command]

Maximize the ECB-speedbar-window, i.e. delete all other ECB-windows, so only one ECB-window and the edit-window(s) are visible (and maybe a compile-window). Does nothing if the speedbar-window is not visible within the ECB-frame.

`methods-filter` [Command]

Apply a filter to the Methods-buffer to reduce the number of entries. So you get a better overlooking. There are six choices:

- Filter by protection: Just insert the protection you want the Methods-buffer being filtered: private, protected or public!
- Filter by regexp: Insert the filter as regular expression.
- Filter by tag-class: You can filter by the tag-classes of current major-mode. The available tag-classes come from the variable `semantic--symbol->name-assoc-list`. The are normally methods, variables etc.
- Filter by current type: In languages which have types like Java or C++ this filter displays only the current type and all its members (e.g. attributes and methods). If ECB can not identify the current type in the source-buffer or in the methods-window then nothing will be done.
- Filter by a filter-function: Such a function gets two arguments: a tag and the source-buffer of this tag. If the tag should be displayed (i.e. not being filtered out) then the function has to return not nil otherwise nil.
- No special filter: This means to display all tags specified with the option `ecb-show-tokens`. If currently some of the above filters are applied they will be all removed.
- Delete the last added: This removes only the topmost filter-layer, means that filter added last.

The protection-, current-type- and the tag-class-filter are only available for semantic-supported sources.

Be aware that the tag-list specified by the option `ecb-show-tags` is the basis of all filters, i.e. tags which are excluded by that option will never be shown regardless of the filter type here!

All tags which match the applied filter(s) will be displayed in the Methods-buffer.

If called with a prefix-argument or when optional arg INVERSE is not nil then an inverse filter is applied to the Methods-buffer, i.e. all tags which do NOT match the choosen filter will be displayed in the Methods-buffer!

Per default the choosen filter will be applied on top of already existing filters. This means that filters applied before are combined with the new filter. This behavior can changed via the option `ecb-methods-filter-replace-existing`. But regardless of the setting in `ecb-methods-filter-replace-existing` applying one of the not-inverse filters protection, tag-class or current-type always replaces exactly already existing filters of that type. On the other hand applying more than one inverse tag-class- or protection-filter can make sense.

Such a filter is only applied to the current source-buffer, i.e. each source-buffer can have its own tag-filters.

The current active filter will be displayed in the modeline of the Methods-buffer [regexp, prot (= protection), tag-class, function (= filter-function)]. If an inverse filter has been applied then this is signalized by a preceding caret `^`. If currently more than 1 filter is applied then always the top-most filter is displayed in the modeline but the fact of more than 1 filter is visualized by the number of the filters - included in parens. You can see all currently applied filters by moving the mouse over the filter-string in modeline of the Methods-buffer: They will displayed as help-echo.

See the option `ecb-default-tag-filter` if you search for automatically applied default-tag-filters.

`methods-filter-current-type` [Command]
Display in the Methods-buffer only the current type and its members. For further details see `ecb-methods-filter`.

`methods-filter-delete-last` [Command]
Remove the most recent filter from the Methods-buffer. For further details see `ecb-methods-filter`.

`methods-filter-function` **&optional** *inverse* [Command]
Filter the methods-buffer by a function. If INVERSE is not nil (called with a prefix arg) then an inverse filter is applied. For further details see `ecb-methods-filter`.

`methods-filter-nofilter` [Command]
Remove any filter from the Methods-buffer. For further details see `ecb-methods-filter`.

`methods-filter-protection` **&optional** *inverse* [Command]
Filter the methods-buffer by protection. If INVERSE is not nil (called with a prefix arg) then an inverse filter is applied. For further details see `ecb-methods-filter`.

`methods-filter-regexp` **&optional** *inverse* [Command]
Filter the methods-buffer by a regexp. If INVERSE is not nil (called with a prefix arg) then an inverse filter is applied. For further details see `ecb-methods-filter`.

`methods-filter-tagclass` **&optional** *inverse* [Command]
Filter the methods-buffer by tag-class. If INVERSE is not nil (called with a prefix arg) then an inverse filter is applied. For further details see `ecb-methods-filter`.

`minor-mode` **&optional** *arg* [Command]
Toggle ECB minor mode. With prefix argument *ARG*, turn on if positive, otherwise off. Return non-`nil` if the minor mode is enabled.

`nav-goto-previous` [Command]
Go backward in the navigation history-list, see ⟨undefined⟩ [Back/forward navigation], page ⟨undefined⟩.

`nav-goto-next` [Command]
Go forward in the navigation history-list, see ⟨undefined⟩ [Back/forward navigation], page ⟨undefined⟩.

`rebuild-methods-buffer` [Command]

Updates the methods buffer with the current buffer after deleting the complete previous parser-information, means no semantic-cache is used! Point must stay in an edit-window otherwise nothing is done. This method is merely needed for semantic parsed buffers if semantic parses not the whole buffer because it reaches a not parse-able code or for buffers not supported by semantic but by imenu or etags.

Examples when a call to this function can be necessary:

- If an Elisp-file is parsed which contains in the middle a defun X where the closing) is missing then semantic parses only until this defun X is reached and you will get an incomplete ECB-method buffer. In such a case you must complete the defun X and then call this function to completely reparse the Elisp-file and rebuild the ECB method buffer!
- For not semantic supported buffers which can be parsed by imenu or etags (see `ecb-process-non-semantic-files`) because for these buffers there is no built-in auto-rebuild mechanism. For these buffers this command calls `ecb-rebuild-methods-buffer-for-non-semantic`.

For non-semantic-sources supported by etags the option `ecb-auto-save-before-etags-methods-rebuild` is checked before rescanning the source-buffer and rebuilding the methods-buffer.

If point is in one of the ecb-windows or in the compile-window then this command rebuids the methods-buffer with the contents of the source-buffer the last selected edit-window.

`redraw-layout` **&optional** *ARG* [Command]

Redraw the ECB screen.

Do not call this command from elisp-program but only interactively!

Called without a prefix-argument the state of the ECB-frame-layout will preserved. This means:

- The state of compile-window (hidden or visible) will be preserved but if visible then the height will be as specified in `ecb-compile-window-height`.
- The state of the ECB-windows will be preserved (hidden or visible) but if visible then the sizes will be as specified in the layout (and with the options `ecb-windows-width` and `ecb-windows-height`) or as stored with `ecb-store-window-sizes`.

If called with ONE prefix-argument (*`[C-u]`*) then the layout will be drawn with all ECB-windows and also with a visible compile-window (when `ecb-compile-window-height` is not nil). The splitting-state of the edit-area will be preserved.

If called with TWO prefix-arguments (i.e. hitting *`[C-u]`* twice: (*`[C-u] [C-u]`*) then an emergency-redraw will be performed. This means the same as if called with one prefix-argument (s.a.) but the splitting-state of the edit-area will NOT be preserved but all edit-windows besides the current one will be deleted. Use this only if there are some anomalies after standard redraws!

If the variable `ecb-redraw-layout-quickly` is not nil then the redraw is done by the `ecb-redraw-layout-quickly` function, otherwise by `ecb-redraw-layout-full`.

Please not: It's strongly recommended to use the quick redraw only if you have really slow machines where a full redraw takes several seconds because the quick redraw is not really safe and has some annoying drawbacks! On normal machines the full redraw should be done in << 1s so there should be no need for the quick version!

`restore-default-window-sizes` [Command]
Resets the sizes of the ECB windows to their default values.

`restore-window-sizes` [Command]
Sets the sizes of the ECB windows to their stored values. See option `ecb-layout-window-sizes` and command `ecb-store-window-sizes`.

`select-ecb-frame` [Command]
Selects the `ecb-frame` if ECB is activated - otherwise reports an error.

`show-help` **&optional** *format* [Command]
Shows the online help of ECB either in Info or in HTML format depending on the value of `ecb-show-help-format`. If called with prefix argument, i.e. if *FORMAT* is not nil then the user is prompted to choose the format of the help (Info or HTML). If an error about not finding the needed help-file occurs please take a look at the options `ecb-help-info-start-file` and `ecb-help-html-start-file`!

Note: If you got ECB as a standard XEmacs-package maybe the HTML-online-documentation is not included.

`show-layout-help` [Command]
Select a name of a layout and shows the documentation of the associated layout-function. At least for the built-in layouts the documentation contains a picture of the outline of the chosen layout.

`show-tip-of-the-day` [Command]
Show tip of the day if `ecb-tip-of-the-day` is not nil or if called interactively.

`sources-filter` [Command]
Apply a filter to the sources-buffer to reduce the number of entries. So you get a better overlooking. There are three choices:

- Filter by extension: Just insert the extension you want the Sources-buffer being filtered. Insert the extension without leading dot!
- Filter by regexp: Insert the filter as regular expression.
- No filter: This means to display an entry for every file in the current selected directory (all except these filter already filtered out by `ecb-source-file-regexps` and `ecb-sources-exclude-cvsignore`).

Such a filter is only applied to the current selected directory, i.e. each directory has its own filtered sources-buffer.

`store-window-sizes` **&optional** *FIX* [Command]
Stores the sizes of the ECB windows for the current layout. The size of the ECB windows will be set to their stored values when `ecb-redraw-layout` or `ecb-restore-window-sizes` is called. To reset the window sizes to their

default values call `ecb-restore-default-window-sizes`. Please read also the documentation of `ecb-layout-window-sizes`!

The windows sizes are stored per default as fractions of current frame-width and -height of the ecb-frame, so the stored values will "work" for other frame sizes too. If a permanent compile-window is visible then ECB will tell you that window-sizes should be stored with hidden compile-window and ask you if you want proceed; if you proceed then the window-heights will be stored as fractions of current (frame-height minus current visible compile-window-height) so you should ensure that the current compile-window has its standard-height as specified in `ecb-compile-window-height`!. If *FIX* is not nil (means called with a prefix argument) then always the fixed values of current width and height are stored!

`submit-problem-report` [Command]
Submit a problem report for the ECB to the ECB mailing-list. This command generates in the edit-window a problem-report which contains already the current values of all ECB options, the current backtrace-buffer if there is any and the current message-buffer. You will be asked for a problem-report subject and then you must insert a description of the problem. Please describe the problem as detailed as possible!

`toggle-auto-expand-tag-tree` **&optional** *arg* [Command]
Toggle auto expanding of the ECB-methods-buffer. With prefix argument *ARG*, make switch on if positive, otherwise switch off. If the effect is that auto-expanding is switched off then the current value of `ecb-auto-expand-tag-tree` is saved so it can be used for the next switch on by this command.

`toggle-compile-window` **&optional** *arg* [Command]
Toggle the visibility of the compile-window of ECB. With prefix argument ARG, make visible if positive, otherwise invisible. The height of the compile-window is always the current value of `ecb-compile-window-height`! If called and `ecb-compile-window-height` is nil then ECB asks for the height of the compile-window, sets this height as new value of `ecb-compile-window-height` and displays the compile-window (so if you have called this command by mistake and you do not want a compile-window you have to quit with C-G).

`toggle-compile-window-height` **&optional** *arg* [Command]
Toggle whether the `ecb-compile-window` is enlarged or not. If *ARG* `>` 0 then shrink or enlarge the the compile-window according to the value of `ecb-enlarged-compilation-window-max-height`. But never shrink below the value of `ecb-compile-window-height`. If *ARG* `<=` 0 then shrink `ecb-compile-window` to `ecb-compile-window-height` and if *ARG* is nil then toggle the enlarge-state.

`toggle-ecb-windows` **&optional** *arg* [Command]
Toggle visibility of the ECB-windows. With prefix argument *ARG*, make visible if positive, otherwise invisible. This has nothing to do with (de)activating ECB but only affects the visibility of the ECB windows. ECB minor mode remains active!

`toggle-layout` **&optional** *last-one* [Command]
Toggles between the layouts defined in `ecb-toggle-layout-sequence` (See also option `ecb-show-sources-in-directories-buffer`). Note: This function works by changing the options `ecb-layout-name` but only for current Emacs-session.

If optional argument *LAST-ONE* is not nil (e.g. called with a prefix-arg) then always the last selected layout was choosen regardless of the setting in `ecb-toggle-layout-sequence`. The last selected layout is always that layout which was current direct before the most recent layout-switch. So now a user can switch to another layout via 'ecb-change-layout' and always come back to his previous layout via *`[C-u]`* `ecb-toggle-layout`.

`toggle-scroll-other-window-scrolls-compile` **&optional** *ARG* [Command]
Toggle the state of `ecb-scroll-other-window-scrolls-compile-window`. With prefix argument *ARG*, set it to `t`, otherwise to `nil`. For all details about the scroll-behavior of `scroll-other-window` see the advice documentation of `other-window-for-scrolling`.

`toggle-window-sync` **&optional** *arg* [Command]
Toggle auto synchronizing of the ECB-windows. With prefix argument *ARG*, switch on if positive, otherwise switch off. If the effect is that auto-synchronizing is switched off then the current value of the option `ecb-window-sync` is saved so it can be used for the next switch on by this command. See also the option `ecb-window-sync`.

`update-directories-buffer` [Command]
Updates the ECB directories buffer.

`upgrade-options` [Command]
Check for all ECB-options if their current value is compatible to the defined type. If not upgrade it to the new type or reset it to the default-value of current ECB. Try also to upgrade renamed options. Displays all upgraded or reset options with their old (before the upgrade/reset) and new values.

`window-sync` [Command]
Synchronizes all special ECB-buffers with current buffer.

Depending on the contents of current buffer this command performs different synchronizing tasks but only if ECB is active and point stays in an edit-window.

- If current buffer is a file-buffer then all special ECB-tree-buffers are synchronized with current buffer.
- If current buffer is a dired-buffer then the directory- and the sources-tree-buffer are synchronized if visible

In addition to this the hooks in `ecb-current-buffer-sync-hook` run.

Most of these functions are also available via the menu "ECB" and also via the ECB key-map with prefix *`C-c .`* (see `ecb-minor-mode` for a complete list of the keybindings).

7 Customizing ECB

This chapter describes how to customize ECB for your personal taste. The first section introduces some general aspects (which you should really know!), the second one gives an overview of the most important options and the third one lists all options of ECB (divided into the customize groups).

7.1 General aspects for customizing ECB

This chapter contains all important informations you should know about customizing ECB. The first section gives an answer to the question "`setq` or `customize`" and the second section describes what to do when you have to customize ECB for a lot of people.

7.1.1 Setq or customize - what should i use?

The best way to customize all the options of ECB is via the customize-feature of (X)Emacs, i.e. means calling the commands `customize-option` or `customize-group` etc. This is also the strongly recommended way!

But of course you can also use `setq` or some Elisp-code to change the values of many but not all of the options. The values of the following options **MUST NOT** be changed via `setq` or Elisp-code but only with the customize-feature!

- `ecb-bucket-node-display`
- `ecb-compile-window-height`
- `ecb-compile-window-temporally-enlarge`
- `ecb-compile-window-width`
- `ecb-exclude-parents-regexp`
- `ecb-fix-window-size`
- `ecb-font-lock-tags`
- `ecb-highlight-tag-with-point-delay`
- `ecb-key-map`
- `ecb-layout-name`
- `ecb-layout-window-sizes`
- `ecb-mode-line-data`
- `ecb-mode-line-display-window-number`
- `ecb-mode-line-prefixes`
- `ecb-show-node-info-in-minibuffer`
- `ecb-show-tags`
- `ecb-source-path`
- `ecb-toggle-layout-sequence`
- `ecb-tag-display-function`
- `ecb-tree-do-not-leave-window-after-select`
- `ecb-type-tag-display`
- `ecb-type-tag-expansion`

- `ecb-use-speedbar-instead-native-tree-buffer`
- `ecb-window-sync-delay`
- `ecb-windows-height`
- `ecb-windows-width`

7.1.2 Site-wide customizing of ECB

If you are the administrator for an Emacs-site, means you are responsible for the basic customization of a lot of Emacs users, then you maybe need a way to customize Emacs and ECB without changing everyones '`.emacs`'-file and normally you will do this with the file '`site-start.el`'. You can customize all options of ECB in a central '`site-start.el`' (even the options mentioned above!) but you **MUST NOT** do this via `setq` but you have to use a mechanism like the following[1]!

This section describes two methods how to pre-customize ECB site-wide. The elisp-code contained in the following two subsections has to be copied to the file '`site-start.el`' before it can be used.

But ensure for both methods that you customize the options with the correct lisp format. Read carefully the docstrings of the options you want to customize from within Elisp-code!

7.1.2.1 Storing all option-settings in the users custom-file

The mechanism described here defines all site-wide-settings in a file '`site-lisp.el`' but stores the values in the users `custom-file` which is probably '`.emacs`'!

First two helper functions are needed, namely `customize-option-get-value` and `customize-save-variable-save` whereas the latter one sets the value for an option via the customize-mechanism (and is therefore allowed for the setq-forbidden options!) but only if the option has no saved value until now (i.e. the user has not saved this option for future sessions until now)

[1] At least for the options for which `setq` is explicitly forbidden, but it is recommended to use always such a mechanism

```
(defun customize-option-get-value (option type)
  "Return the value of a customizable option OPTION with TYPE, where TYPE
can either be 'standard-value \(the default-value of the defcustom) or
'saved-value \(the value stored persistent by the user via customize)."
  (let ((val (car (get option type))))
    (cond ((not (listp val)) val)
          ((equal 'quote (car val)) (car (cdr val)))
          (t (car val)))))

(defun customize-save-variable-save (option value &optional override)
  "Calls `customize-save-variable' with OPTION and VALUE if OPTION is a
custom-type and if OPTION has no saved-value until now.
If OVERRIDE is a function or lambda-form then it is called with two arguments:
- OLD-SAVED-VAL: The saved value of OPTION
- NEW-VALUE: see argument VALUE.
OVERRIDE is only called if OPTION has already a saved-value. If OVERIDE
returns not nil then `customize-save-variable' is called for OPTION with VALUE
even if OPTION has no saved-value until now."
  (and (get option 'custom-type)
       (or (not (get option 'saved-value))
           (and (functionp override)
                (funcall override
                         (customize-option-get-value option 'saved-value)
                         value)))
       (progn
         (message "Overriding saved value for option %s with %s" option value)
         (customize-save-variable option value))))
```

With `customize-save-variable-save` all ECB-options can be site-wide pre-customized like follows:

```
(customize-save-variable-save 'ecb-show-tags
                              '((include collapsed nil)
                                (parent collapsed nil)
                                (type flattened nil)
                                (variable collapsed name)
                                (function flattened name)
                                (rule flattened name)
                                (section flattened nil)
                                (def collapsed name)
                                (t collapsed name)))
(customize-save-variable-save 'ecb-font-lock-tags t)
;; add here more options of ECB it you want
```

7.1.2.2 Using a special setq for site-wide settings

The mechanism above saves the pre-customized values always in the users `custom-file` (probably '`.emacs`'). If this is not preferred, then you can use the following mechanism but

of course the offered `setq-save` is only allowed for options which are not setq-forbidden (see ⟨undefined⟩ [setq or customize], page ⟨undefined⟩).

The mechanism below does not change the users `custom-file`. This mechanism is needed especially if ECB should be autoloaded and all site-wide settings should first loaded when ECB is activated by the user. This can be achieved for example via[2]:

```
(require 'ecb-autoloads))
(eval-after-load "ecb"
  '(require 'site-ecb))
```

In such a situation the whole `custom-file` of a user is mostly loaded **before** ECB is activated and therefore before the site-wide-settings are loaded. So the users own customizations are loaded before the site-wide ones.

The `setq-save`-mechanism described below prevents the users own customisations contained in his `custom-file` from being overridden by the site-wide setq-settings. If `setq` would be used for the site-wide settings then in an autoload-situation the site-wide settings would override the users-settings and this should not be done!

First two helper-macros are needed:

```
(defmacro custom-saved-p (option)
  "Return only not nil if OPTION is a defcustom-option and has a
saved value. Option is a variable and is literal \(not evaluated)."
  `(and (get (quote ,option) 'custom-type)
        (get (quote ,option) 'saved-value)))

(defmacro setq-save (option value)
  "Sets OPTION to VALUE if and only if OPTION is not already saved
by customize. Option is a variable and is literal \(not evaluated)."
  `(and (not (custom-saved-p ,option))
        (set (quote ,option) ,value)))
```

With `setq-save` all "not-setq-forbidden"-ECB-options can be site-wide pre-customized like follows:

```
(setq-save ecb-tree-indent 4)
(setq-save ecb-tree-expand-symbol-before t)
(setq-save ecb-primary-secondary-mouse-buttons 'mouse-1--mouse-2)
```

7.2 The most important options of ECB

Here are the most important options (it is recommended to check at least the following options before working with ECB). You can customize them via the customize-group "ecb-most-important" or via the command `ecb-customize-most-important`.

`ecb-source-path`
: Where ECB can find your sources. You must set this option!

`ecb-show-help-format`
: Should the online help of ECB be displayed in the standard Info format or in HTML format in a web-browser.

[2] The file '`site-ecb.el`' contains all site-wide settings for ECB

`ecb-auto-activate`
`ecb-major-modes-show-or-hide`
: Auto. activation of ECB after start (see ⟨undefined⟩ [Automatic activation], page ⟨undefined⟩) or major-mode-based showing or hiding the ecb-windows.

`ecb-winman-escreen-number`
`ecb-winman-winring-name`
: Support of several window-managers (see ⟨undefined⟩ [Window-managers and ECB], page ⟨undefined⟩).

`ecb-key-map`
: All ECB-keybindings incl. a common prefix-key (see ⟨undefined⟩ [Using the keyboard], page ⟨undefined⟩).

`ecb-new-ecb-frame`
: Should ECB create a new frame at activation time.

`ecb-primary-secondary-mouse-buttons`
`ecb-mouse-click-destination`
: Define how to use the mouse (see ⟨undefined⟩ [Using the mouse], page ⟨undefined⟩).

`ecb-tree-buffer-style`
`ecb-tree-expand-symbol-before`
`ecb-tree-indent`
`ecb-truncate-lines`
: The look&feel of the trees in the tree-buffers. The former option defines the general style of the tree-buffers and the latter ones allow to customize the ascii-style tree-buffers (maybe you like a value of 4 for the latter one if you display the expand-symbol before (see ⟨undefined⟩ [Tree-buffer styles], page ⟨undefined⟩).

`ecb-source-file-regexps`
: Which files will (not) be shown in ECB.

`ecb-show-node-info-in-minibuffer`
: When and which node-info should be displayed in the minibuffer?

`ecb-layout-name`
`ecb-compile-window-height`
`ecb-compile-window-width`
`ecb-other-window-behavior`
: The ECB layout, means which windows you want to be displayed in the ECB-frame and also the location of these windows (see ⟨undefined⟩ [Changing the ECB-layout], page ⟨undefined⟩).

`ecb-compilation-buffer-names`
: Which buffers should be treaten as “compilation-buffers” and therefore displayed in the compile-window of ECB - if there is any.

`ecb-tag-display-function`
`ecb-type-tag-display`
`ecb-type-tag-expansion`
`ecb-show-tags`
: How to display the entries in the ECB-method window for semantic supported sources (see ⟨undefined⟩ [Customizing the display], page ⟨undefined⟩). These options take only effect for semantic-sources (see ⟨undefined⟩ [Definition of semantic- and non-semantic-sources], page ⟨undefined⟩).

`ecb-process-non-semantic-files`
: Displaying file-contents for not by semantic supported files too, e.g. for LaTeX- and perl-sources (see ⟨undefined⟩ [Non-semantic sources], page ⟨undefined⟩).

But to make ECB working best for you it is also recommended to have a look at ⟨undefined⟩ [Customizable options], page ⟨undefined⟩!

7.3 All customizable options of ECB

All customization of ECB is divided into the following "customize groups". You can highly customize all the ECB behavior/layout so just go to these groups and you will see all well documented ECB-options.

Please note: All options in the following subsections are listed without the prefix "ecb-" (e.g. the option `ecb-layout-name` is listed with name "layout-name"). This has been done for a better readable option index. See ⟨undefined⟩ [Option Index], page ⟨undefined⟩.

7.3.1 Group ecb-general

This group contains general settings for the Emacs code browser:

`activate-before-layout-draw-hook` [User Option]
: Normal hook run at the end of activating the ecb-package by running `ecb-activate`. This hooks are run after all the internal setup process but directly before(!) drawing the layout specified in `ecb-layout` (means before dividing the frame into several windows).

  A senseful using of this hook can be maximizing the Emacs-frame for example, because this should be done before the layout is drawn because ECB computes the size of the ECB-windows with the current frame size! If you need a hook-option for the real end of the activating process (i.e. after the layout-drawing) look at `ecb-activate-hook`.

  IMPORTANT: The difference between this hook and `ecb-redraw-layout-before-hook` is that the latter one is evaluated always before the layout is redrawn (for example after calling `ecb-redraw-layout`) whereas the former one (this hook) is only evaluated exactly once during the activation-process of ECB. So during the activation process there is the following sequence of hooks:

  1. `ecb-activate-before-layout-draw-hook` \(this one)
  2. `ecb-redraw-layout-before-hook`
  3. <Drawing the layout>
  4. `ecb-redraw-layout-after-hook`
  5. `ecb-activate-hook`

`activate-hook` [User Option]
Hook run at the end of activating ECB by `ecb-activate`. This hooks are run at the real end of the activating process, means after the layout has been drawn!. If you need hooks which are run direct before the layout-drawing look at `ecb-activate-before-layout-draw-hook`.

`activation-selects-ecb-frame-if-already-active` [User Option]
Trying to activate an already activated ECB selects the ECB-frame. If t then the ECB-frame is selected, if nil then it is not. If 'ask then ECB asks if the ECB-frame should be selected if the current-frame is not the `ecb-frame`.

`auto-activate` [User Option]
Automatically startup ECB when Emacs starts up. This should only be true if you always want to run `ecb-activate`.

`auto-compatibility-check` [User Option]
Check at ECB-startup if all ECB-options have correct values. If not nil then all ECB-options are checked if their current value have the correct type. It the type is incorrect the option is either auto. upgraded to the new type or reset to the default-value of current ECB if no upgrade is possible. This feature can also upgrade options which are renamed in current ECB and try to transform the old-value to the new named option. After startup all upgraded or reset options are displayed with their old (before upgrade/reset) and new values. See also the commands `ecb-upgrade-options` and `ecb-display-upgraded-options`. If this option is off then the user can perform the check and reset manually with `ecb-upgrade-options`. See ⟨undefined⟩ [Auto. option-upgrading], page ⟨undefined⟩.

`before-activate-hook` [User Option]
Normal hook run at the beginning of activating the ecb-package by running `ecb-activate`. These hooks run before any other tasks of the activating process are performed. If any of these hooks returns nil then ECB will not be activated!

This can be used to check some conditions and then only start ECB if all conditions are true. For example a function could be added which returns only nil if Gnus is running. Then calling `ecb-activate` or `ecb-minor-mode` will only start ECB if Gnus is not already running.

`before-deactivate-hook` [User Option]
Normal hook run at the beginning of deactivating ECB by running `ecb-deactivate`. These hooks run before any other tasks of the deactivating process are performed. If any of these hooks returns nil then ECB will not be deactivated! See also `ecb-before-activate-hook`.

`bucket-node-display` [User Option]
How ECB displays bucket-nodes in a ECB tree-buffer. Bucket-nodes have only one job: Nodes with similar properties will be dropped into one bucket for such a common property and all these nodes will be added as children to the bucket-node. Besides being expandable and collapsable a bucket-node has no senseful action assigned. Examples for bucket-nodes are "[+] Variables, "[+] Dependencies" etc. in the Methods-buffer or buckets which combine filenames with same extension under a bucket-node with name this extension.

This option defines how bucket-node should be displayed. The name of the bucket-node is computed by ECB but you can define a prefix, a suffix and a special face for the bucket-node

The default are empty prefix/suffix-strings and `ecb-bucket-node-face`. But an alternative can be for example '("[" "]" nil) which means no special face and a display like "[+] [<bucket-name>]".

`clear-caches-before-activate` [User Option]

Clear all ECB internal caches before startup. If t then ECB clears all its internal caches before starting up. Caches are used for files- and subdirs (see `ecb-cache-directory-contents` and `ecb-cache-directory-contents-not`) for semantic-tags and for the history-filter.

This caches are completely clean at load-time of the ECB-library!

Default is nil, because is makes sense not to clear these caches at start-time because ECB is often deacticated temporally especially in combination with window-managers like escreen.el. In these situations the internal state of ECB should be preserved for next activation.

`current-buffer-sync-hook` [User Option]

Normal hook run at the end of `ecb-current-buffer-sync`.

See documentation of `ecb-current-buffer-sync` for conditions when synchronization takes place and so in turn these hooks are evaluated.

Precondition for such a hook: Current buffer is the buffer of the current selected edit-window.

Postcondition for such a hook: Point must stay in the same edit-window as before evaluating the hook.

Important note: If `ecb-window-sync` is not nil `ecb-current-buffer-sync` is running either every time Emacs is idle or even after every command (see `ecb-window-sync-delay`). So these hooks can be really called very often! Therefore each function of this hook should/must check in an efficient way at beginning if its task have to be really performed and then do them only if really necessary! Otherwise performance of Emacs could slow down dramatically!

It is strongly recommended that each function added to this hook uses the macro `ecb-do-if-buffer-visible-in-ecb-frame` at beginning! See `ecb-speedbar-current-buffer-sync` and `ecb-eshell-current-buffer-sync` for examples how to use this macro!

`deactivate-hook` [User Option]

Normal hook run at the end of deactivating (but before the ecb-layout is cleared!) ECB by running `ecb-deactivate`.

`debug-mode` [User Option]

If not nil ECB displays debug-information in the Messages-buffer. This is done for some critical situations concerning semantic-tags and their overlays (or extends for XEmacs). Normally you should not need this switched on! But if you get errors like "destroyed extend" for XEmacs or "wrong-argument-type" concerning overlays for

GNU Emacs then you should switch on this option and submitting a bug-report to the ecb-mailing-list (`ecb-submit-problem-report`) after getting the error again!

`grep-function` [User Option]

Function used for performing a grep. The popup-menu of the tree-buffers "Directories", "Sources" and "History" offer to grep the "current" directory:

- Directory-buffer: The grep is performed in the current popup-directory after clicking the right mouse-button onto a node.
- Sources-buffer: The grep is performed in the current selected directory.
- History-buffer: The grep is performed in the directory of the current popup-source after clicking the right mouse-button onto a node.

`grep-recursive-function` [User Option]

Function used for performing a recursive grep. For more Details see option 'ecb-grep-function' and replace "grep" with "recursive grep".

`key-map` [User Option]

Specifies all keybindings for the ECB minor-mode key-map. The value is a cons-cell where the car is a common-prefix key for all the keybindings. The cdr is a list of keybindings each of them a list again. A key-binding has the following form:

```
'(<common-prefix-flag> <keysequence> <function>) where
```

`<common-prefix-flag>`
: If t then the common-prefix-key defined as car of the value (see above) is used.

`<keysequence>`
: If the common prefix-key is used then the final key-binding is the concatenation of the common-prefix-key (see above) and this keysequence.

`<function>:`
: The function to bind to the key. This can also be a lambda-expression .

It is highly recommended to use one of the standard keys C-c or C-x as first key of your common-prefix-key!

You MUST change this option via customize to take effect!

All keysequences must be inserted as a string and must follow the syntax needed by `read-kbd-macro` or `kbd`. This means you can insert the key in the same manner *`C-h k`* displays keysequences. Here is the summary of the syntax:

Text is divided into "words" separated by whitespace. Except for the words described below, the characters of each word go directly as characters of the keysequence. The whitespace that separates words is ignored. Whitespace in the macro must be written explicitly, as in *`C-c SPC`*.

- The special words RET, SPC, TAB, DEL, LFD, ESC, and NUL represent special control characters. The words must be written in uppercase.
- A word in angle brackets, e.g., <return>, <down>, <left> or <f1>, represents a function key. (Note that in the standard configuration, the function key <return> and the control key RET are synonymous.). You can use angle brackets on the words RET, SPC, etc., but they are not required there.

- Keys can be written by their ASCII code, using a backslash followed by up to six octal digits. This is the only way to represent keys with codes above .
- One or more prefixes M- (meta), C- (control), S- (shift), A- (alt), H- (hyper), and s- (super) may precede a character or key notation. For function keys, the prefixes may go inside or outside of the brackets: C-<down> = <C-down>. The prefixes may be written in any order: M-C-x = C-M-x. Prefixes are not allowed on multi-key words, e.g., C-abc, except that the Meta prefix is allowed on a sequence of digits and optional minus sign: M–123 = M– M-1 M-2 M-3.
- The ^ notation for control characters also works: ^M = C-m.

`major-modes-show-or-hide` [User Option]

List of major-modes which show or hide the ecb-windows. The value is a cons-cell where the car contains all major-mode-symbols which should show the special ecb-windows and the cdr contains all major-mode-symbols which should hide the special ecb-windows. If the symbol of a major-mode is neither contained in the car-"show-list" nor in the cdr-"hide-list" then the visibility-state of the ecb-windows does not change.

`minor-mode-text` [User Option]

String to display in the mode line when ECB minor mode is active. (When the string is not empty, make sure that it has a leading space.)

Because for ECB it is quite obvious if it is active or not when the ECB-windows are visible this text is only display in the modeline if the ECB-windows are hidden.

`mouse-click-destination` [User Option]

Destination of a mouse-button click. Defines in which edit-window (if splitted) ECB does the "right" action (opening a source, jumping to a method/variable etc.) after clicking with the primary mouse-button (see `ecb-primary-secondary-mouse-buttons`) onto a node. There are two possible choices:

- `left-top`: Does the "right" action always in the left/topmost edit-window.
- `last-point`: Does the "right" action always in that edit-window which had the point before.

This is if the user has clicked either with the primary mouse-button or has activated a popup-menu in the tree-buffer.

If the edit-area is not splitted this setting doesn't matter.

A click with the secondary mouse-button (see again `ecb-primary-secondary-mouse-buttons` does the "right" action always in another edit-window related to the setting in this option: If there are two edit-windows then the "other" edit-window is used and for more than 2 edit-windows the "next" edit-window is used (whereas the next edit-window of the last edit-window is the first edit-window).

Note: If the tree-buffers are used with the keyboard instead with the mouse then this option takes effect too because *`RET`* is interpreted as primary mouse-button and *`C-RET`* as secondary mouse-button!

`run-ediff-in-ecb-frame` [User Option]
Run ediff-sessions in the same frame as ECB is running. If not nil then ECB ensures that ediff runs in the same frame as ECB and ECB restores exactly the "before-ediff"-window-layout after quiting ediff. If nil then ediff decides in which frame it will run - depending on the current window-layout (e.g. if the ecb-windows are currently hidden) this can be the ecb-frame but this can also be a newly created frame or any other frame.

`stealthy-tasks-delay` [User Option]
Time Emacs must be idle before ECB runs its stealthy tasks. Currently ECB performes the following stealthy tasks:

Prescann directories for emptyness
Prescann directories and display them as empty or not-empty in the directories-buffer. See the documentation of the option `ecb-prescan-directories-for-emptyness` for a description.

File is read only
Check if sourcefile-items of the directories- or sources-buffer are read-only or not. See documentation of the option `ecb-sources-perform-read-only-check`.

Version-control-state
Checks the version-control-state of files in directories which are managed by a VC-backend. See the option `ecb-vc-enable-support`.

Here the interval is defined ECB has to be idle before starting with these stealthy tasks. It can be a floating-point value in seconds. The value can also be changed during running ECB.

`tip-of-the-day` [User Option]
Show tip of the day at start time of ECB.

`tip-of-the-day-file` [User Option]
File where tip-of-the-day cursor is stored.

`use-recursive-edit` [User Option]
Tell ECB to use a recursive edit. If set then it can easily be deactivated by (keyboard-escape-quit).

`version-check` [User Option]
Checks at start-time if the requirements are fulfilled. It checks if the required versions of the libraries semantic, eieio and speedbar are installed and loaded into Emacs.

It is strongly recommended to set this option to not `nil`!

`window-sync` [User Option]
Synchronize the ECB-windows automatically with current edit window. If `always` then the synchronization takes place always a buffer changes in the edit window, if `nil` then never. If a list of major-modes then only if the `major-mode` of the new buffer belongs NOT to this list.

But in every case the synchronization only takes place if the current-buffer in the current active edit-window has a relation to files or directories. Examples for the former one are all programming-language-modes, `Info-mode` too, an example for the latter one is `dired-mode`. For all major-modes related to non-file/directory-buffers like `help-mode`, `customize-mode` and others never an autom. synchronization will be done!

It's recommended to exclude at least `Info-mode` because it makes no sense to synchronize the ECB-windows after calling the Info help. Per default also `dired-mode` is excluded but it can also making sense to synchronize the ECB-directories/sources windows with the current directory in the dired-buffer.

IMPORTANT NOTE: Every time the synchronization is done the hook `ecb-current-buffer-sync-hook` is evaluated.

`window-sync-delay` [User Option]

Time Emacs must be idle before the ECB-windows are synchronized with current edit window. If nil then there is no delay, means synchronization takes place immediately. A small value of about 0.25 seconds saves CPU resources and you get even though almost the same effect as if you set no delay.

7.3.2 Group ecb-tree-buffer

This group contains general settings related to the tree-buffers of ECB:

`common-tree-buffer-after-create-hook` [User Option]

Local hook running at the end of each tree-buffer creation. Every function of this hook is called once without arguments direct after creating a tree-buffer of ECB and it's local key-map. So for example a function could be added which performs calls of `local-set-key` to define new keybindings for EVERY tree-buffer.

The following keys must not be rebind in all tree-buffers:

- *`RET`* and all combinations with *`Shift`* and *`Ctrl`*
- *`TAB`*
- `C-t`

`primary-secondary-mouse-buttons` [User Option]

Primary- and secondary mouse button for using the ECB-buffers. A click with the primary button causes the main effect in each ECB-buffer:

- ECB Directories: Expanding/collapsing nodes and displaying files in the ECB Sources buffer.
- ECB sources/history: Opening the file in that edit-window specified by the option `ecb-mouse-click-destination`.
- ECB Methods: Jumping to the method in that edit-window specified by the option `ecb-mouse-click-destination`.

A click with the primary mouse-button while the SHIFT-key is pressed called the POWER-click and does the following (depending on the ECB-buffer where the POWER-click occurs):

- ECB Directories: Refreshing the directory-contents-cache (see `ecb-cache-directory-contents`).

- ECB sources/history: Only displaying the source-contents in the method-buffer but not displaying the source-file in the edit-window.
- ECB Methods: Narrowing to the clicked method/variable/ect... (see `ecb-tag-visit-post-actions`). This works only for semantic supported sources but not for imenu- or etags-supported ones!

In addition always the whole node-name is displayed in the minibuffer after a POWER-click \(for this see also 'ecb-show-node-info-in-minibuffer').

The secondary mouse-button is for opening (jumping to) the file in another edit-window (see the documentation `ecb-mouse-click-destination`).

The following combinations are possible:

– primary: mouse-2, secondary: C-mouse-2 (means mouse-2 while CTRL-key is pressed). This is the default setting.

– primary: mouse-1, secondary: C-mouse-1

– primary: mouse-1, secondary: mouse-2

Please note: If the tree-buffers are used with the keyboard instead with the mouse then *RET* is interpreted as primary mouse-button and *C-RET* as secondary mouse-button!

If you change this during ECB is activated you must deactivate and activate ECB again to take effect

`tree-buffer-style` [User Option]

The style of the tree-buffers. There are three different styles available:

Image-style (value `image`): Very nice and modern - just try it. For this style the options `ecb-tree-indent` and `ecb-tree-expand-symbol-before` have no effect! Note: GNU Emacs <= 21.3.X for Windows does not support image-display so ECB uses always 'ascii-guides even when here 'image is set!

Ascii-style with guide-lines (value `ascii-guides`):

```
[-] ECB
 |  [+] code-save
 `- [-] ecb-images
     |  [-] directories
     |   |  [-] height-15
     |   |   |  * close.xpm
     |   |   |  * empty.xpm
     |   |   |  * leaf.xpm
     |   |   `- * open.xpm
     |   |  [+] height-17
     |   |  [+] height-19
     |   `- [+] height-21
     |  [x] history
     |  [x] methods
     `- [x] sources
```

Ascii-style without guide-lines (value `ascii-no-guides`) - this is the style used by ECB <= 1.96:

```
[-] ECB
    [+] code-save
    [-] ecb-images
        [-] directories
            [-] height-15
                * close.xpm
                * empty.xpm
                * leaf.xpm
                * open.xpm
            [+] height-17
            [+] height-19
            [+] height-21
        [x] history
        [x] methods
        [x] sources
```

With both ascii-styles the tree-layout can be affected with the options `ecb-tree-indent` and `ecb-tree-expand-symbol-before`.

`tree-do-not-leave-window-after-select` [User Option]

Tree-buffers which stay selected after a key- or mouse-selection. If a buffer (either its name or the variable-symbol which holds the name) is contained in this list then selecting a tree-node either by RET or by a mouse-click doesn't leave that tree-buffer after the node-selection but performes only the appropriate action (opening a new source, selecting a method etc.) but point stays in the tree-buffer. In tree-buffers not contained in this option normaly a node-selection selects as "last" action the right edit-window or maximizes the next senseful tree-buffer in case of a currently maximized tree-buffer (see `ecb-maximize-next-after-maximized-select`).

The buffer-name can either be defined as plain string or with a symbol which contains the buffer-name as value. The latter one is recommended for the builtin ECB-tree-buffers because then simply the related option-symbol can be used.

A special remark for the `ecb-directories-buffer-name`: Of course here the value of this option is only relevant if the name of the current layout is contained in `ecb-show-sources-in-directories-buffer` or if the value of `ecb-show-sources-in-directories-buffer` is 'always and the clicked ot hitted node represents a sourcefile (otherwise this would not make any sense)!

The setting in this option is only the default for each tree-buffer. With the command `ecb-toggle-do-not-leave-window-after-select` the behavior of a node-selection can be changed fast and easy in a tree-buffer without customizing this option, but of course not for future Emacs sessions!

`tree-easy-hor-scroll` [User Option]

Scroll step for easy hor. scrolling via mouse-click in tree-buffers. XEmacs has horizontal scroll-bars so invisible parts beyond the right window-border of a tree-buffer can always made visible very easy.

GNU Emacs does not have hor. scroll-bars so especially with the mouse it is quite impossible to scroll smoothly right and left. The functions `scroll-left` and `scroll-right` can be annoying and are also not bound to mouse-buttons.

If this option is a positive integer S then in all ECB-tree-buffers the keys *M-mouse-1* and M-mouse-3 are bound to scrolling left rsp. right with scroll-step S - clicking with *mouse-1* or *mouse-2* onto the edge of the modeline has the same effect, i.e. if you click with mouse-1 onto the left (rsp right) edge of the modeline you will scroll left (rsp. right).

Additionally C-M-mouse-1 and C-M-mouse-3 are bound to scrolling left rsp. right with scroll-step window-width - 2.

Default is a scroll-step of 5. If the value is nil then no keys for horizontal scrolling are bound.

tree-expand-symbol-before [User Option]

Show the expand symbol before the items in a tree. When the expand-symbol is located before the items then the tree looks like:

```
[-] ECB
    [+] code-save
    [-] ecb-images
        [-] directories
```

When located after then the tree looks like:

```
ECB [-]
  code-save [+]
  ecb-images [-]
    directories [-]
```

The after-example above use a value of 2 for ecb-tree-indent whereas the before-example uses a value of 4.

It is recommended to display the expand-symbol before because otherwise it could be that with a deep nested item-structure with and/or with long item-names (e.g. a deep directory-structure with some long subdirectory-names) the expand-symbol is not visible in the tree-buffer and the tree-buffer has to be horizontal scrolled to expand an item.

tree-image-icons-directories [User Option]

Directories where the images for the tree-buffer can be found. This is a cons cell where:

car: Default directory where the default images for the tree-buffer can be found. It should contain an image for every name of tree-buffer-tree-image-names. The name of an image-file must be: "ecb-<NAME of TREE-BUFFER-TREE-IMAGE-NAMES>.<ALLOWED EXTENSIONS>".

cdr: This is a list where each element is a cons again with: car is the buffer name of the tree-buffer for which a special image-path should be used. The buffer-name can either be defined as plain string or with a symbol which contains the buffer-name as value. The latter one is recommended for the builtin ECB-tree-buffers because then simply the related option-symbol can be used (e.g. the symbol ecb-directories-buffer-name). The cdr is the the full-path of an additional image-directorie which is searched first for images needed for the related tree-buffer. If the image can not be found in this directory then the default-directory (see above) is searched. If the

image can't even be found there the related ascii-symbol is used - which is defined in `tree-buffer-tree-image-names`. If a tree-buffer is not contained in this list then there is no additional special image-directory for it.

ECB comes with predefined images in several different heights - so for the most senseful font-heights of a tree-buffer a fitting image-size should be available. The images reside either in the subdirectory "ecb-images" of the ECB-installation or - if ECB is installed as regular XEmacs-package - in the ECB-etc data-directory (the directory returned by (locate-data-directory "ecb").

`tree-incremental-search` [User Option]

Enable incremental search in the ECB-tree-buffers. For a detailed explanation see the online help section "Working with the keyboard in the ECB buffers". If you change this during ECB is activated you must deactivate and activate ECB again to take effect.

`tree-indent` [User Option]

Indent size for tree buffer. If you change this during ECB is activated you must deactivate and activate ECB again to take effect.

`tree-mouse-action-trigger` [User Option]

When the tree-buffer mouse-action should be triggered. This option determines the moment a mouse-action in a tree-buffer is triggered. This can be either direct after pressing a mouse-button (value `button-press`) or not until releasing the mouse-button (value: `button-release`).

If you change this during ECB is activated you must deactivate and activate ECB again to take effect!

`tree-navigation-by-arrow` [User Option]

Enable smart navigation in the tree-windows by horiz. arrow-keys. If not nil then the left- and right-arrow keys work in the ECB tree-window in the following smart way if onto an expandable node:

- Left-arrow: If node is expanded then it will be collapsed otherwise point jumps to the next "higher" node in the hierarchical tree (higher means the next higher tree-level or - if no higher level available - the next higher node on the same level).
- Right-arrow: If node is not expanded then it will be expanded. Onto a not expandable node the horizontal arrow-keys go one character in the senseful correct direction.

If this option is changed the new value takes first effect after deactivating ECB and then activating it again!

`tree-truncate-lines` [User Option]

Truncate lines in ECB buffers. If a buffer (either its name or the variable-symbol which holds the name) is contained in this list then line-truncation is switched on for this buffer otherwise it is off.

The buffer-name can either be defined as plain string or with a symbol which contains the buffer-name as value. The latter one is recommended to switch on line-truncation

for one of the builtin ECB-tree-buffers because then simply the related option-symbol can be used. To truncate lines in the builtin directories tree-buffer just add the symbol `ecb-directories-buffer-name` to this option.

If you change this during ECB is activated you must deactivate and activate ECB again to take effect.

`truncate-long-names` [User Option]

Truncate long names that don't fit in the width of the ECB windows. If you change this during ECB is activated you must deactivate and activate ECB again to take effect.

7.3.3 Group ecb-directories

This group contains settings for the directories-buffer in the ECB:

`add-path-for-not-matching-files` [User Option]

Add path of a file to `ecb-source-path` if not already contained. This is done during the auto. windows synchronization which happens if a file is opened not via the file/directory-browser of ECB. In such a situation ECB adds the path of the new file auto. to `ecb-source-path` at least temporally for the current Emacs session. This option defines two things:

1. Should only the root-part (which means for Unix-like systems always '/' and for windows-like systems the drive) of the new file be added as source-path to `ecb-source-path` or the whole directory-part? For remote-files (e.g. tramp, ange-ftp- or efs-files) the root-part is the complete host-part + the root-dir at that host (example: /berndl@ecb.sourceforge.net:/ would be the root-part of /berndl@ecb.sourceforge.net:/tmp/test.txt).
2. Should this path be added for future sessions too?

The value of this option is a cons-cell where the car is a boolean for 1. and the cdr is a boolean for 2.

A value of not nil for the car (1.) is reasonably if a user often opens files not via the ECB-browser which are not located in any of the paths of `ecb-source-path` because then only one path for each drive (windows) or the root-path (Unix) is added to the directory buffer of ECB.

`auto-expand-directory-tree` [User Option]

Automatically expand the directory tree to the current source file. There are three options:

- `best`: Expand the best-matching source-path
- `first`: Expand the first matching source-path
- `nil`: Do not automatically expand the directory tree.

`after-directory-change-hook` [User Option]

Hook which run directly after the selected directory has changed. This means not onyl after a click onto a directory in the directory-window of ECB but it means this hook runs always when the current directory changes regardless of the trigger of this change. So for example it runs also when you just switches from one buffer to another via

`switch-to-buffer` or `switch-to-buffer-other-window` and the directory of these filebuffers is different but only when auto-synchronizing of the ECB-windows is on (see `ecb-window-sync`). It runs not when switching between buffers and the associated files reside in the same directory.

Each function added to this hook will be called with two arguments: The directory which was current _before_ the directory-change-trigger and the directory which was now the current (i.e. after the trigger).

Example: If you switch from a filebuffer "~/.emacs" to a filebuffer "/tmp/test.txt" then the functions of this hook will be called with the two arguments "~" and "/tmp".

`cache-directory-contents` [User Option]

Cache contents of directories.

This can be useful if `ecb-source-path` contains directories with many files and subdirs, especially if these directories are mounted net-drives ("many" means here something > 500, dependent of the speed of the net-connection and the machine). Or if it contains remote-source-paths which means paths in the sense of tramp, ange-ftp or efs. For these directories actualizing the sources- and/or directories- buffer of ECB (if displayed in current layout!) can slow down dramatically so a caching increases speed a lot.

The value of this option is a list where each element is a cons-cell and looks like:

```
(<dir-regexp> . <filenumber threshold>) with
```

`<dir-regexp>`:
Regular expression a directory must match to be cached.

`<filenumber threshold>`:
Number of directory contents must exceed this number.

A directory will only be cached if and only if the directory-name matches at least one rexexp of this option and its content-number exceeds the related threshold AND the directory-name matches NOT any regexp of `ecb-cache-directory-contents-not`!

The cache entry for a certain directory will be refreshed and actualized only by using the POWER-click (see `ecb-primary-secondary-mouse-buttons`) in the directories-buffer of ECB (see ⟨undefined⟩ [Using the mouse], page ⟨undefined⟩).

Default-value: ECB caches the contents of all remote directories regardless of the size and all other directories if more than 50 entries are contained.

Examples:

An entry `("/usr/home/john_smith/bigdir*" . 1000)` means the contents of every subdirectory of the home-directory of John Smith will be cached if the directory contains more than 1000 entries and its name begins with "bigdir".

An entry `(".*" . 1000)` caches every directory which has more than 1000 entries.

An entry `("^/\\([^:/]*@\\)?\\([^@:/]*\\):.*" . 0)` caches every remote (in the sense of tramp, ange-ftp or efs) directory regardless of the number of entries."

Please note: If you want your home-dir being cached then you MUST NOT use "~" because ECB tries always to match full path-names!

`cache-directory-contents-not` [User Option]

Do not cache the contents of certain directories. The value of this option is a list where the each element is a regular expression a directory must match if it should not being cached.

If a directory-name matches at least one of the regexps of this option the directory-contents will never being cached. See `ecb-cache-directory-contents` to see when a directory will be cached.

This option can be useful when normally all directories with a certain amount of content (files and subdirs) should be cached but some special directories not. This can be achieved by:

- Setting `ecb-cache-directory-contents` to ((".*" . 500)): Caches all directories with more then 500 entries
- Setting `ecb-cache-directory-contents-not` to a value which matches these directories which should not being cached (e.g. ("/usr/home/john_smith") excludes the HOME-directory of John Smith from being cached).

Please note: If you want your home-dir exclude from being cached then you MUST NOT use "~" because ECB tries always to match full path-names!

`directories-buffer-after-create-hook` [User Option]

Local hook running after the creation of the directories-buffer. Every function of this hook is called once without arguments direct after creating the directories-buffer of ECB and it's local key-map. So for example a function could be added which performs calls of `local-set-key` to define new keybindings only for the directories-buffer of ECB.

The following keys must not be rebind in the directories-buffer: *`F2`*, *`F3`* and *`F4`*

`directories-buffer-name` [User Option]

Name of the ECB directory buffer. Because it is not a normal buffer for editing you should enclose the name with stars, e.g. " *ECB Directories*".

If it is necessary for you you can get emacs-lisp access to the buffer-object of the ECB-directory-buffer by this name, e.g. by a call of `set-buffer`.

Changes for this option at runtime will take affect only after deactivating and then activating ECB again!

`directories-menu-sorter` [User Option]

Function which re-sorts the menu-entries of the directories buffer.

If a function then this function is called to re-arrange the menu-entries of the combined menu-entries of the user-menu-extensions of `ecb-directories-menu-user-extension` and the built-in-menu `ecb-directories-menu`. If nil then no special sorting will be done and the user-extensions are placed in front of the built-in-entries.

The function get one argument, a list of menu-entries. For the format of this argument see `ecb-directories-menu-user-extension`. The function must return a new list in the same format. Of course this function can not only re-arrange the entries but also delete entries or add new entries.

`directories-menu-user-extension` [User Option]

Static user extensions for the popup-menu of the directories buffer. Value is a list of elements of the following type: Each element defines a new menu-entry and is either:

1. Menu-command: A list containing two sub-elements, whereas the first is the function (a function symbol) being called if the menu-entry is selected and the second is the name of the menu-entry.
2. Separator: A one-element-list and the element is the string "—": Then a non-selectable menu-separator is displayed.
3. Submenu: A list where the first element is the title of the submenu displayed in the main-menu and all other elements are either menu-commands (see 1) or separators (see 2) or another submenu (see c). This allows deep nested menu-submenu-structures. Currently a level of 4 is allowed but in general there could be an infinite depth of nesting but it makes no sense - if possible at all - to define infinite nested defcustom-types. So there is a limit of 4 levels but tis is not a hard limit: Just increase the value of the `ecb-max-submenu-depth` **BEFORE** first loading ECB!

The function of a menu-command must follow the following guidelines: Such a function must be defined with the macro `tree-buffer-defpopup-command`! This macro defines a new popup-command whereas the newly defined command gets one argument *NODE*. See the docstring of `tree-buffer-defpopup-command` for further details.

Example for the definition of such a menu-function:

```
(tree-buffer-defpopup-command ecb-my-special-dir-popup-function
  "Prints the name of the directory of the node under point."
  (let ((node-data=dir (tree-node-get-data node)))
     (message ``Dir under node: %s'' node-data=dir)))
```

Per default the static user-extensions are added at the beginning of the built-in menu-entries of `ecb-directories-menu` but the whole menu can be re-arranged with `ecb-directories-menu-sorter`.

These menu-extensions are static. A dynamic menu-extension can be achieved via `ecb-directories-menu-user-extension-function`.

`directories-menu-user-extension-function` [User Option]

Dynamic user extensions for the popup-menu of the directories buffer. A function which has to return a list in the same format like the option `ecb-directories-menu-user-extension`. This function is called when the user opens the popup-menu for the directories buffer.

If no dynamically evaluated menu-extensions should be added to the directories-buffer the function has to return nil. Therefore the default-value of this option is `ignore`.

Per default the dynamic user-extensions are added in front of the static extensions of `ecb-directories-menu-user-extension` but the whole menu can be re-arranged with `ecb-directories-menu-sorter`.

`directories-show-node-info` [User Option]

When to display which node-info in the directories-buffer. Define which node info should displayed after moving the mouse over a node (or after a shift click onto the node) in the directories-buffer.

You can define "when" a node-info should be displayed:

- always: Node info is displayed by moving with the mouse over a node.
- if-too-long: Node info is only displayed by moving with the mouse over a node does not fit into the window-width of the tree-buffer window. In the ECB directories buffer this means also if a node is shortend or if the node has an alias (see `ecb-source-path`).
- shift-click: Node info is only displayed after a shift click with the primary mouse button onto the node.
- never: Node info is never displayed.

You can define what info should be displayed:

- name: Only the full node-name is displayed.
- path: The full-path of the node is displayed.

Do NOT set this option directly via setq but use always customize!

`display-default-dir-after-start` [User Option]

Automatically display current default-directory after activating ECB.

If a file-buffer is displayed in the current active edit-window then ECB synchronizes its tree-buffers to this file-buffer - at least if the option `ecb-window-sync` it not nil. So for this situation `ecb-display-default-dir-after-start` takes no effect but this option is for the case if no file-buffer is displayed in the edit-window after startup:

If true then ECB selects autom. the current default-directory after activation even if no file-buffer is displayed in the current active edit-window. This is useful if ECB is autom. activated after startup of Emacs and Emacs is started without a file-argument. So the directory from which the startup has performed is auto. selected in the ECB-directories buffer and the ECB-sources buffer displays the contents of this directory.

`excluded-directories-regexps` [User Option]

Directories that should not be included in the directories list. The value of this variable should be a list of regular expression.

`prescan-directories-for-emptyness` [User Option]

Prescan directories for emptyness. ECB does this so directories are displayed as empty in the directories-buffer even without user-interaction (i.e. in previous ECB-versions the emptyness of a directory has been first checked when the user has clicked onto a directory). ECB optimizes this check as best as possible but if a directory contains a lot of subdirectories which contain in turn a lot of entries, then expanding such a directory or selecting it would take of course more time as without this check - at least at the first time (all following selects of a directory uses the cached information if its subdirectories are empty or not). Therefore ECB performs this check stealthy (see `ecb-stealthy-tasks-delay`) so normally there should no performance-decrease or

additional waiting-time for the user. There is one exception: For remote directories (in the sense of tramp, ange-ftp, or efs) this check can descrease performance even if performed stealthy and interruptable. Therefore this option offers three possible settings:

- `t` Switch on this feature
- `unless-remote` Switch on this feature but not for remote directories. The term "remote" means here directories which are used via tramp, ange-ftp or efs. So mounted directories are counted not as remote directories here even if such a directory is maybe hosted on a remote machine. But normally only directories in a LAN are mounted so there should be no performance-problems with such mounted directories.
- `nil` Switch off this feature completely.

The option `ecb-prescan-directories-exclude-regexps` offers are more fine granularity to exclude certain directories from this prescan.

`host-accessible-check-valid-time` [User Option]
Time in seconds a cached accessible-state of a remote host is valid. This option is a list where each element specifies how long for a certain remote host the cached ping-state (i.e. if the host is accessible or not) should be valid. During this time-intervall ECB pings such a remote host only once, all other checks use the cached value of that real check. But it the cached value is older than the value of this option ECB will ping again.

Per default ECB discards after 1 minute the cached ping-state of each remote host. But if you are sure that a certain remote host is always accessible (i.e. means in consequence that you are always online when working with ECB and remote-paths) then add an entry to this option with a high valid-interval.

Examples: An entry (".*sourceforge.*" . 3600) ensures that all remote hosts machting the string "sourceforge" will only once pinged during one hour. Or (".*" . 300) would ensure that every remote host would be pinged only once during 5 minutes.

`ping-options` [User Option]
List of options for the ping program. These options can be used to limit how many ICMP packets are emitted. Ping is used to test if a remote host of a remote path (e.g. a tramp-, ange-ftp- or efs-path) is accessible See also `ecb-ping-program`.

`ping-program` [User Option]
Program to send network test packets to a host. See also `ecb-ping-options`.

`prescan-directories-exclude-regexps` [User Option]
Which directories should be excluded from the empty-prescan. If a directory matches any of the regexps of this option it will not be prescanned for emptyness - This option takes only effect if `ecb-prescan-directories-for-emptyness` is not nil.

`show-sources-in-directories-buffer` [User Option]
Show source files in directories buffer.

`source-path` [User Option]
Paths where to find code sources. Each path can have an optional alias that is used as it's display name. If no alias is set, the path is used as display name.

`source-path` [User Option]
Paths where to find code sources. Each path can have an optional alias that is used as it's display name. If no alias is set, the path is used as display name.

Lisp-type of tis option: The value must be a list L whereas each element of L is either

- a simple string which has to be the full path of a directory (this string is displayed in the directory-browser of ECB) or
- a 2-element list whereas the first element is the full path of a directory (string) and the second element is an arbitrary alias (string) for this directory which is then displayed instead of the underlying directory.

`use-speedbar-instead-native-tree-buffer` [User Option]
If true then uses speedbar for directories, sources or methods. This means that speedbar is integrated in the ECB-frame and is displayed in that window normally displaying the standard ECB-directories-buffer, ECB-sources-buffer or ECB-methods-buffer.

This option takes effect in all layouts which contain either a directory window, a sources window or a method window.

This option can have four valid values:

- `nil`: Do not use speedbar (default)
- `dir`: Use speedbar instead of the standard directories-buffer
- `source`: Use speedbar instead of the standard sources-buffer
- `method`: Use speedbar instead of the standard methods-buffer

Note: For directories and sources a similar effect and usability is available by setting this option to `nil` (or `method`) and setting `ecb-show-sources-in-directories-buffer` to not `nil`, because this combination displays also directories and sources in one window.

`ecb-use-speedbar-instead-native-tree-buffer` is for people who like the speedbar way handling directories and source-files or methods and want it in conjunction with ECB.

7.3.4 Group ecb-sources

This group contains settings for the sources-buffer in the ECB:

`read-only-check-exclude-regexps` [User Option]
Which directories should be excluded from the sources-read-only-check. If a directory matches any of the regexps of this option their sources will not be checked if they are writable - This option takes only effect if `ecb-sources-perform-read-only-check` is not nil.

`show-source-file-extension` [User Option]
Show the file extension of source files.

`source-file-regexps` [User Option]

Specifies which files are shown as source files.

This is done on directory-base, which means for each directory-regexp the files to display can be specified. If more than one directory-regexp matches the current selected directory then always the first one (and its related file-exclude/include-regexps) is used! If no directory-regexp matches then all files are displayed for the currently selected directory.

Important note: It is recommended that the *LAST* element of this list should contain an always matching directory-regexp (`".*"`)!

So the value of this option is a list of cons-cells where the car is a directory regexp and the cdr is a 2 element list where the first element is a list of exclude regexps and the second element is a list of include regexps. A file is displayed in the source-buffer of ECB iff: The file does not match any of the exclude regexps OR the file matches at least one of the include regexps.

But regardless of the value of this option a file F is never displayed in the sources-buffer if the directory matches `ecb-sources-exclude-cvsignore` and the directory contains a file .cvsignore which contains F as an entry!

There are three predefined and useful combinations of an exclude and include regexp:

- All files
- All, but no backup, object, lib or ini-files (except .emacs and .gnus). This means all files except those starting with ".", "#" or ending with "~", ".elc", ".obj", ".o", ".lib", ".dll", ".a", ".so". (but including .emacs and .gnus)
- Common source file types (.c, .java etc.)

In addition to these predefined values a custom exclude and include combination can be defined.

Tips for the directory- and file-rexexps: `"$^"` matches no files/directories, `".*"` matches all files/directories.

`sources-buffer-after-create-hook` [User Option]

Local hook running after the creation of the sources-buffer. Every function of this hook is called once without arguments direct after creating the sources-buffer of ECB and it's local key-map. So for example a function could be added which performs calls of `local-set-key` to define new keybindings only for the sources-buffer of ECB.

`sources-buffer-name` [User Option]

Name of the ECB sources buffer. Because it is not a normal buffer for editing you should enclose the name with stars, e.g. "*ECB Sources*".

If it is necessary for you you can get emacs-lisp access to the buffer-object of the ECB-sources-buffer by this name, e.g. by a call of `set-buffer`.

Changes for this option at runtime will take affect only after deactivating and then activating ECB again!

`sources-exclude-cvsignore` [User Option]

Specify if files contained in a '`.cvsignore`' should be excluded.

Value is a list of regular expressions or nil. If you want to exclude files listed in a '`.cvsignore`'-file from being displayed in the ecb-sources-buffer then specify a regexp for such a directory.

If you want to exclude the contents of '`.cvsignore`'-files for every directory then you should add one regexp ".*" which matches every directory.

If you never want to exclude the contents of '`.cvsignore`'-files then set this option to nil.

`sources-menu-sorter` [User Option]

Function which re-sorts the menu-entries of the directories buffer.

If a function then this function is called to sort the menu-entries of the combined menu-entries of the user-menu-extensions of `ecb-sources-menu-user-extension` and the built-in-menu `ecb-sources-menu`. If nil then no special sorting will be done and the user-extensions are placed in front of the built-in-entries.

For the guidelines for such a sorter-function see `ecb-directories-menu-sorter`.

`sources-menu-user-extension` [User Option]

Static user extensions for the popup-menu of the sources buffer. For further explanations see `ecb-directories-menu-user-extension`.

The node-argument of a menu-function contains as data the filename of the source for which the popup-menu has been opened.

Per default the static user-extensions are added at the beginning of the built-in menu-entries of `ecb-sources-menu` but the whole menu can be re-arranged with `ecb-sources-menu-sorter`.

`sources-menu-user-extension-function` [User Option]

Dynamic user extensions for the popup-menu of the sources buffer. A function which has to return a list in the same format like the option `ecb-sources-menu-user-extension`. This function is called when the user opens the popup-menu for the sources buffer.

If no dynamically evaluated menu-extensions should be added to the sources-buffer the function has to return nil. Therefore the default-value of this option is `ignore`.

Per default the dynamic user-extensions are added in front of the static extensions of `ecb-sources-menu-user-extension` but the whole menu can be re-arranged with `ecb-sources-menu-sorter`.

`sources-perform-read-only-check` [User Option]

Check if source-items in the tree-buffers are read-only. If a sourcefile is read-only then it will be displayed with that face set in the option `ecb-source-read-only-face`.

Because this check can be take some time if files are used via a mounted net-drive ECB performs this check stealthily (see `ecb-stealthy-tasks-delay`) so normally the user should not see a performance-decrease or additional waiting-time. But to get sure this option offers three choices: `t`, `unless-remote` and `nil`. See `ecb-prescan-directories-for-emptyness` for an explanation for these three choices.

The option `ecb-read-only-check-exclude-regexps` offers are more fine granularity to exclude the sources of certain directories from the read-only state-check.

`sources-show-node-info` [User Option]

When to display which node-info in the sources-buffer. Define which node info should displayed after moving the mouse over a node (or after a shift click onto the node) in the sources-buffer.

You can define "when" a node-info should be displayed: See `ecb-directories-show-node-info` for the possible choices.

You can define what info should be displayed:

- name: Only the full node-name is displayed.
- file-info: File infos for this file are displayed.
- file-info-full: Fill infos incl. full path for this file are displayed.

Do NOT set this option directly via setq but use always customize!

`sources-sort-ignore-case` [User Option]

Ignore case for sorting the source-files of the Sources-buffer. See also `ecb-sources-sort-method`.

`sources-sort-method` [User Option]

Defines how the source files are sorted.

- `name`: Sorting by name.
- `extension`: Sorting first by extension and then by name.
- `nil`: No sorting, means source files are displayed in the sequence returned by `directory-files` (called without sorting).

See also `ecb-sources-sort-ignore-case`

7.3.5 Group ecb-methods

This group contains settings for the methods-buffer in the ECB:

`auto-expand-tag-tree` [User Option]

Expand the methods-tag-tree automatically if node invisible.

This option has only an effect if option `ecb-highlight-tag-with-point` is switched on too. There are three possible choices:

- `nil`: No auto. expanding of the method buffer.
- `expand-spec`: Auto expand the method-buffer nodes if the node belonging to current tag under point is invisible because its parent-node is collapsed. But expanding is only done if the type of the tag under point in the edit-buffer is contained in `ecb-methods-nodes-expand-spec`.
- `all`: Like expand-spec but expands all tags regardless of the setting in `ecb-methods-nodes-expand-spec`.

This options takes only effect for semantic-sources - means sources supported by semantic!

`auto-expand-tag-tree-collapse-other` [User Option]

Auto. expanding the tag-tree collapses all not related nodes. There are several choices:

- Only if on tag: This means collapsing all nodes which have no relevance for the currently highlighted node will be collapsed, because they are not necessary to make the highlighted node visible. But do this only if point stays onto a tag in the selected edit-window.
- Always: Same as before but collapse also when point doesn't stays on a tag (e.g. between two defuns in elisp) in the selected edit-window. This means in such a situation a full collapsing of the methods-buffer.
- Never: Do not automatically collapse the methods-buffer.

`auto-update-methods-after-save` [User Option]
Automatically updating the ECB method buffer after saving a source.

`default-tag-filter` [User Option]
Default tag-filters for certain files. This option allow to define default tag-filters for certain files which are applied automatically after loading such a file into a buffer. The possible filters are the same as offered by the command `ecb-methods-filter` and they are applied in the same manner - the only difference is they are applied automatically. Please be aware that symbol-filters (e.g. protection-symbols like public or private) must not be inserted with quotes whereas a filter-regexp has to be inserted with surrounding double-quotes! In addition backslashes in a regexp have to be doubled!

For each file-spec (a major-mode plus a file-regexp which both specify a file for which filters should be applied) there can be as much filters as needed - they are layered like with `ecb-methods-filter` too.

Tag-classes which are completely hidden or excluded by the option `ecb-show-tags` will never being displayed in the Methods-buffer regardless of the filters of this option!

`disable-semantic-threshold-alist` [User Option]
Threshold for disabling semantic-parsing Define a threshold fpr buffer-size. Exceeding this threshold disables parsing current buffer by semantic.

This functionality is set on a major-mode base, i.e. for every major-mode a different setting can be used. The value of this option is a list of cons-cells:

- The car is either a major-mode symbol or the special symbol 'default which means if no setting for a certain major-mode is defined then the cdr of the 'default cons-cell is used.
- The cdr is an integer which defines the threshold for the buffer-size for this major-mode.

Example:

```
((default . 1000000)
  (c-mode . 200000))
```

This example whould not parse c-mode buffers exceeding a buffer-size of 200000. And buffers of all other modes would be only parsed if smaller than 1000000.

A setting of `((c-mode . 200000))` would only restrict c-mode buffers to a size of 200000 but would parse all other buffer regardless their size.

`display-image-icons-for-semantic-tags` [User Option]

Display nice and pretty icons for semantic-tags in the Methods-buffer. This option takes only effect if Emacs can display images and if `ecb-tree-buffer-style` is set to `image`.

`exclude-parents-regexp` [User Option]

Regexps which parent classes should not be shown in the methods buffer (see also `ecb-show-parents`). If nil then all parents will be shown if `ecb-show-parents` is not nil.

This options takes only effect for semantic-sources - means sources supported by semantic!

`expand-methods-switch-off-auto-expand` [User Option]

Switch off auto expanding in the ECB-method buffer. If on then auto expanding is switched off after explicit expanding or collapsing by `ecb-expand-methods-nodes`.

This is done with `ecb-toggle-auto-expand-tag-tree` so after the switch off the auto expanding feature can again switched on quickly.

But after explicitly expanding/collapsing the methods-buffer to a certain level the auto. expanding could undo this when the node belonging to current tag under point in the current active edit-window is invisible after `ecb-expand-methods-nodes` - then the auto. expand feature would make this node immediately visible and destroys the explicitly set expand-level.

`font-lock-tags` [User Option]

Adds font-locking (means highlighting) to the ECB-method buffer.

This options takes only effect for semantic-sources - means sources supported by semantic!

`highlight-tag-with-point` [User Option]

How to highlight the method or variable under the cursor.

- `highlight-scroll`: Always scroll the method buffer, so the current method of the edit-window is highlighted in the method-window.
- `highlight`: Only highlight the current method of the edit window in the method window if the method is visible in the method-window.
- `nil`: No highlighting is done.

See also `ecb-highlight-tag-with-point-delay`.

This options takes only effect for semantic-sources - means sources supported by semantic!

`highlight-tag-with-point-delay` [User Option]

Time Emacs must be idle before current tag is highlighted. If nil then there is no delay, means current tag is highlighted immediately. A small value of about 0.25 seconds saves CPU resources and you get even though almost the same effect as if you set no delay. But such a delay prevents also "jumping backward/forward" during scrolling within java-classes if point goes out of method-definition into class-definition. Therefore the default value is a delay of 0.25 seconds.

This options takes only effect for semantic-sources - means sources supported by semantic!

`methods-buffer-after-create-hook` [User Option]

Local hook running after the creation of the methods-buffer. Every function of this hook is called once without arguments direct after creating the methods-buffer of ECB and it's local key-map. So for example a function could be added which performs calls of `local-set-key` to define new keybindings only for the methods-buffer of ECB.

`methods-buffer-name` [User Option]

Name of the ECB methods buffer. Because it is not a normal buffer for editing you should enclose the name with stars, e.g. " *ECB Methods*".

If it is necessary for you you can get emacs-lisp access to the buffer-object of the ECB-methods-buffer by this name, e.g. by a call of `set-buffer`.

Changes for this option at runtime will take affect only after deactivating and then activating ECB again!

`methods-filter-replace-existing` [User Option]

How the methods-filter should be applied to existing filters. There are three different choices:

- `never`: This is the default and means that calling `ecb-methods-filter` always adds the new filter on top of already existing filters. So you can combine several filter to one combined like this example: 'Display only all public methods having the string "test" in its name.' With this setting the filters can only be cleared by calling `ecb-methods-filter` and then choosing "nothing".
- `always`: This means that `ecb-methods-filter` always clears a previous filter before applying the new one.
- `ask`: ECB asks if the new filter should replace the existing ones.

`methods-menu-sorter` [User Option]

Function which re-sorts the menu-entries of the directories buffer.

If a function then this function is called to sort the menu-entries of the combined menu-entries of the user-menu-extensions of `ecb-methods-menu-user-extension` and the built-in-menu `ecb-methods-menu`. If nil then no special sorting will be done and the user-extensions are placed in front of the built-in-entries.

For the guidelines for such a sorter-function see `ecb-directories-menu-sorter`.

`methods-menu-user-extension` [User Option]

Static user extensions for the popup-menu of the methods buffer. For further explanations see `ecb-directories-menu-user-extension`.

The node-argument of a menu-function contains as data the semantic-tag of the method/variable/tag for which the popup-menu has been opened.

Per default the static user-extensions are added at the beginning of the built-in menu-entries of `ecb-methods-menu` but the whole menu can be re-arranged with `ecb-methods-menu-sorter`.

`methods-menu-user-extension-function` [User Option]

Dynamic user extensions for the popup-menu of the methods buffer. A function which has to return a list in the same format like the option `ecb-methods-menu-user-extension`. This function is called when the user opens the popup-menu for

the methods buffer. For an example how such a function can be programmed see `ecb-methods-menu-editwin-entries`.

If no dynamically evaluated menu-extensions should be added to the methods-buffer the function has to return nil. Therefore the default-value of this option is `ignore`.

Per default the dynamic user-extensions are added in front of the static extensions of `ecb-methods-menu-user-extension` but the whole menu can be re-arranged with `ecb-methods-menu-sorter`.

`methods-nodes-collapse-spec` [User Option]

Semantic tag-types collapsed by `ecb-expand-methods-nodes`. For valid values of this option see `ecb-methods-nodes-expand-spec`!

This options takes only effect for semantic-sources - means sources supported by semantic!

`methods-nodes-expand-spec` [User Option]

Semantic tag-types expanded by `ecb-expand-methods-nodes`.

The value of this option is either the symbol `all` (all tags are expanded regardless of their type) or a list of symbols where each symbol is a valid semantic tag-type. For a description of semantic tag types see option `ecb-show-tags`.

But this option also defines if bucket-nodes in the ECB-method-buffer (e.g. "[Variables]") should be expanded. Therefore valid symbols for this list are also all cars of the variable returned by `ecb--semantic-symbol->name-assoc-list`.

If there is a bucket-name (the node-name stripped of the settings in `ecb-bucket-node-display`) which is not contained as cdr in the value returned by `ecb--semantic-symbol->name-assoc-list` then the symbol with this bucket-name as name is also a valid symbol for this list. Example: In ECB there are buckets "[Parents]". The bucket-name is "Parents" and the valid symbol-name is then `Parents`.

This options takes only effect for semantic-sources - means sources supported by semantic!

`methods-separate-prototypes` [User Option]

Separate function-prototypes from the real functions. This is for example useful for C++ and C because these languages distinct between a method-prototype (rsp. function-prototype for C) and the method (rsp. function for C) itself. If this option is not nil then ECB separates the prototypes from the real function/methods. Then with `ecb-show-tags` the user can define different display-settings for each of them. If this option is nil then the prototypes and the real functions are filled in the same bucket and displayed plain and there is no sorting between prototypes and functions possible. If this option is switched on then it is senseful that `ecb-show-tags` contains for all modes which distinct between prototypes and real functions/methods two entries for the tag-type 'function - see the documentation of this option.

`methods-show-node-info` [User Option]

When to display which node-info in the methods-buffer. Define which node info should displayed after moving the mouse over a node (or after a shift click onto the node) in the methods-buffer.

You can define "when" a node-info should be displayed: See `ecb-directories-show-node-info` for the possible choices.

You can define what info should be displayed:

- name: Only the full node name is displayed.
- name+type: The full name + the type of the node (function, class, variable) is displayed.

Do NOT set this option directly via setq but use always customize!

`post-process-semantic-taglist` [User Option]

Define mode-dependent post-processing for the semantic-taglist. This is an alist where the car is a major-mode symbol and the cdr is a list of function-symbols of functions which should be used for post-processing the taglist (returned by `ecb--semantic-bovinate-toplevel`) for a buffer in this major-mode. The first function in the list is called with current semantic taglist of current buffer and must return a valid taglist again. All other functions are called with the result-taglist of its preceding function and have to return a new taglist again.

For oo-programming languages where the methods of a class can be defined outside the class-definition (e.g. C++, Eieio) the function `ecb-group-function-tags-with-parents` can be used to get a much better method-display in the methods-window of ECB, because all method implementations of a class are grouped together.

Another senseful usage is to filter out certain tags, e.g. prototype tags in `c-mode`. For this you can set `ecb-filter-c-prototyp-tags`.

This options takes only effect for semantic-sources - means sources supported by semantic!

`show-only-positioned-tags` [User Option]

Show only nodes in the method-buffer which are "jump-able". If not nil then ECB displays in the method-buffer only nodes which are "jump-able", i.e. after selecting it by clicking or with RET then ECB jumps to the corresponding location in the edit-window. Example: With CLOS or Eieio source-code there can exist some position-less nodes like variable-attributes in a `defclass` form which are only displayed if this option is nil. Displaying such nodes can be senseful even if they can not be jumped.

This options takes only effect for semantic-sources - means sources supported by semantic!

`show-tags` [User Option]

How to show tags in the methods buffer first time after find-file. This functionality is set on a major-mode base, i.e. for every major-mode a different setting can be used. The value of this option is a list of cons-cells:

The car is either a major-mode symbol or the special symbol 'default which means if no setting for a certain major-mode is defined then the cdr of the 'default cons-cell is used. This option should always contain a default-setting!

The cdr is a list where each element represents a type of tags:

```
(<tag type> <display type> <sort method>)
```

There can be more than 1 element for a certain <tag type>. This is for example useful for C++ and C because these languages distinct between a method-prototype (rsp. function-prototype for C) and the method (rsp. function for C) itself. The default value of these option contains two entries for <tag type> is `function` whereas the first one is responsible for the "real" methods (rsp. functions) and the second one for the prototypes. So if the methods should be flattened and the prototypes collapsed the show-tags-list for C++ and C must contain two entries for <tag type> `function`, the first one defined as `flattened` and the second one defined as `collapsed`.

The tags in the methods buffer are displayed in the order as they appear in this list.

`<tag type>`
: A Semantic tag type symbol (function, variable, rule, include etc.) or one of the following:
  - `t`: All tag types not specified anywhere else in the list.
  - `parent`: The parents of a type.

`<display type>`
: A symbol which describes how the tags of this type shall be shown:
  - `expanded`: The tags are shown in an expanded node.
  - `collapsed`: The tags are shown in a collapsed node.
  - `flattened`: The tags are added to the parent node.
  - `hidden`: The tags are not shown.

`<sort method>`
: A symbol describing how to sort the tags of this type:
  - `name`: Sort by the tag name.
  - `access`: Sort by tag access (public, protected, private) and then by name.
  - `nil`: Don't sort tags. They appear in the same order as in the source buffer.

This options takes only effect for semantic-sources - means sources supported by semantic!

`tag-display-function` [User Option]

Function to use for displaying tags in the methods buffer. This functionality is set on major-mode base, i.e. for every major-mode a different function can be used. The value of this option is a list of cons-cells:

- The car is either a major-mode symbol or the special symbol 'default which means if no function for a certain major-mode is defined then the cdr of the 'default cons-cell is used.
- The cdr is the function used for displaying a tag in the related major-mode.

Every function is called with 3 arguments:

1. The tag
2. The parent-tag of tag (can be nil)
3. The value of `ecb-font-lock-tags`.

Every function must return the display of the tag as string, colorized if the third argument is not nil.

The following functions are predefined:

- For each element E of `ecb--semantic-format-function-alist` exists a function with name "ecb–<(cdr E)>". These functions are just aliase to the builtin format-functions of semantic. See the docstring of these functions to see what they do. Example: (semantic-name-nonterminal . semantic-format-tag-name) is an element of `ecb--semantic-format-function-alist`. Therefore the alias-function for this element is named `ecb--semantic-format-tag-name`.
- For every cdr in `ecb--semantic-format-function-alist` with name "semantic-XYZ" a function with name "ecb-XYC" is predefined. The differences between the semantic- and the ECB-version are:
  - The ECB-version displays for type tags only the type-name and nothing else (exception: In c++-mode a template specifier is appended to the type-name if a template instead a normal class).
  - The ECB-version displays type-tags according to the setting in `ecb-type-tag-display`. This is useful for better recognizing different classes, structs etc. in the ECB-method window.

  For all tags which are not types the display of the ECB-version is identical to the semantic version. Example: For `ecb--semantic-format-tag-name` (one of the builtin semantic formatters) the pendant is `ecb-format-tag-name`.

This functionality also allows the user to display tags as UML. To enable this functionality set the function for a major-mode \(e.g. `jde-mode`) to `ecb--semantic-format-tag-uml-concise-prototype`, `ecb--semantic-format-tag-uml-prototype`, or `ecb--semantic-format-tag-uml-abbreviate` the ECB-versions of these functions.

If the value is `nil`, i.e. neither a function for a major-mode is defined nor the special 'default, then `ecb--semantic-format-tag-prototype` is used for displaying the tags.

This options takes only effect for semantic-sources - means sources supported by semantic!

`tag-jump-sets-mark` [User Option]

Set the mark after jumping to a tag from the ECB-method buffer. If set the user can easily jump back.

`tag-visit-post-actions` [User Option]

Actions to perform after visiting a tag from the Method-buffer. With this option actions can be added which will be performed after visiting the start of the tag in the source-buffer.

This functionality is set on a `major-mode` base, i.e. for every `major-mode` a different setting can be used. The value of this option is a list of cons-cells:

- The car is either a `major-mode` symbol or the special symbol 'default.
- The cdr is a list of action-functions or nil.

ECB first performs all actions defined for the special symbol 'default (if any) and then all actions defined for current `major-mode` (if any).

ECB offers some predefined senseful action-functions. Currently there are: `ecb-tag-visit-highlight-tag-header` `ecb-tag-visit-smart-tag-start` `ecb-tag-visit-recenter` `ecb-tag-visit-recenter-top` `ecb-tag-visit-goto-doc-start` `ecb-tag-visit-narrow-tag` See the documentation of these function for details what they do.

But you can add any arbitrary function if the following conditions are fulfilled: The function gets the semantic tag as argument, returns the (new) point after finishing its job and the function must not put the point outside the tag-boundaries of the tag-argument.

`type-tag-display` [User Option]

How to display semantic type-tags in the methods buffer. Normally all tag displaying, colorizing and facing is done by semantic according to the value returned by `ecb--semantic-format-face-alist` and the semantic display-function (e.g. one from `ecb--semantic-format-tag-functions`). But sometimes a finer distinction in displaying the different type specifiers of type-tags can be useful. For a description when this option is evaluated look at `ecb-tag-display-function`!

This functionality is set on a major-mode base, i.e. for every major-mode a different setting can be used. The value of this option is a list of cons-cells:

- The car is either a major-mode symbol or the special symbol 'default which means if no setting for a certain major-mode is defined then the cdr of the 'default cons-cell is used.
- The cdr is a list of 3-element-lists:
  1. First entry is a semantic type specifier in string-form. Current available type specifiers are for example "class", "interface", "struct", "typedef" and "enum". In addition to these ones there is also a special ECB type specifier "group" which is related to grouping tags (see `ecb-post-process-semantic-taglist` and `ecb-group-function-tags-with-parents`). Any arbitrary specifier can be set here but if it is not "group" or not known by semantic it will be useless.
  2. Second entry is a flag which indicates if the type-specifier string from (1.) itself should be removed (if there is any) from the display.
  3. Third entry is the face which is used in the ECB-method window to display type-tags with this specifier. ECB has some predefined faces for this (`ecb-type-tag-class-face`, `ecb-type-tag-interface-face`, `ecb-type-tag-struct-face`, `ecb-type-tag-typedef-face`, `ecb-type-tag-union-face`, `ecb-type-tag-enum-face` and `ecb-type-tag-group-face`) but any arbitrary face can be set here. This face is merged with the faces semantic already uses to display a tag, i.e. the result is a display where all face-attributes of the ECB-face take effect plus all face-attributes of the semantic-faces which are not set in the ECB-face (with XEmacs this merge doesn't work so here the ECB-face replaces the semantic-faces; this may be fixed in future versions).

The default value is nil means there is no special ECB-displaying of type-tags in addition to the displaying and colorizing semantic does. But a value like the following could be a useful setting:

```
((default
   ("class" t ecb-type-tag-class-face)
   ("group" nil ecb-type-tag-group-face))
 (c-mode
  ("struct" nil ecb-type-tag-struct-face)
  ("typedef" nil ecb-type-tag-typedef-face)))
```

This means that in `c-mode` only "struct"s and "typedef"s are displayed with special faces (the specifiers itself are not removed) and in all other modes "class"s and grouping-tags (see `ecb-tag-display-function`, `ecb-group-function-tags-with-parents`) have special faces and the "class" specifier-string is removed from the display.

This options takes only effect for semantic-sources - means sources supported by semantic!

`type-tag-expansion` [User Option]

Default expansion of semantic type-tags. Semantic groups type-tags in different type-specifiers. Current available type specifiers are for example "class", "interface", "struct", "typedef", "union" and "enum". In addition to these ones there is also a special ECB type-specifier "group" which is related to grouping tags (see `ecb-post-process-semantic-taglist`).

This option defines which type-specifiers should be expanded at file-open-time. Any arbitrary specifier can be set here but if it is not "group" or not known by semantic it will be useless.

This functionality is set on a major-mode base, i.e. for every major-mode a different setting can be used. The value of this option is a list of cons-cells:

- The car is either a major-mode symbol or the special symbol `default` which means if no setting for a certain major-mode is defined then the cdr of the `default` cons-cell is used.
- The cdr is either a list of type-specifiers which should be expanded at file-open-time or the symbol `all-specifiers` (then a type-tag is always expanded regardless of its type-specifier).

This options takes only effect for semantic-sources - means sources supported by semantic!

7.3.6 Group ecb-history

This group contains settings for the history-buffer in the ECB:

`history-buffer-after-create-hook` [User Option]

Local hook running after the creation of the history-buffer. Every function of this hook is called once without arguments direct after creating the history-buffer of ECB and it's local key-map. So for example a function could be added which performs calls of `local-set-key` to define new keybindings only for the history-buffer of ECB.

`history-buffer-name` [User Option]
Name of the ECB history buffer. Because it is not a normal buffer for editing you should enclose the name with stars, e.g. "*ECB History*".

If it is necessary for you you can get emacs-lisp access to the buffer-object of the ECB-history-buffer by this name, e.g. by a call of `set-buffer`.

Changes for this option at runtime will take affect only after deactivating and then activating ECB again!

`history-exclude-file-regexps` [User Option]
List of regexps which exclude source-files from being historized. Be aware that each always full filenames (ie. incl. full path) are matched against these regexps! Therefore be carefore with regexps beginning with `^`!

`history-item-name` [User Option]
The name to use for items in the history buffer.

`history-menu-sorter` [User Option]
Function which re-sorts the menu-entries of the directories buffer.

If a function then this function is called to sort the menu-entries of the combined menu-entries of the user-menu-extensions of `ecb-history-menu-user-extension` and the built-in-menu `ecb-history-menu`. If nil then no special sorting will be done and the user-extensions are placed in front of the built-in-entries.

For the guidelines for such a sorter-function see `ecb-directories-menu-sorter`.

`history-menu-user-extension` [User Option]
Static user extensions for the popup-menu of the history buffer. For further explanations see `ecb-directories-menu-user-extension`.

The node-argument of a menu-function contains as data the filename of the source for which the popup-menu has been opened.

Per default the static user-extensions are added at the beginning of the built-in menu-entries of `ecb-history-menu` but the whole menu can be re-arranged with `ecb-history-menu-sorter`.

`history-menu-user-extension-function` [User Option]
Dynamic user extensions for the popup-menu of the history buffer. A function which has to return a list in the same format like the option `ecb-history-menu-user-extension`. This function is called when the user opens the popup-menu for the history buffer.

If no dynamically evaluated menu-extensions should be added to the history-buffer the function has to return nil. Therefore the default-value of this option is `ignore`.

Per default the dynamic user-extensions are added in front of the static extensions of `ecb-history-menu-user-extension` but the whole menu can be re-arranged with `ecb-history-menu-sorter`.

`history-show-node-info` [User Option]
When to display which node-info in the history-buffer. Define which node info should displayed after moving the mouse over a node (or after a shift click onto the node) in the history-buffer.

You can define "when" a node-info should be displayed: See `ecb-directories-show-node-info` for the possible choices.

You can define what info should be displayed: See `ecb-directories-show-node-info` for the possible choices.

Do NOT set this option directly via setq but use always customize!

`history-sort-ignore-case` [User Option]
Ignore case for sorting the history-entries. See also `ecb-history-sort-method`.

`history-sort-method` [User Option]
Defines how the entries in the history-buffer are sorted.

- `name`: Sorting by name (default).
- `extension`: Sorting first by extension and then by name.
- `nil`: No sorting, means the most recently used buffers are on the top of the history and the seldom used buffers at the bottom.

See also `ecb-history-sort-ignore-case`.

`kill-buffer-clears-history` [User Option]
Define if `kill-buffer` should also clear the history. There are three options:

- `auto`: Removes automatically the corresponding history-entry after the buffer has been killed.
- `ask`: Asks, if the history-entry should be removed after the kill.
- `nil`: `kill-buffer` does not affect the history (this is the default).

7.3.7 Group ecb-analyse

`analyse-bucket-element-face` [User Option]
Basic face for displaying elements of buckets in the ECB-analyse-buffer. This defines the basic face for the elements of category-buckets like Context, Prefix, Completions etc. in the ECB-analyse-buffer.

Changes take first effect after finishing and reactivating ECB!

`analyse-bucket-node-face` [User Option]
Basic face for displaying a bucket-node in the ECB-analyse-buffer. This defines the basic face for the bucket-nodes like Context, Prefix, Completions etc. in the ECB-analyse-buffer.

Changes take first effect after finishing and reactivating ECB!

`analyse-buffer-after-create-hook` [User Option]
Local hook running after the creation of the analyse-buffer. Every function of this hook is called once without arguments direct after creating the analyse-buffer of ECB and it's local key-map. So for example a function could be added which performs calls of `local-set-key` to define new key-bindings only for the analyse-buffer of ECB.

`analyse-buffer-name` [User Option]

Name of the ECB analyse buffer. Because it is not a normal buffer for editing you should enclose the name with stars, e.g. "*ECB Analyse*".

If it is necessary for you you can get emacs-lisp access to the buffer-object of the ECB-analyse-buffer by this name, e.g. by a call of `set-buffer`.

Changes for this option at runtime will take affect only after deactivating and then activating ECB again!

`analyse-collapsed-buckets` [User Option]

Buckets collapsed when displaying the current semantic analysis. The semantic analyse-modul offers several categories of analysis which are called buckets here. These are for example:

Context: The current context, which is the current function/method, variable, class etc. (what exactly depends on the programming language) point is in. This means not the current function/method/variable/class-name point stand on but the current surrounding context. Example: If point stays somewhere within a defun-definition in emacs-lisp or within a java-method then this defun rsp. method is the context. In object oriented languages this can be the full hierachy, i.e. not only the current method, but the current method, the class of this method, the superclass of this class and so on!

Arguments: The arguments of the context if the context is a function/method.

Local Variables: All accessible and bound local variables visible at current point.

Prefix: The currently parsed prefix, which is mostly the current identifier point stands on.

Assignee: See the semantic user-manual

Function: Current function-name point stands on.

Argument #: When point is located within a function-call then this is the number of the argument point stands on.

Completions: All possible completions for current prefix (see above). This is probably the most helpful bucket.

If one of these categories/buckets are not needed per default then add the bucket-name (s.a.) to this option and ECB will per default collapse this bucket. So most needed buckets are better visible in the analyse-buffer.

`analyse-face` [User Option]

Face used for highlighting current entry in the analyse buffer. If the face `ecb-default-highlight-face` is used then the display of all ECB-tree-buffers can be changed by modifying only the face `ecb-default-highlight-face`.

Changes take first effect after finishing and reactivating ECB!

`analyse-fontified-buckets` [User Option]

Buckets whose elements should be fontified as in the methods-buffer. If the name of a category/bucket is contained in this option then all elements of this bucket will be displayed as in the methods-buffer - at least if an element is a semantic-tag. This means if `ecb-font-lock-tags` is not nil these elements will be fontified and also

displayed with an appropriate icon if possible. The default value does this only for the Context-bucket because for most of the other buckets this makes not really much sense.

For available buckets see `ecb-analyse-collapsed-buckets`.

For the faces used to display a bucket-node itself or bucket-elements not fontified see the options `ecb-analyse-bucket-node-face` rsp. `ecb-analyse-bucket-element-face`.

`analyse-gen-tag-info-fn` [User Option]

Which info should be displayed for a tag of the analyse-buffer. If nil then the default information about a tag will be displayed. If a function then this function gets as argument the tag for which tag-information should be displayed. This function has to return a string which will be then display as tag-info. This string has to be fully formatted (e.g. must already include line-breaks if the tag-info should be displayed in several lines).

See `ecb-analyse-show-tag-info-fn` how the tag-info is displayed.

`analyse-general-face` [User Option]

Basic face for the ECB analyse buffer. This defines the basic face the whole history buffer should displayed with. If the face `ecb-default-general-face` is used then the display of all ECB-tree-buffers can be changed by modifying only the face `ecb-default-general-face`.

Changes take first effect after finishing and reactivating ECB!

`analyse-show-node-info` [User Option]

When to display which node-info in the history-buffer. Define which node info should displayed after moving the mouse over a node (or after a shift click onto the node) in the history-buffer.

You can define "when" a node-info should be displayed: See `ecb-directories-show-node-info` for the possible choices.

You can define what info should be displayed:

- name: The full name of the node
- full-info: All infos available to a node.

Do NOT set this option directly via setq but use always customize!

`analyse-show-tag-info-fn` [User Option]

How to display the tag-info for a tag of the analyse-buffer. The value of this option is a function which will be called with the info-string generated for the current tag of the analyse-buffer. This function must do all things necessary for displaying this info. When this function is called the window stored in `ecb-last-edit-window-with-point` is the selected window!

ECB offers two builtin ways: Display the info in the echo-area (via the function `message`) or in a temp-buffer in the edit-area (via the function `ecb-analyse-show-tag-info-in-temp-buffer`). Default is echo-area-display.

See also `ecb-analyse-gen-tag-info-fn`.

7.3.8 Group ecb-layout

This group contains settings for the screen-layout of the ECB:

`activate-before-new-frame-created-hook` [User Option]
Normal hook run before the new ECB-frame is created if `ecb-new-ecb-frame` is not nil (otherwise this hook is not evaluated).

`advice-window-functions-signal-error` [User Option]
Signal an error if an adviced function can not do its job. If not nil then an error is signaled if one of the adviced user-commands can not do its job. So for example if the user tries to split the compile-window or an ecb-tree-window or if one tries to switch to another buffer in one of the ecb-tree-windows. For details see the documentation of each of the adviced functions to get info when an error is signaled.

If this option is nil then no error is signaled but the called adviced function does simply nothing.

Default is nil but it can also be useful to signal errors - so you see when call a function in a situation which is not supported by this function.

`fix-window-size` [User Option]
Fix size of the ECB-windows/buffers even after frame-resizing. The fix type (valid values are nil, t, width and height) can either be set on a layout-basis (means a different value for each layout) or one value can be set for all layouts. For the latter case there is an additional value `auto` which choose autom. the senseful fix-type depending on the current layout-type: For top-layouts the fix-type `height` and for all other layout-types the fix-type `width`.

For a detailed description of the valid values see documentation of `window-size-fixed` which is newly introduced in GNU Emacs 21 and is only available there. Therefore this option takes only effect with GNU Emacs >= 21. This option has no effect with XEmacs because it does not support the feature `window-size-fixed`.

Note1: Manually resizing the ECB-windows via `enlarge-window`, `shrink-window`, `mouse-drag-vertical-line` and `mouse-drag-mode-line` is still possible even if the window-sizes are fixed for frame-resizing!

Note2: The description of `window-size-fixed` in the elisp-info-manual is more detailed than the description offered by *C-h v*!

Note3: With Emacs < 22 there seems to be no distinction between 'width, 'height and t. Therefore this option takes no effect (means all ecb-windows have always unfixed sizes) with Emacs < 22 if `ecb-compile-window-height` is not nil.

Per default no window-size fixing has been done.

`hide-ecb-windows-after-hook` [User Option]
Hooks run direct after the ECB windows have been hidden. Hiding was done either by `ecb-toggle-ecb-windows` or `ecb-hide-ecb-windows`.

IMPORTANT: Hiding the ECB-windows is internally done by calling `ecb-redraw-layout` and therefore also the hooks `ecb-redraw-layout-before-hook` and `ecb-redraw-layout-after-hook` are evaluated. The hook-sequence is analogous to that described in `ecb-show-ecb-windows-after-hook`.

`hide-ecb-windows-before-hook` [User Option]

Hook run direct before the ECB windows will be hidden. Hiding is done either by `ecb-toggle-ecb-windows` or `ecb-hide-ecb-windows`. This means that at runtime of this hook all the ECB-tree-windows of current layout are visible.

IMPORTANT: Hiding the ECB-windows is internally done by calling `ecb-redraw-layout` and therefore also the hooks `ecb-redraw-layout-before-hook` and `ecb-redraw-layout-after-hook` are evaluated. The hook-sequence is analogous to that described in `ecb-show-ecb-windows-before-hook`.

`ignore-display-buffer-function` [User Option]

Adviced `display-buffer` ignores `display-buffer-function`. This means, that the adviced version of `display-buffer` ignores the value of `display-buffer-function` when called for the `ecb-frame`. If this variable should not be ignored then the function of `display-buffer-function` is completely responsible which window is used for the buffer to display - no smart ECB-logic will help to deal best with the ECB-window-layout! You can define if and when `display-buffer-function` should be ignored:

- only when persistent compile window is used - i.e. if `ecb-compile-window-height` is not nil
- always when ECB is active - that means ignore when ECB is active otherwise not - this is the default value
- never, the adviced version of `display-buffer` always uses the value of `display-buffer-function` if the value is a function.

`ignore-special-display` [User Option]

Ignore special-display-handling. This means, that all values of `special-display-function`, `special-display-buffer-names` and `special-display-regexps` are ignored

- only when persistent compile window is used - i.e. if `ecb-compile-window-height` is not nil - this is the default value.
- always when ECB is active - that means no special-display-handling of buffers when ECB is active
- never, i.e. special-dislay-handling depends on the values of the options `special-display-function`, `special-display-buffer-names` and `special-display-regexps`.

`layout-always-operate-in-edit-window` [User Option]

Adviced window functions work always in the edit-window. If we are in an ECB special buffer (methods, directories, etc), and any of the adviced windowing functions is called interactively, we will select first an edit-window according to the value of `ecb-mouse-click-destination`. This is useful if you have any functions that use such functions and you don't want them to fail with an error complaining that the current buffer can not be split, or something similar.

Because this may not be desirable in all situations and for all adviced functions this can be enabled separately for function where it is senseful. If the symbol of an adviced function is contained in the value of this option, then the edit-window is first selected

otherwise either an error is reported or some other special reaction (depends on `ecb-advice-window-functions-signal-error`); see the documentation of the adviced functions for this.

Per default this is only enabled for `switch-to-buffer`.

`layout-debug-mode` [User Option]

Write debug-information about layout-operations in the Messages-buffer. Normally there should be no need to set this option to true but if there are problems to display buffers in the compile-window of ECB (e.g. buffers which should be displayed there aren't or the temporally enlarging-mechanism does not do what you think it should do etc...) then please do the following steps:

1. Set `ecb-layout-debug-mode` to not nil
2. Reproduce the wrong behavior exactly by repeating all the operations which lead to the problem.
3. Now send immediately a bug report with `ecb-submit-problem-report`.
4. Set `ecb-layout-debug-mode` back to nil if you do not want further debugging output in the *Messages* buffer

`layout-name` [User Option]

Select a window layout of ECB. Value is any arbitrary string. There are four different types of layouts: left, right, top and left-right, which means the location of the ECB-tree-windows in the ECB-frame. Currently there are 20 predefined layouts; names see below. You can savely try out any of them by changing this value and saving it only for the current session. If you are sure which layout you want you can save it for future sessions. To get a picture of the layout for name <name> call 'ecb-show-layout-help'. `ecb-layout-function-9`.

Currently available layouts:

- Left layouts: left1 left2 left3 left4 left5 left6 left7 left8 left9 left10 left11 left12 left13 left14 left15
- Right layouts: right1
- Top layouts: top1 top2
- Left-right layouts: leftright1 leftright2

Regardless of the settings you define here: If you have destroyed or changed the ECB-screen-layout by any action you can always go back to this layout with `ecb-redraw-layout`

`layout-window-sizes` [User Option]

Specifies the sizes of the ECB windows for each layout. The easiest way (and also the strongly recommended way) to change this variable is to change the window sizes by dragging the window borders using the mouse and then store the window sizes by calling the command `ecb-store-window-sizes`. Next time the layout is redrawn the values stored in this option will be used.

If `ecb-store-window-sizes` is used then the windows sizes are stored per default as fractions of current frame-width and -height of the ecb-frame, so the stored values

will "work" for other frame sizes too. But if you call `ecb-store-window-sizes` with a prefix-argument then the fixed values of current width and height are stored!

If this option is set "by hand" (i.e. not by `ecb-store-window-sizes`) then the following is important:

- It is recommended to use fractions of frame-width and -height!.
- The order of the sequence of the inserted window sizes must be the same as `other-window` (the not-adviced version!) would walk!

`maximize-ecb-window-after-selection` [User Option]

If not nil maximize current tree-window after selection. When selecting another not-tree-window after such an automatic maximizing all tree-windows of current layout are displayed again. But a tree-window is not maximized if either a node has been selected via primary- oder secondarc mouse-button or the popup-menu of that tree-buffer has been opened.

`maximize-next-after-maximized-select` [User Option]

Maximizes the next logical tree-window after a maximized node-selection. Selecting a node in a maximized tree-window is handled very smart by ECB:

If a tree-buffer-name is not contained in this option then selecting a node in this maximized tree-buffer automatically "minimizes" that tree-window (i.e. displays all windows of the current layout) so the user can perform the next logical step (e.g. the next logical step after selecting a directory in the directories-buffer is to select a source-file - therefore the sources-buffer of current layout has to be displayed - if the current layout contains one; the next logical step of selecting a source-file is probably to jump to a certain tag via the methods-buffer).

If a tree-buffer-name is contained in this option then selecting a node in this tree-buffer automatically maximizes the next logical tree-window (e.g. directories –> sources, sources/history –> methods). But if the current maximized tree-buffer is also contained in the option `ecb-tree-do-not-leave-window-after-select` (see also the tree-buffer-command `ecb-toggle-do-not-leave-window-after-select` which is bound to `C-t` in each tree-buffer) then ECB does *not* maximize the next logical tree-window but point stays in the currently maximized tree-buffer so for example the user can select more than one node (e.g. more than one source-file from the sources-buffer.

The tree-buffer-name can either be defined as plain string or with a symbol which contains the buffer-name as value. The latter one is recommended for the builtin ECB-tree-buffers because then simply the related option-symbol can be used (e.g. `ecb-directories-buffer-name`, `ecb-sources-buffer-name` or `ecb-history-buffer-name`).

In future versions this option will probably also allow to define the next logical tree-buffer for a tree-buffer - currently this is hard-coded as follows:

- directories –next-logical–> sources
- sources –next-logical–> methods
- history –next-logical–> methods.

`maximize-next-after-maximized-select` [User Option]

Maximizes the next logical tree-window after a maximized node-selection. Selecting a node in a maximized tree-window is handled very smart by ECB:

If this option is nil then selecting a node in a maximized directories- sources- or history-tree-buffer automatically "minimizes" that tree-window (i.e. displays all windows of the current layout) so the user can perform the next logical step (e.g. the next logical step after selecting a directory in the directories-buffer is to select a source-file - therefore the sources-buffer of current layout has to be displayed - if the current layout contains one; the next logical step of selecting a source-file is probably to jump to a certain tag via the methods-buffer).

If this option is not nil then selecting a node in a maximized directories- sources- or history-tree-buffer automatically maximizes the next logical tree-window (directories –> sources, sources/history –> methods). But if the current maximized tree-buffer is contained in the option `ecb-tree-do-not-leave-window-after-select` (see also the tree-buffer-command `ecb-toggle-do-not-leave-window-after-select` which is bound to `C-t` in each tree-buffer) then ECB does *not* maximize the next logical tree-window but point stays in the currently maximized tree-buffer so for example the user can select more than one source-file from the sources-buffer.

`new-ecb-frame` [User Option]

Create a new frame at activation time of ECB.

`other-window-behavior` [User Option]

The behavior of ECB concerning getting an "other window". The following settings are possible:

`all`:

ECB will cycle through all windows of the ECB-frame or scroll simply the next window in the ECB-frame, means it behaves like the original `other-window` rsp. the original `other-window-for-scrolling`.

`only-edit`:

ECB will only cycle through the edit-windows of ECB or only scroll another edit-window. If the selected window is not an edit-window then it behaves like with value `all`.

`edit-and-compile`:

Like `only-edit` plus the compile window if any. If the selected window is neither an edit-window nor the compile-window then it behaves like with value `all`.

`smart`:

With this setting ECB tries to choose the `other-window`-destination or the "other window" to scroll in a smart and intuitive way: If point is in one of the edit-windows and if the edit-area is splitted then always the "next" edit-window is choosen (whereas the next edit-window of the last edit-window is the first edit-window)- if the edit-area is unsplitted then the compile-window is used if there is one. In the context of an `other-window`-call the *ARG* of `other-window` will be taken into account.

If one of the special ecb-windows is selected then always the "next" ecb-window is choosen (whereas the next ecb-window of the last ecb-window is the first ecb-window).

In the context of an `other-window`-call the *ARG* of `other-window` will be taken into account. If there is only one ecb-window then ECB considers also the edit-windows

If the compile-window is selected then always the last edit-window which had the point will be used unless `other-window` has been called with a prefix-argument unequal 1.

If there is an active minibuffer:

Regardless of the allowed values above ECB handles the situation of an active minibuffer during a call to `other-window` or `scroll-other-window` like follows:

If the minibuffer-window is selected then ECB always chooses the window `minibuffer-scroll-window` points to (when this variable is set, otherwise the compile-window or the last selected edit-window is choosen) when the called command is called to choose the 1. next window (always true for scrolling another window or true when `other-window` called without prefix-arg or with prefix-arg equal 1). Otherwise the window ARG steps away is choosen (in case of `other-window`).

If there is an active minibuffer but the minibuffer-window is not selected then `other-window` and `scroll-other-window` behave like the original version.

In addition to the allowed values above the value of this option can also be a function:

A function:

This function gets seven arguments:

1. A canonical list of all currently visible windows of the `ecb-frame`
2. A canonical list of all currently visible edit-windows
3. A canonical list of all currently visible ecb-windows
4. The window-object of the compile-window if there is any.
5. The minibuffer-window of the ECB-frame if there is an active minibuffer.
6. The result of the function `ecb-where-is-point` - see the documentation of this function for details.
7. An integer which indicates how many steps away from the current selected window the "other-window" is. Is nil when this function is called in another context then for `other-window`.

The function has to return a window-object which is then used as "other window" for the command `other-window` or for scrolling another window (e.g. with `scroll-other-window`). Such a function has to handle properly all situation for itself. `ecb-get-other-window-smart` is an example for such a function.

`redraw-layout-after-hook` [User Option]

Hooks run direct before the ECB windows will be shown either by `ecb-toggle-ecb-windows` or `ecb-show-ecb-windows`. This means that at runtime of this hook the ECB-windows are already visible.

`redraw-layout-before-hook` [User Option]

Hooks run direct before the ECB-layout will be redrawn by either `ecb-redraw-layout`.

`redraw-layout-quickly` [User Option]

If non-nil, we will attempt to redraw the layout quickly. Please read also carefully the documentation of `ecb-redraw-layout`.

`select-edit-window-on-redraw` [User Option]
Select the first edit window on `ecb-redraw-layout`.

`show-ecb-windows-after-hook` [User Option]
Hooks run direct before the ECB windows will be shown either by `ecb-toggle-ecb-windows` or `ecb-show-ecb-windows`. This means that at runtime of this hook the ECB-windows are already visible.

IMPORTANT: Showing the hidden ECB-windows is internally done by calling `ecb-redraw-layout` and therefore also the hooks `ecb-redraw-layout-before-hook` and `ecb-redraw-layout-after-hook` are evaluated. So there is the following sequence of hooks during the process of showing the hidden ECB-windows:

1. `ecb-show-ecb-windows-before-hook`
2. `ecb-redraw-layout-before-hook`
3. <redrawing the layout to show the hidden ECB-windows>
4. `ecb-redraw-layout-after-hook`
5. `ecb-show-ecb-windows-after-hook`

So be aware which code you add to which hook!

`show-ecb-windows-before-hook` [User Option]
Hooks run direct before the ECB windows will be shown either by `ecb-toggle-ecb-windows` or `ecb-show-ecb-windows`. This means that at runtime of this hook the ECB-windows are still hidden.

IMPORTANT: Showing the hidden ECB-windows is internally done by calling `ecb-redraw-layout` and therefore also the hooks `ecb-redraw-layout-before-hook` and `ecb-redraw-layout-after-hook` are evaluated. So there is the following sequence of hooks during the process of showing the hidden ECB-windows:

1. `ecb-show-ecb-windows-before-hook`
2. `ecb-redraw-layout-before-hook`
3. <redrawing the layout to show the hidden ECB-windows>
4. `ecb-redraw-layout-after-hook`
5. `ecb-show-ecb-windows-after-hook`

So be aware which code you add to which hook!

`split-edit-window-after-start` [User Option]
Sets if and how the edit window should be splitted after ECB-start. But be aware: This option determines only if and how the edit-window should be splitted at start-time of ECB. There are five different values allowed for this option:

- `nil`: Do not split the edit-area of ECB after activation, i.e. there will be only one edit-window after starting ECB.
- `horizontal`: Split the edit-area in 2 edit-windows side by side.
- `vertical`: Split the edit-area in 2 edit-windows, one above the other.
- `before-activation`: Split the edit-area as before the ECB-start, i.e. the edit-area will have after start a window-layout as the whole frame had before the start of ECB.

- `before-deactivation`: Split the edit-area into a window-layout ECB had in its edit-area direct before the ECB-deactivation. This value preserves the full state between activations of ECB, i.e. the visibility of the ECB-windows, the visibility of a compile-window and also the full split-state of the edit-area. But this can only be done if important layout-options have not been changed in the meanwhile. These are the options `ecb-layout-name`, `ecb-compile-window-height`, `ecb-compile-window-width`, `ecb-windows-width` and `ecb-windows-height`.

Default value is `before-deactivation`.

Some remarks to the value `before-activation`: If this value has been set then ECB needs three permanent adivces even when ECB is deactivated: `split-window`, `delete-window` and `delete-other-windows`. But these advices do not change any behavior of these functions but only storing in an internal ECB-variable the facts that a window has been splitted or deleted. In addition to this these advices are 100% error-save, means the functionality of the original functions will be performed in every(!) case even if within the advice an error occurs (but normally there can no errors occur in these advices because they are very simple). Conclusion: If you want really all ECB-advices being disabled after deactivating ECB then you have to set this option to other values then `before-activation`. But setting this variable to this value is really completely save.

`toggle-layout-sequence` [User Option]

Toggle sequence for layout toggling with `ecb-toggle-layout`. Every element of this list has to be a valid layout-name i.e. either one of the predefined layouts or one of the user-defined layouts.

You can add here as many layouts as you want but to use this option most effective you should not add more than 2 or 3 layouts so every layout can be accessed very fast by toggling with `ecb-toggle-layout`. It is also senseful to add layouts which have the same principal outline, i.e. all their tree-buffers are on the same side of the frame and the tree-buffer-"column" (or -"row") has identical size for the layouts.

Recommended values are for example:

- ("left10" "left15"), toggles between methods and directories/history
- ("left10" "left13"), toggles between methods and directories
- ("left10" "left14"), toggles between methods and history
- ("left10" "left13" "left14"), toggles between methods, history and directories

See also option `ecb-show-sources-in-directories-buffer`.

This option makes only sense if the value is a list with more than 1 element!

`windows-height` [User Option]

The height of the ECB windows in lines for top-layouts. If the number is less than 1.0 the width is a fraction of the frame height.

`windows-width` [User Option]

The width of the ECB windows in columns for left- and right layouts. If the number is less than 1.0 the width is a fraction of the frame width.

7.3.9 Group ecb-compilation

This group contains settings for the compile window of ECB:

`compilation-buffer-names` [User Option]

Additional buffer names that should be displayed in the compile-window. Buffer names can either be defined as strings or as regexps. If the buffer-name of a buffer matches one of the defined string or regexp then it will be displayed in the compile-window of ECB even if `compilation-buffer-p` says nil for this buffer.

It is not recommended to add the eshell-buffer-names to this list because ECB already handles the eshell-integration as best as possible (see ⟨undefined⟩ [Using eshell], page ⟨undefined⟩).

See also the options `ecb-compilation-major-modes` and `ecb-compilation-predicates`.

`compilation-major-modes` [User Option]

Additional major-mode that should be displayed in the compile-window. All buffers of a major-mode contained in this list are displayed in the compile-window even if `compilation-buffer-p` says nil for such a buffer.

It is not recommended to add `eshell-mode` to this list because ECB already handles the eshell-integration as best as possible (see ⟨undefined⟩ [Using eshell], page ⟨undefined⟩).

`compilation-predicates` [User Option]

Predicates when a buffer should be treated as compilation-buffer. Every element of this list has to be a function or lambda-expression which gets as argument a buffer-object and which has to return not nil when this buffer should be treated as compilation-buffer (even if `compilation-buffer-p` says nil) and therefore be displayed in the compile-window of ECB (if there is any).

In combination with the values of `ecb-compilation-buffer-names` and `ecb-compilation-major-modes` ECB decides when a buffer is displayed in the compile-window.

Default value is the function `comint-check-proc` which returns not nil when the buffer is related to a living process.

`compile-window-height` [User Option]

Height of the persistent compilation-window of ECB. If you want a compilation window shown at the bottom of the ECB-layout then set here the height of it (Default is a height of 5). If you redraw the current layout with `ecb-redraw-layout` then the compilation window (if any) has the height you set here. If the number is less than 1.0 the height is a fraction of the frame height.

If you do not set a persistent compilation window then doing a compilation or displaying temp-buffers (e.g. *Help*-buffers) splits temporally the edit window vertically if the edit window is not splitted already or uses another edit window temporally for compilation output if the edit window is already splitted. This is the recommended value for this option because this is the standard-behavior of Emacs.

Beware: If you set a persistent compilation window then ECB displays all buffers for which `ecb-compilation-buffer-p` returns not nil in that persistent compilation

window. If a buffer which should being displayed there is not displayed there then try to modify the options `ecb-compilation-buffer-names`, `ecb-compilation-major-modes` or `ecb-compilation-predicates` (in this sequence).

See also the options `ecb-compile-window-temporally-enlarge` and `ecb-enlarged-compilation-window-max-height` and also the command `ecb-toggle-compile-window-height`!

ECB offers the functionality of such a persistent compile-window regardless if the special ECB-windows are visible or not (see the command `ecb-toggle-ecb-windows`).

Regardless of the settings you define here: If you have destroyed or changed the ECB-screen-layout by any action you can always go back to this layout with `ecb-redraw-layout`

`compile-window-prevent-shrink-below-height` [User Option]

Allow the compile-window to be shrunken below its height. A non nil value means ECB prevents the compile-window from being shrunken below the threshold of `ecb-compile-window-height` by displaying temp-buffers (e.g. *Help* etc.) or after running compilations or greps. But interactively it is always allowed to shrink it to every height!

If nil then ECB does nothing to prevent being shrunken below the value of `ecb-compile-window-height`.

Default is t.

`compile-window-temporally-enlarge` [User Option]

Let Emacs temporally enlarge the compile-window of the ECB-layout. This option has only an effect if `ecb-compile-window-height` is not nil!

The following values are possible:

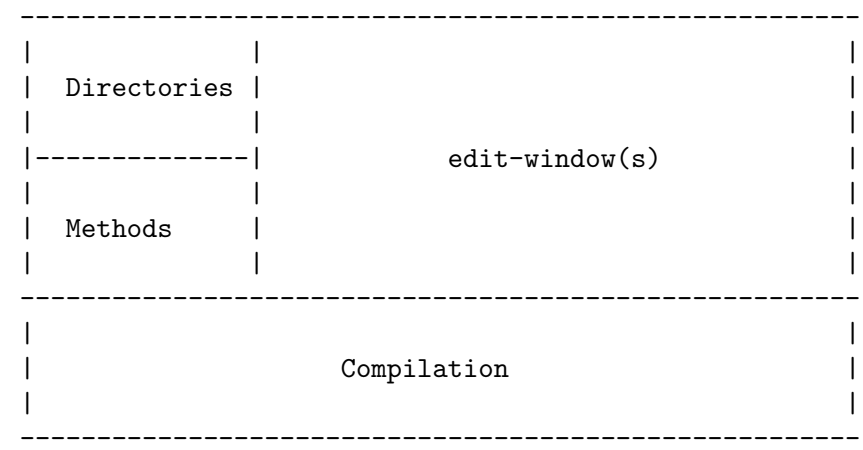
- `after-display`: After displaying a "compilation-buffer" (in the sense of `ecb-compilation-buffer-p`!) in the compile-window of ECB. For the max. height of the enlarged compile-window see the option `ecb-enlarged-compilation-window-max-height`.
- `after-selection`: Selecting the `ecb-compile-window` auto. enlarges it and deselecting (means leaving `ecb-compile-window`) auto. shrinks it. Enlarging and shrinking the `ecb-compile-window` is done with `ecb-toggle-compile-window-height`. See also the documentation of this function!
- `both`: The combination of 'after-display and 'after-selection.
- `nil`: ECB fixes always the height of the `ecb-compile-window` at the value of `ecb-compile-window-height`.

To restore the ECB-layout after such a buffer-enlarge just call `ecb-toggle-compile-window-height` or `ecb-redraw-layout`.

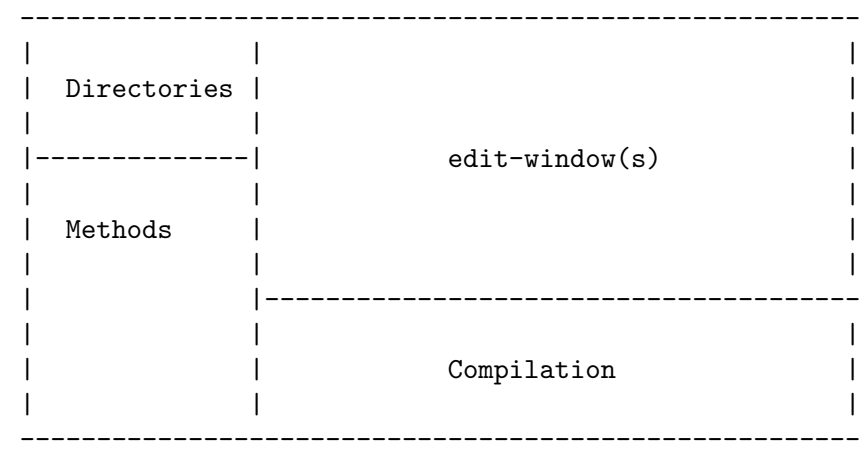
`compile-window-width` [User Option]

Width of the compile-window.

Possible values are `frame` and `edit-window`. With `frame` the compile-window looks like:



With `edit-window` the compile-window looks like:



This option takes only effect if `ecb-compile-window-height` is not nil!

`change-layout-preserves-compwin-state` [User Option]

Changing the layout preserves the state of the compile-window. This is for example useful if the user toggles between several layouts (see `ecb-toggle-layout`) and wants to preserve the hidden-state of the compile-window.

`enlarged-compilation-window-max-height` [User Option]

The max height of the compile-window after enlarging it. The max height of the compilation window after enlarged by `ecb-toggle-compile-window-height`. The following values are allowed:

`best`:

ECB fits the height of the compile-window exactly to the size of its current contents but never shrinks below the value of `ecb-compile-window-height` or enlarges over the half of the frame-height of the ECB-frame. The values of the options `compilation-window-height` and `temp-buffer-max-height` are taken into account dependent of the current `major-mode` of the buffer in the compile-window: If `compilation-mode` then `compilation-window-height` is used otherwise `temp-buffer-max-height`.

`half`:

1/2 the frame-height of the ECB-frame

Any number:

Max height in lines. If the number is less than 1.0 the height is a fraction of the frame height (e.g. 0.33 results in a max-height of 1/3 the frame-height).

`scroll-other-window-scrolls-compile-window` [User Option]
`scroll-other-window` scrolls always the compile-window. For all details about the scroll-behavior of `scroll-other-window` see the advice documentation of `other-window-for-scrolling`.

7.3.10 Group ecb-create-layout

This group contains settings for creating new ECB-layouts:

`create-layout-file` [User Option]
File where all layouts created by `ecb-create-new-layout` are stored.

`ecb-create-layout-frame-height` [User Option]
Frame height of the layout creation frame.

`ecb-create-layout-frame-width` [User Option]
Frame width of the layout creation frame.

7.3.11 Group ecb-face-options

This group contains settings for all faces used in ECB:

`directories-general-face` [User Option]
Basic face for the ECB directories buffer. This defines the basic face the whole directory buffer should displayed with. If the face `ecb-default-general-face` is used then the display of all ECB-tree-buffers can be changed by modifying only the face `ecb-default-general-face`.

Changes take first effect after finishing and reactivating ECB!

`directory-face` [User Option]
Face used for highlighting current directory in the directories buffer. If the face `ecb-default-highlight-face` is used then the display of all ECB-tree-buffers can be changed by modifying only the face `ecb-default-highlight-face`.

Changes take first effect after finishing and reactivating ECB!

`directory-not-accessible-face` [User Option]
Face for not accessible dirs in the directories buffer.

`history-face` [User Option]
Face used for highlighting current history-entry in the history buffer. If the face `ecb-default-highlight-face` is used then the display of all ECB-tree-buffers can be changed by modifying only the face `ecb-default-highlight-face`.

Changes take first effect after finishing and reactivating ECB!

`history-general-face` [User Option]
Basic face for the ECB directory buffer. This defines the basic face the whole history buffer should displayed with. If the face `ecb-default-general-face` is used then the display of all ECB-tree-buffers can be changed by modifying only the face `ecb-default-general-face`.

Changes take first effect after finishing and reactivating ECB!

`method-face` [User Option]
Face used for highlighting current method, class or variable in the methods buffer. If the face `ecb-default-highlight-face` is used then the display of all ECB-tree-buffers can be changed by modifying only the face `ecb-default-highlight-face`.

Changes take first effect after finishing and reactivating ECB!

`method-non-semantic-face` [User Option]
Face used for for displaying tags of sources not supported by semantic.

Changes take first effect after finishing and reactivating ECB!

`methods-general-face` [User Option]
Basic face for the ECB methods buffer. This defines the basic face the whole methods buffer should displayed with. If the face `ecb-default-general-face` is used then the display of all ECB-tree-buffers can be changed by modifying only the face `ecb-default-general-face`.

Changes take first effect after finishing and reactivating ECB!

`source-face` [User Option]
Face used for highlighting current source in the sources buffer. If the face `ecb-default-highlight-face` is used then the display of all ECB-tree-buffers can be changed by modifying only the face `ecb-default-highlight-face`.

Changes take first effect after finishing and reactivating ECB!

`source-in-directories-buffer-face` [User Option]
Face for source files in the directories buffer.

`sources-general-face` [User Option]
Basic face for the ECB sources buffer. This defines the basic face the whole sources buffer should displayed with. If the face `ecb-default-general-face` is used then the display of all ECB-tree-buffers can be changed by modifying only the face `ecb-default-general-face`.

Changes take first effect after finishing and reactivating ECB!

`source-read-only-face` [User Option]
Face for read-only sources.

`tag-header-face` [User Option]
Face used for highlighting the tag header after jumping to it by clicking onto a node in the methods buffer.

7.3.12 Group ecb-faces

This group contains all faces used in ECB:

`ecb-bucket-node-face:`
: Face which can be used for displaying bucket tags in the methods buffer. See also `ecb-bucket-node-display`.

`ecb-default-general-face:`
: Basic face for all ECB tree-buffers. It's recommended to define here the font-family, the font-size, the basic color etc.

  In GNU Emacs 21.X all faces (even the face `ecb-default-highlight-face`) used in the ECB tree-buffers inherit from this face. Therefore the default attributes like font etc. of a face used in a tree-buffer can be very easily changed with face `ecb-default-general-face`.

  With XEmacs there is no inheritance-feature but the options `ecb-directories-general-face`, `ecb-sources-general-face`, `ecb-methods-general-face` and `ecb-history-general-face` offer the choice to use the face `ecb-default-general-face` so also with XEmacs the basic face-settings can be easily changed just by customizing the face `ecb-default-general-face`!

`ecb-default-highlight-face:`
: Define basic face for highlighting the selected node in an ECB tree-buffer.

  In GNU Emacs 21.X all highlighting faces in the ECB tree-buffers inherit from this face. Therefore the default attributes like font etc. of a face used in a tree-buffer for highlighting the current tag can be very easily changed with face `ecb-default-highlight-face`.

  With XEmacs there is no inheritance-feature but the options `ecb-directory-face`, `ecb-source-face`, `ecb-method-face` and `ecb-history-face` offer the choice to use the face `ecb-default-highlight-face` so also with XEmacs the basic face-settings can be easily changed just by customizing the face `ecb-default-highlight-face`!

`ecb-directories-general-face:`
: Basic face for the ECB directories buffer. Its recommended to define here the font-family, the font-size, the basic color etc.

`ecb-directory-face:`
: Define face used for highlighting current directory in the directories buffer.

`ecb-directory-not-accessible-face`
: Define a face for not accessible dirs in the directories buffer.

`ecb-history-face:`
: Define face used for highlighting current history-entry in the history buffer.

`ecb-history-general-face:`
: Basic face for the ECB history buffer. Its recommended to define here the font-family, the font-size, the basic color etc.

`ecb-method-face:`
: Define face used for highlighting current method, class or variable in the methods buffer.

`ecb-methods-general-face:`
: Basic face for the ECB methods buffer. Its recommended to define here the font-family, the font-size, the basic color etc.

`ecb-method-non-semantic-face:`
: Define face used for displaying tags of sources not supported by semantic.

`ecb-mode-line-data-face`
: Define face for the data in the mode-line. See `ecb-mode-line-data`.

`ecb-mode-line-prefix-face`
: Define face for the prefix in the mode-line. See `ecb-mode-line-prefixes`.

`ecb-source-face:`
: Define face used for highlighting current source in the sources buffer.

`ecb-source-in-directories-buffer-face:`
: Define a face for displaying sources in the directories buffer.

`ecb-sources-general-face:`
: Basic face for the ECB sources buffer. Its recommended to define here the font-family, the font-size, the basic color etc.

`ecb-source-read-only-face`
: Define a face for read-only sources

`ecb-tag-header-face:`
: Define face used for highlighting the tag header after jumping to it by clicking onto a node in the methods buffer.

`ecb-tree-guide-line-face:`
: Define face for the guide-lines in the tree-buffers. See the option `ecb-tree-buffer-style` for a explanation of guide-lines.

`ecb-type-tag-class-face:`
: Define face used with option `ecb-type-tag-display`.

`ecb-type-tag-enum-face:`
: Define face used with option `ecb-type-tag-display`.

`ecb-type-tag-group-face:`
: Define face used with option `ecb-type-tag-display`.

`ecb-type-tag-interface-face:`
: Define face used with option `ecb-type-tag-display`.

`ecb-type-tag-struct-face:`
: Define face used with option `ecb-type-tag-display`.

`ecb-type-tag-typedef-face:`
: Define face used with option `ecb-type-tag-display`.

`ecb-type-tag-union-face:`
: Define face used with option `ecb-type-tag-display`.

`ecb-mode-line-win-nr-face`
: Define face for the window-number in the mode-line. See `ecb-mode-line-display-window-number`.

Just call `customize-face <face-name>` to customize these faces for your personal taste. Or customize the related option in the group ⟨undefined⟩ [ecb-face-options], page ⟨undefined⟩.

7.3.13 Group ecb-download

This group contains settings for downloading and installing a new ECB from within ECB:

`download-delete-archive` [User Option]

Should the downloaded archive be deleted after successful installation or after failure during the installation-process. Possible values are:

- `only-after-success`: Archive is only deleted after successful installation but not if a failure occurs during the installation process.
- `always`: Archive is also deleted if an error occurs.
- `nil`: Archive will never be deleted.

`download-install-parent-dir` [User Option]

Parent directory where downloaded packages are installed.

ECB installs a downloaded package in this directory, i.e. the downloaded archive X.tar.gz will be extracted in this directory so afterwards this directory contains a new subdirectory X which contains the downloaded package.

This directory must be write-able!

`download-package-version-type` [User Option]

Version type ECB is allowed to download for upgrading.

If you want to upgrade to a newer ECB-version via `ecb-download-ecb` or if you must upgrade to newer semantic- eieio- and/or speedbar-versions (because ECB requires these newer versions) then this option specifies which version-types are allowed. ECB checks on the download-sites of ECB/semantic/eieio/speedbar which versions are currently available and then downloads always the latest version matching the specified type:

- 2: Get the newest version of all stable versions available.
- 1: Get the newest version of all stable and beta versions available.
- 0: Get the newest version of all stable, beta and alpha versions available.
- -1: Ask before downloading in the minibuffer for a version (TAB-completion of all available versions is possible).

So, 2 means stable, 1 means stable and betas, 0 means stable, betas and alphas and -1 means ask the user for a version.

Per default stable and beta-versions are allowed (value 1).

But all versions must match the restrictions of the specified min- and max-versions of the required packages. For this see the file README!

`download-url` [User Option]

URL where download-able ECB-versions are located. The ECB-archive-file (e.g. ecb-1.70.tar.gz) will be appended to this URL and `ecb-download-ecb` will try to download this archive.

Note: Normally this URL should never change but who knows...

`gzip-setup` [User Option]
Configuration for the gzip-utility. For a description about the possible settings see `ecb-wget-setup`.

`tar-setup` [User Option]
Configuration for the tar-utility. For a description about the possible settings see `ecb-wget-setup`.

`wget-setup` [User Option]
Configuration for the wget-utility. Value is a cons-cell where:

- car is the name of the wget-executable - if the executable can not be found in the *PATH* then it must be a full path.
- cdr is the path type of the file-arguments of this binary. Possible values are `cygwin`, `windows` and `other` whereas the latter one is used for all Unix, Linux, Mac OS etc... If `cygwin` is set then the cygpath-utility must be in the PATH!

7.3.14 Group ecb-help

This group contains settings for the ECB online help:

`help-html-path` [User Option]
Path where the ECB online help in HTML format resides. This must be the location of the '`ecb.html`' which comes with the ECB distribution. If is installed by unpacking the archive available on the ECB website then this is the subdir `ecb-help-html-subdir` of the installation directory of ECB. If it is installed as XEmacs-package (e.g. via the package manager of XEmacs) then this is probably either the directory "../../html/" or "../../etc/ecb/html/" (both relative to the Elisp directory of ECB).

The path can either be an absolute path or a path relative to the directory where the Elisp files of ECB are.

Normally there should be no need to change this option!

`help-info-path` [User Option]
Path where the ECB online help in info format resides. This must be the location of the '`ecb.info`' which comes with the ECB distribution. If is installed by unpacking the archive available on the ECB website then this is the subdir `ecb-help-info-subdir` of the installation directory of ECB. If it is installed as XEmacs-package (e.g. via the package manager of XEmacs) then this is probably the directory "../../info/" (relative to the Elisp directory of ECB).

The path can either be an absolute path or a path relative to the directory where the Elisp files of ECB are.

Normally there should be no need to change this option!

`show-help-format` [User Option]
The format `ecb-show-help` shows its online help. Allowed values are 'info (for the Info format) and 'html (for HTML format). If the value is 'html then `browse-url-browser-function` says which browser is used.

Note: If you got ECB as a standard XEmacs-package maybe the HTML-online-documentation is not included.

7.3.15 Group ecb-eshell

This group contains settings for eshell integration within the ECB:

`eshell-auto-activate` [User Option]
Startup the eshell and display it in the compile-window. If current layout does not display a compile-window (see `ecb-compile-window-height`) then nothing is done.

`eshell-enlarge-when-eshell` [User Option]
Enlarge the compile-window if it is selected by `eshell`. This takes only effect if the command `eshell` is called!

`eshell-fit-window-to-command-output` [User Option]
Fit the compile-window after an eshell-command to the output. This is done by the function `ecb-eshell-fit-window-to-output` which is added to `eshell-post-command-hook` ie. which is running autom. after each eshell-command.

`eshell-synchronize` [User Option]
Synchronize eshell with the default-directory of current source-buffer. The synchronization is done by `ecb-eshell-current-buffer-sync` which can be called interactively but normally it is called autom. by the `ecb-current-buffer-sync-hook-internal`.

7.3.16 Group ecb-speedbar

`speedbar-before-activate-hook` [User Option]
Hook running directly before ECB activates the integrated speedbar.

For example this hook can be used to change the expansion-mode of the integrated speedbar via `speedbar-change-initial-expansion-list`. Example: (speedbar-change-initial-expansion-list “buffers”).

7.3.17 Group ecb-non-semantic

This group contains settings for parsing and displaying non-semantic files:

`auto-save-before-etags-methods-rebuild` [User Option]
Automatic saving of current buffer before rebuilding its methods.

This option is only relevant for sources which are supported and parsed by etags (see `ecb-process-non-semantic-files`). Because etags is an external tool a source-buffer can only be reparsed if the buffer is saved to disk. So the command `ecb-rebuild-methods-buffer` checks for sources which are not supported by semantic or imenu if either this option is t or if the major-mode of the source-buffer is contained in this list: In both cases ECB saves the current source-buffer before it re-runs etags for reparsing the source. If nil or if the major-mode is not contained then no automatic saving will be done!

For all source supported by semantic or by imenu this option takes no effect.

`non-semantic-exclude-modes` [User Option]
Exclude modes from parsing with imenu or etags. Per default, ECB tries to parse all file-types not supported by semantic with imenu or etags or some other method (for

details see the option `ecb-non-semantic-parsing-function`). If a file-type can not be parsed by semantic, imenu or etags than this simply results in an empty method-buffer for this file. But nevertheless you will get a message "Sorry, no support for a file of that extension" which comes from the speedbar-library and can not switched off. Therefore if a `major-mode` is known as not parse-able by semantic, imenu or etags it can be added to this option and then it will be excluded from being tried to parsed.

`non-semantic-methods-initial-expand` [User Option]
Initially expand all tags for not by semantic supported sources. This option can be customized on a major-mode basis, i.e. if a `major-mode` is contained in this option then all tags for this modes will be initially expanded - otherwise not.

`non-semantic-parsing-function` [User Option]
Define mode-dependent parsing functions for non-semantic files. This is an alist where the car is a major-mode symbol and the cdr is a function-symbol of a function which should be used for parsing a non-semantic buffer, i.h. a buffer for which no semantic grammar exists. Such a function gets one argument - the filename of current buffer - and has to generate and return a tag/tag list which is understandable by `speedbar-insert-generic-list`. speedbar has already included two functions `speedbar-fetch-dynamic-imenu` and `speedbar-fetch-dynamic-etags` which can be used for parsing buffers with imenu rsp. etags.

This option takes only effect if `ecb-process-non-semantic-files` is not nil: Then ECB checks for non-semantic buffers if current `major-mode` is contained in this option and if yes, then the specified parsing function is called; if not then the cars of the elements of `speedbar-dynamic-tags-function-list` are called in that sequence they are listed in this variable. See option `speedbar-dynamic-tags-function-list` for further details.

In most cases imenu-parsing is preferable over etags-parsing because imenu operates on Emacs-buffers and needs no external tool and therefore parsing works also if current contents of a buffer are not saved to disk. But maybe sometimes etags may return better parsing results

IMPORTANT: if imenu-parsing should be used then the option `speedbar-use-imenu-flag` must be set to not nil!

`process-non-semantic-files` [User Option]
Display content of non-semantic-files in the ECB-methods-buffer. See also `ecb-non-semantic-parsing-function`.

`rebuild-non-semantic-methods-before-hook` [User Option]
Hook running at beginning of the function `ecb-rebuild-methods-buffer-for-non-semantic`. This function is always called by the command `ecb-rebuild-methods-buffer` for not semantic supported source-types.

Every function of this hook gets one argument: The complete filename of the current source-buffer in the edit-window. The Method-buffer is only rebuild by `ecb-rebuild-methods-buffer-for-non-semantic` if either the hook contains no function (the default) or if no function of this hook returns nil! See `run-hook-with-args-until-failure` for description how these function are processed.

7.3.18 Group ecb-winman

This group contains settings for supporting several window-managers:

`winman-escreen-number` [User Option]
Number of the escreen which is reserved for ECB. If you go to the escreen with this number you go always to the escreen with activated ECB. All other escreen-numbers are escreens with deactivated ECB!

`winman-winring-name` [User Option]
Name of the winring-window-configuration reserved for ECB. If you go to the window-configuration with this name you go always to the window-configuration with activated ECB. All other window-configuration are configurations with deactivated ECB!

7.3.19 Group ecb-mode-line

This group contains settings for the modelines of the ECB-tree-buffers:

`mode-line-data` [User Option]
Data shown in the modelines of the special ECB-buffers. Everey element of this list is a cons-cell where the car is used to define a buffer-name and the cdr to define the modeline-data for that buffer. For details about how to defining a buffer-name see `ecb-mode-line-prefixes` - its completely the same.

The cdr is the data for ths modeline and can either be the symbol `sel-dir` or `sel-source` whereas the former one displays the current selected directory as modeline-data and the latter one the current selected source-file (without path).

In addition to these two predefined values for every special ECB-buffer a plain string (which is displayed) or a function can be specified which gets three args (name of the buffer, current selected directory and current selected source-file) and must return a string which will be displayed in the modeline (or nil if no data should be displayed). Such a function can add the text-property `help-echo` to the result-string. Then this help-string will be displayed when the user moves the mouse over this section of the modeline.

If a special ECB-buffer should not display special data in its modeline then this buffer-name should either not being added to this option or added with "No data" (= nil as cdr).

The whole modeline of the special ECB-buffer consists of the prefix of `ecb-mode-line-prefixes` and the data of `ecb-mode-line-data` - eventually prepended by the window-number, see `ecb-mode-line-display-window-number`.

`mode-line-data-face` [User Option]
Face used for the data in the mode-line. See `ecb-mode-line-data`. For XEmacs the face should inherit from the face `modeline` (see `set-face-parent`)!

`mode-line-display-window-number` [User Option]
Display in the modeline of every special ECB-window the window-number. The left-top-most window in a frame has the window-number 0 and all other windows are numbered with increasing numbers in the sequence, functions like `other-window` or `next-window` would walk through the frame.

This can be used to jump to windows by number with commands like:

```
(defun my-switch-to-window-number (number)
  ``Switch to the nth window''
  (interactive ``P'')
  (if (integerp number)
      (select-window (nth number (window-list)))))
```

Currently this feature is only available for GNU Emacs 21.X, because neither GNU Emacs < 21 nor XEmacs can evaluate dynamically forms in the mode-line.

`mode-line-prefixes` [User Option]

Prefixes shown in the modelines of the special ECB-buffers. The displayed prefix then looks like: "[<PREFIX>[:]]", means if a prefix is defined for an special ECB-buffer then a single space is prepended and if there is additional text to display (e.g. the current directory in the sources buffer, see `ecb-mode-line-data`) then also the string ": " is appended.

Everey element of this list is a cons-cell where the car is used to define a buffer-name and the cdr to define the modeline-prefix for that buffer.

The buffer-name can either be defined as plain string or with a symbol which contains the buffer-name as value. The latter one is recommended to define a prefix for one of the builtin ECB-tree-buffers because then simply the related option-symbol can be used. To add a prefix for the builtin directories tree-buffer just set the symbol `ecb-directories-buffer-name` as car.

The cdr is the prefix for a buffer and can either be a string which used as it is or a function-symbol which is called with three argument (the buffer-name, the current selected directory and the current selected source-file) and must return either nil (for no prefix) or a string which is then used a prefix. Such a function can add the text-property `help-echo` to the result-string. Then this help-string will be displayed when the user moves the mouse over this section of the modeline.

If a special ECB-buffer should not have a prefix in its modeline then this buffer-name should either not being added to this option or added with "No prefix" (= nil as cdr).

`mode-line-prefix-face` [User Option]

Face used for the prefix in the mode-line. See `ecb-mode-line-prefixes`. For XEmacs the face should inherit from the face `modeline` (see `set-face-parent`)!

`mode-line-win-nr-face` [User Option]

Face used for the window-number in the mode-line. See `ecb-mode-line-display-window-number`. For XEmacs the face should inherit from the face `modeline` (see `set-face-parent`)!

7.3.20 Group ecb-version-control

This group contains settings for the version-control-support of ECB:

`vc-directory-exclude-regexps` [User Option]

Which directories should be excluded from VC-state-check. If a directory matches any of the regexps of this option the VC-state of its sources will not be checked - This option takes only effect if `ecb-vc-enable-support` is not nil.

`vc-enable-support` [User Option]

Enable support for version-control (VC) systems. If on then in the directories-buffer (if the value of the option `ecb-show-sources-in-directories-buffer` is on for current layout), the sources-buffer and the history-buffer all file-items are displayed with an appropriate icon in front of the item-name to indicate the VC-state of this item. If off then no version-control-state checking is done.

Because this check can be take some time if files are managed by a not local Version-control-server ECB performs this check stealthy (see `ecb-stealthy-tasks-delay`) so normally there should no performance-decrease or additional waiting-time for the user. But to get sure this option offers three choices: `t`, `unless-remote` and `nil`. See the option `ecb-prescan-directories-for-emptyness` for an explanation for these three choices.

The option `ecb-vc-directory-exclude-regexps` offers are more fine granularity to exclude the sources of certain directories from the VC-state-check.

See `ecb-vc-supported-backends` and `ecb-vc-state-mapping` how to customize the VC-support itself.

`vc-state-mapping` [User Option]

Mapping between VC-states from the backends and ECB-known VC-states. ECB understands the following state-values:

`up-to-date`
: The working file is unmodified with respect to the latest version on the current branch, and not locked.

`edited`
: The working file has been edited by the user. If locking is used for the file, this state means that the current version is locked by the calling user.

`needs-patch`
: The file has not been edited by the user, but there is a more recent version on the current branch stored in the master file.

`needs-merge`
: The file has been edited by the user, and there is also a more recent version on the current branch stored in the master file. This state can only occur if locking is not used for the file.

`added`
: The working file has already been added/registered to the VC-system but not yet commited.

`unlocked-changes`
: The current version of the working file is not locked, but the working file has been changed with respect to that version. This state can only occur for files with locking; it represents an erroneous condition that should be resolved by the user.

`ignored`
: The version-control-system ignores this file (e.g. because included in a .cvsignore-file in case of CVS).

`unknown`
: The state of the file can not be retrieved; probably the file is not under a version-control-system.

All state-values a check-vc-state-function of `ecb-vc-supported-backends` can return must have a mapping to one of the ECB-state-values listed above. If for a certain backend-VC-state no mapping can be found then per default 'edited is assumed!

The default value of this option maps already the possible returned state-values of `vc-state` and `vc-recompute-state` (both GNU Emacs) and `vc-cvs-status` (Xemacs) to the ECB-VC-state-values.

`vc-supported-backends` [User Option]

Define how to to identify the VC-backend and how to check the state. The value of this option is a list containing cons-cells where the car is a function which is called to identify the VC-backend for a DIRECTORY and the cdr is a function which is called to check the VC-state of the FILEs contained in DIRECTORY.

Identify-backend-function: It gets a full directory-name as argument - always without ending slash (rsp. backslash for native Windows-XEmacs) - and has to return a unique symbol for the VC-backend which manages that directory (e.g. 'CVS for the CVS-system or 'RCS for the RCS-system) or nil if the file is not managed by a version-control-system.

Check-vc-state-function: It gets a full filename (ie. incl. the complete directory-part) and has to return a symbol which indicates the VC-state of that file. The possible returned values of such a check-vc-state-function have to be mapped with `ecb-vc-state-mapping` to the allowed ECB-VC-state values.

ECB runs for a certain DIRECTORY all identify-backend-functions in that order they are listed in this option. For the first which returns a value unequal nil the associated check-state-function is used to retrieve the VC-state of all sourcefiles in that DIRECTORY.

There is no need for the identify-backend-function or the check-vc-state-function to cache any state because ECB automatically caches internally all necessary informations for directories and files for best possible performance.

To prepend ECB from checking the VC-state for any file set `ecb-vc-enable-support` to nil.

Default value for GNU Emacs: Support for CVS, RCS, SCCS and Subversion (for the later one the most recent version of the VC-package incl. the vc-svn library is needed) is added per default. To identify the VC-backend the functions `ecb-vc-managed-by-CVS`, `ecb-vc-managed-by-RCS` rsp. `ecb-vc-managed-by-SCCS` rsp. `ecb-vc-managed-by-SVN` are used. For all three backends the function `ecb-vc-state` of the VC-package is used.

Default value for XEmacs: XEmacs contains only a quite outdated VC-package, especially there is no backend-independent check-vc-state-function available (like `vc-state` for GNU Emacs). Only for CVS a check-vc-state-function is available: `vc-cvs-status`. Therefore ECB adds per default only support for CVS and uses `ecb-vc-managed-by-CVS` rsp. `vc-cvs-status`.

Example for GNU Emacs: If `vc-recompute-state` (to get real state-values not only heuristic ones) should be used to check the state for CVS-managed files and `vc-state` for all other backends then an element (ecb-vc-dir-managed-by-CVS . vc-recompute-state) should be added at the beginning of this option.

`vc-xemacs-exclude-remote-cvs-repository` [User Option]

Exclude directories with a remote cvs-repository from VC-check. This option takes only effect for XEmacs and is needed cause of the outdated VC-package of XEmacs which offers no heuristic state-checking and also no option `vc-cvs-stay-local`. So this option takes only effect if `vc-cvs-stay-local` is not avaiable. In this case ECB treats directories which are managed by CVS but have a remote repository as if the directory would be not managed by CVS (so the files are not checked for their VC-state). This si done to avoid blocking XEmacs when running full cvs-commands (e.g. "cvs status") over the net.

Note: When ECB can find the option `vc-cvs-stay-local` then this option will automatically take no effect regardless which Emacs-version is used.

8 Submitting a problem report

If you run into problems with ECB you should first take a look into

- ⟨undefined⟩ [FAQ], page ⟨undefined⟩ or
- ⟨undefined⟩ [Conflicts and bugs], page ⟨undefined⟩ or
- ⟨undefined⟩ [Tips and tricks], page ⟨undefined⟩ or
- the appropriate section of this online-manual.

If your problem(s) still remain you can/should send a problem report to the ECB mailing list `ecb-list@lists.sourceforge.net`. ECB offers you a command which does all necessary for you to send a problem report. Just call `ecb-submit-problem-report`! Please read the documentation of this command, see ⟨undefined⟩ [Interactive ECB commands], page ⟨undefined⟩.

IMPORTANT: Cause of extra appearance of SPAM in the mailing-lists, SourceForge has changed its policy: Now it is only possible to post to the mailing-list for users who have subscribed this mailing-list. So please be aware you will not be able to send comments, bug reports and improvement suggestions before you have subscribed the ECB-mailing-list. See the section "Mailing-list" at the ECB-website at `http://ecb.sourceforge.net` how to do this.

If you think there are problems concerning parsing-results for certain sources supported by semantic then you should call `ecb-dump-semantic-toplevel` for the problematic source-buffer **BEFORE** you call `ecb-submit-problem-report` because this “dump”-command generates for the current-buffer a new buffer with name “*ecb-tag-dump*” which contains all semantic-tags for this source. The contents of this “*ecb-tag-dump*” will then autom. be added to the problem-report generated by `ecb-submit-problem-report`!

This command creates a problem-report buffer for you. After that you get a menu “Mail” (dependent on the mail-package used, the menu can have a different name) with commands to send the problem report. But for this the variable `mail-user-agent` must be configured right for your system. If you cant get working this mechanism you can simply copy the whole problem-report buffer after filling it out and sending it with your standard mail-client to `ecb-list@lists.sourceforge.net`!

Please read also the documentation of the option `ecb-debug-mode` and switch on the debug mode of ECB (also available in the Help-menu of ECB!) before submitting a problem-report!

9 Upgrading and downloading packages

This chapter describes all aspects of upgrading to a newer version of ECB.

The first section describes how to download and install a new package from the web, where "package" means either ECB itself or the required libraries semantic, eieio and speedbar.

After installing a new ECB-version ECB checks if the values of the customized ECB-options are still compatible. If not ECB does some smart things. This is the topic of the second section.

9.1 Downloading new versions of ECB and/or required packages

ECB offers the possibility to upgrade to newer versions direct from the ECB-website. This can be done if the following requirements are satisfied:

1. A connection to the web is available
2. The tools "wget", "tar" and "gzip" are installed

   With Unix-systems these tools are in the standard-distribution. If you are running any Microsoft Windows system then you need cygwin[1] which offers these tools too. On every system these tools must reside in the `PATH` environment-variable!

   If you are behind a firewall and you have to use a proxy you maybe need the following wget-configuration in your file '`~/.wgetrc`':

   ```
   # Define your proxies (where 8080 and 8081 are examples
   # for the port-numbers)
   http_proxy = http://your.proxy.com:8080
   ftp_proxy  = http://your.ftpproxy.com:8081

   # If you do not want to use proxy at all, set this to off.
   use_proxy = on
   ```

If these requirements are satisfied you can download and install both ECB itself and also the required versions of semantic, eieio and speedbar:

- Download a new ECB-version with `ecb-download-ecb`:

  A description for this command you will find in ⟨undefined⟩ [Interactive ECB commands], page ⟨undefined⟩. Check also the options of the customize-group 'ecb-download' (see ⟨undefined⟩ [ecb-download], page ⟨undefined⟩).

- Download and install of required packages:

  ECB checks at load-time if the packages semantic, eieio and speedbar are at least installed and at start-time if the required versions of semantic, eieio and speedbar (see '`README`') are installed and loaded into Emacs. If not you will be asked if you want auto. download and install them. If you confirm this then ECB does the following:

  1. Checking which versions are available at the download-site of the required packages. With the option `ecb-download-package-version-type` you can specify

[1] cygwin is available at `http://cygwin.com/`

which type of versions (only stable, stable and betas or stable, betas and alphas) you allow to download and install. This option offers also the choice of asking you for a certain version. Depending on this setting ECB either ask you which version it should download and install or chooses autom. the newest version available which is matching both its min-max-requirements and the setting in `ecb-download-package-version-type`.

NOTE: Currently there are only beta-versions of speedbar available therefore this option has to be set to 1 (allow stable and beta versions). But the speedbar beta-versions are very stable!

2. Downloading and installing the right version (see 1.) of the required packages. ECB downloads and installs the new package versions in subdirectories of `ecb-download-install-parent-dir`.

If both of these actions succeed then you will get a message-buffer which tells you something like:

```
------------------------------------------------------------------
Current state of the required packages semantic, eieio, speedbar:

- semantic author-version must be [1.4, 1.4.9]:
  Installed in /usr/local/lib/site-lisp/semantic-1.4

- eieio author-version must be [0.17, 0.17.9]:
  Correct version already loaded!

- speedbar author-version must be [0.14beta1, 0.15.9]:
  Correct version already loaded!

After adding the new directory to your load-path and then
restarting Emacs the new package(s) can be activated.
------------------------------------------------------------------
```

Remark 1: "P author-version must be [x y]" means, that ECB requires package P in a version-number >= x and <= y.

Remark 2: By setting the option `ecb-version-check` to `nil` you can prevent ECB from checking correct versions of semantic, eieio and speedbar but it's strongly recommended not to do this!

9.2 Automatic upgrading of options

9.2.1 User interface for option-upgrading

There are two interactive commands (see ⟨undefined⟩ [Interactive ECB commands], page ⟨undefined⟩):

– `ecb-upgrade-options`: Does all necessary beginning with a incompatibility-check for all options, upgrading/resetting incompatible options and ending with the display of all upgraded or reset options.

– `ecb-display-upgraded-options`: Displays an information-buffer which options have been upgraded or reset. Offers two buttons where the user can decide if the upgraded options should also being saved by ECB for future settings or if the buffer should be killed.

If the option `ecb-auto-compatibility-check` has a non-nil value (which is the default) then ECB does all this stuff automatically at startup. This is very recommended!

If you are interested in some background information, read ⟨undefined⟩ [Background information], page ⟨undefined⟩!

9.2.2 Background information

Big packages like ECB will be enhanced and developed continuously so sometimes a new version must be released. Such packages offer in general a lot of customizable options so probably some of these options change the type or are renamed because the old type and/or name of the option makes no sense in the new release.

Especially options which have changed the type of their value are now a problem for the user which want to upgrade to the latest ECB-version: If the user has saved a certain value for option X in its file '`.emacs`' but the type of this saved value doesn't match the new defined type in the defcustom-form after an ECB-upgrade then there can occur serious problems like ECB can not be started anymore or even Emacs can not be started without errors.

Until now there was only one way to fix these problems: The user must manually edit his file '`.emacs`' and remove all entries for options which have now another type. After this and after restarting Emacs the new default-values of the type-changed options in the new ECB-release are active and the user can go on using Emacs and ECB. But this approach to fix the incompatible-option-problem has two serious drawbacks:

1. The user must manually edit the customize-section in his file '`.emacs`'. This should normally not be done and if then only by old-handed Emacs-users.
2. The customized value of the option X in the old-release (with the old type) is lost because after removing the related entry from the file '`.emacs`' the new default-value is active, so the user must re-customize the option X.

OK, this is one half of the option-upgrade-problem but a new ECB-release can also rename a option from name X to name Y because the new name Y makes much more sense and/or is more mnemonic. If only the name has changed but not the type this is not a serious problem like above but also annoying because the customized value of the old-option X takes no effect in the new release but instead the default-value of the new-option Y is now active. But nevertheless this problem has the drawback number 2 (see above).

The last category of upgrade-problems is a renamed option which has also changed its type.

ECB has a solution for all these problems:

- It checks all customized values of all ECB-options if they are still type-compatible. If not then it tries to upgrade the old-value to the new value-type and if this is not possible then it resets the option to the new default value and offers then to store it via customize in the .emacs-file (or in any file which is used for customized options). But ECB does not touch any customization-file without asking the user!

- It offers a special constant `ecb-upgradable-option-alist` which allows the ECB-maintainers to define special transformings for renamed options so even the value of an old-option X can be savely transformed to the new-option Y and the old setting is not lost.

All these checks and transformings are done at beginning of activating ECB - if the option `ecb-auto-compatibility-check` is not nil. If ECB has recognized incompatible or renamed options it does its upgrading/reseting-job so all ECB-options have correct types so ECB can start correct. After ECB is started it displays a list of all upgraded or reseted option with their old and new values.

10 Tips and tricks

This chapter contains some tips and tricks how to deal best with some situations.

10.1 Changing faces in the ECB tree-buffers

There are two basic faces:

- `ecb-default-general-face`: Basic face for displaying an ECB-tree-buffer.

  Its recommended to define the font-family, the font-size, the basic color etc. with this face.

  In GNU Emacs 21.X all faces (even the face `ecb-default-highlight-face`) used in the ECB tree-buffers inherit from this face. Therefore the default attributes like font etc. of a face used in a tree-buffer can be very easily changed with face `ecb-default-general-face`.

  With XEmacs there is no inheritance-feature but the options `ecb-directories-general-face`, `ecb-sources-general-face`, `ecb-methods-general-face` and `ecb-history-general-face` offer the choice to use the face `ecb-default-general-face` so also with XEmacs the basic face-settings can be easily changed just by customizing the face `ecb-default-general-face`.

- `ecb-default-highlight-face`: Basic face for highlighting the current node in an ECB-tree-buffer.

  In GNU Emacs 21.X all highlighting faces used in the ECB tree-buffers inherit from this face. Therefore the default attributes like font etc. of a highlighting face used in a tree-buffer can be very easily changed with face `ecb-default-highlight-face`.

  With XEmacs there is no inheritance-feature but the options `ecb-directory-face`, `ecb-source-face`, `ecb-method-face` and `ecb-history-face` offer the choice to use the face `ecb-default-highlight-face` so also with XEmacs the basic face-settings can be easily changed just by customizing the face `ecb-default-highlight-face`.

With these faces you can change the basic attributes easily and fast for ALL ECB-tree-buffers. But you are also able to display each ECB-tree-buffer with different faces, see the different options for every tree-buffer mentioned above.

Please note (only for XEmacs users): Cause of the lack of the font-inheritance feature using ONE other font for the ECB-methods buffer can NOT be achieved just by setting `ecb-methods-general-face` to `ecb-default-general-face` and changing the font of this default face. In addition you have to set the same font also for the face `ecb-bucket-node-face` like in the following example:

```
(defconst my-ecb-font
          "-outline-Courier-normal-normal-13-97-96-96-c-*-iso8859-1")
(set-face-font 'ecb-default-general-face my-ecb-font)
(set-face-font 'ecb-bucket-node-face my-ecb-font)
```

This code sets the new defined font `my-ecb-font` as font for all[1] ECB-tree-buffers (incl. the methods buffer).

[1] Of course `ecb-directories-general-face`, `ecb-sources-general-face`, `ecb-methods-general-face` and `ecb-history-general-face` must be set to `ecb-default-general-face`!

10.2 Working with small screens

If your screen is very small so you need every square-centimeter for displaying the buffer which you want to edit, ECB offers you a special layouts, where only the ECB-methods buffer is displayed on top or on left-side. Here comes what you should/can do to work best with ECB in such a situation:

- First customize your ECB:
  1. Customize `ecb-layout-name` to layout-name "top2" (on top) or "left9" (on left-side)
  2. Ensure that `ecb-compile-window-height` is nil.
  3. Optional: Adjust the `ecb-windows-height` rsp. `ecb-windows-width`.
  4. Save your changes.
- To edit your buffers: Call `ecb-toggle-ecb-windows` (also available via the menu "ECB" and by *C-c . lw*) or `ecb-hide-ecb-windows` to hide the ECB-method buffer so you get all the place of your screen for editing.
- To browse and select functions: Call `ecb-toggle-ecb-windows` or `ecb-show-ecb-windows` to make the ECB-method buffer visible if not already. If you want select a method/variable with the keyboard instead with the mouse you should read the section ⟨undefined⟩ [Using the keyboard], page ⟨undefined⟩ in this online help!

The possibility of hiding temporally the ECB windows like described above is also useful for all the other layouts.

10.3 Working with big screens

ECB offers a layout type "left-right" with special ECB-tree-windows on the left and right side of the edit-area. The layouts "leftright1" and "leftright2"are examples for this layout type. See ⟨undefined⟩ [Creating a new ECB-layout], page ⟨undefined⟩ and ⟨undefined⟩ [The layout-engine], page ⟨undefined⟩ for details about how to create or program more layouts of this type.

Such a layout is eventually the best choice for big screens because the several ECB-tree-windows are bigger and can display more informations without scrolling.

With such a layout it can could be senseful to reduce the value of the option `ecb-windows-width` compared to layouts of type left or right. A value of max. 0.25 should be enough.

10.4 Simulating speedbar without an extra frame

You can simulate a speedbar-like layout within ONE frame by doing the following:

1. Customize `ecb-layout-name` to layout name "left9", "left12", "left13" or "left14" dependent to what you like.
2. Optional: Ensure that `ecb-compile-window-height` is nil.
3. Optional: Adjust the `ecb-windows-width`.
4. Optional: Customize `ecb-toggle-layout-sequence` and toggle very fast between several layouts by `ecb-toggle-layout`. See the doc-strings!

5. Optional: Customize `ecb-show-sources-in-directories-buffer` to not nil if the chosen layout (see 1. and 4.) contains a directories-tree-buffer.
6. Save your changes.

But not only simulating speedbar is possible but also full integrating it into the ECB and the ECB-frame, See ⟨undefined⟩ [Integrating speedbar], page ⟨undefined⟩.

10.5 Integrating speedbar in the ECB-frame

It is very easy to integrate speedbar into ECB. There are two different ways to do this:

1. You can either use speedbar in the directories-, sources- or methods-window of ECB instead of the built-in directory-, sources- or methods-browser of ECB. This can be done via the option `ecb-use-speedbar-instead-native-tree-buffer`.
2. Or you can integrate an extra speedbar-window into a layout independent of the existence of a directory-, sources- or methods-window. For this you can either use the built-in layout "left-dir-plus-speedbar" or you have to create your own layout interactively with the command `ecb-create-new-layout`. This way is not described in more details because there is nothing more to describe - just create your layout.

In general integrating speedbar into the ECB-frame makes sense for people...

- ...who like the speedbar way of handling directories and source-files but also like the ECB-way of displaying the buffer-contents (like methods and variables in a source-file). This people should use the option `ecb-use-speedbar-instead-native-tree-buffer` and set it to `dir`.
- ...who like the speedbar way of browsing things like directories, files, file-contents etc. but who dislike the extra speedbar-frame.

Note: It is not necessary to integrate speedbar if you only want parsing sources not supported by semantic. From version 1.94 on ECB supports native parsing and displaying of such sources (see ⟨undefined⟩ [Non-semantic sources], page ⟨undefined⟩)!

Regardless the group you belong, with the speedbar-integration feature of ECB you can combine both worlds, the speedbar- and the ECB-world:

1. Choose a layout which either contains a directories- or a sources-window but not both of them[2].

   Because speedbar has also display-modes for buffers and info-nodes and some other useful things (which can be changed by the speedbar-command `speedbar-change-initial-expansion-list` we recommend layouts like "left15" or "leftright3" for using with speedbar.
2. Set the option `ecb-use-speedbar-instead-native-tree-buffer` to not nil. After this the chosen window of ECB will contain a full featured speedbar (the only difference to standard speedbar is not residing in an extra frame).

Note: If you belong to the first group of people (s.a.) a similar effect and usability is available by setting `ecb-use-speedbar-instead-native-tree-buffer` to nil and setting

[2] Only one of them is needed if you use speedbar because speedbar displays directories and sources in one window. But if you like wasting space then you can also use a layout with both windows...

`ecb-show-sources-in-directories-buffer` to not nil, because this combination displays also directories and sources in one window.

So with the option `ecb-use-speedbar-instead-native-tree-buffer` you have the choice which way of displaying and handling "things" (directories, sources, methods...) you want (the speedbar- or the ECB-way).

During speedbar is running within ECB (i.e. `ecb-use-speedbar-instead-native-tree-buffer` is not nil) the speedbar-command `speedbar` is disabled and the speedbar-command `speedbar-get-focus` switches between the speedbar-window and the edit-window[3].

IMPORTANT: ECB can only integrate speedbar-versions >= 0.14beta1! If you use lower versions of speedbar `ecb-use-speedbar-instead-native-tree-buffer` has no effect.

10.6 Working with large directories

If `ecb-source-path` contains directories with many files and subdirs, especially if these directories are mounted net-drives ("many" means here something > 500, dependent on the speed of the net-connection and the machine), actualizing the sources- and/or directories-buffer of ECB (if displayed in current layout!) can slow down dramatically. If this is a problem the contents of certain directories and also the contents of the sources-buffer can be cached which increases the speed a lot. See the option `ecb-cache-directory-contents`.

IMPORTANT: The full speed-advantage of this cache-mechanism is only available if `ecb-show-sources-in-directories-buffer` is `nil`, i.e. sources of a directory are displayed in the ECB-sources-window. The reason is that only with a sources window the tree-buffer contents for the sources can be cached (i.e. the buffer-content of the ECB-sources-window) whereas with sources displayed in the directories buffer only the disk-contents of a directory are cached - which increases speed too but not so much as with sources displayed in the extra window ECB-sources.

The cache of a directory can be only refreshed by a POWER-click (with mouse or keyboard) onto the related directory-node in the directories-buffer of ECB (see ⟨undefined⟩ [Using the mouse], page ⟨undefined⟩).

See also the option `ecb-cache-directory-contents-not`. Here are some useful settings for both of these options:

- Cache all directories with more than 500 entries: Set `ecb-cache-directory-contents` to ((".*" . 500)) and set `ecb-cache-directory-contents-not` to nil.
- Cache only all directories > 200 beginning with /usr/ Set `ecb-cache-directory-contents` to (("^/usr/.*" . 200)) and set `ecb-cache-directory-contents-not` to nil.
- Cache all directories > 500 but NOT these beginning with /usr/: Set `ecb-cache-directory-contents` to ((".*" . 500)) and set `ecb-cache-directory-contents-not` to ("^/usr/.*").

Another way for getting a faster overlook for large directories with many source-entries is to apply an online-filter to the sources-buffer. This can be done via the command `ecb-sources-filter` or via the popup-menu of the sources-buffer.

[3] The standard behavior is switching between the speedbar-frame and the attached frame, but this make obviously no sense during running speedbar with ECB.

10.7 Working with remote directories

The term "remote" means directories which are remote in the sense of TRAMP[4], ANGE-FTP[5] or EFS[6]. Each of these Emacs-addons is intended to make editing directories and files on remote machines as transparent as possible.

10.7.1 General remarks

ECB supports such remote directoires out of the box and completely transparently, i.e. you can add remote directories to the option `ecb-source-path` without any restriction. ECB will handle these directories transparently with the appropriate tool - either TRAMP, ANGE-FTP or EFS. So when working with such a remote directory is possible without ECB it will be possible too with active ECB - at least as long you are "connected"!

Caution: Suppose you have added a remote dir (e.g. "user@host.at.a.server:/dir/") to `ecb-source-path` and you start ECB when you are offline, means there can be no connection established to the remote computer (e.g. "host.at.a.server"). Each time ECB has to process a remote path ECB pings via the ping-program the remote host (in the example above it would ping the host "host.at.a.server") to test if it is accessible. If not then this path will be ignored by ECB[7]. Ensure that ECB calls your ping-program (see `ecb-ping-program`) with the right options (see `ecb-ping-options`). To avoid to many pings to the same host ECB caches the ping result so there should be no performance decrease. But to ensure still correct accessible-results and to avoid using outdated cache-results ECB discards the cached value of the accessible-state of a certain host after a customizable time-interval (please read the documentation of `ecb-host-accessible-check-valid-time`!).

10.7.2 Excluding remote directories from time-consuming tasks

ECB performs some tasks stealthily and interruptable by the user (see the option `ecb-stealthy-tasks-delay` for additional explanations) because these tasks are time-consuming and could otherwise ECB block. Especially for remote directories these special tasks can cause annoying blocks of Emacs (see ⟨undefined⟩ [Stealthy background tasks], page ⟨undefined⟩).

Therefore it is probably the best to switch on each of the stealthy tasks with the `unless-remote` which is the default activation (see ⟨undefined⟩ [Stealthy background tasks], page ⟨undefined⟩). So a certain stealthy task will be swtiched on for all local directories (and also for all mounted drives in the LAN) but not for real remote directories used via TRAMP, ANGE-FTP or EFS.

---

[4] TRAMP stands for 'Transparent Remote (file) Access, Multiple Protocol'. This package provides remote file editing, similar to ANGE-FTP.

[5] This package attempts to make accessing files and directories using FTP from within Emacs as simple and transparent as possible.

[6] A system for transparent file-transfer between remote hosts using the FTP protocol within Emacs

[7] This avoids long lasting and annoying blocking of ECB when a remote-path is not accessible: Without a ping ECB would always try to open this directory through the appropriate library (e.g. TRAMP) and it would depend on the timeout-mechanism of this library (e.g. TRAMP has 60 seconds) how long ECB would be blocked. First after this timeout ECB could start working! A fast "pre"-ping avoids this problem!

10.7.3 Caching the contents of remote directories

ECB caches per default the contents of remote directories to avoid annoying delays. The cache is done via the option `ecb-cache-directory-contents` which contains an entry which covers the syntax of remote directories. If you do not want this caching (which is strongly recommened) you have to remove this entry from this option.

10.8 Supporting Version control systems

Beginning with version 2.30 ECB supports Version-control systems (in the following named VC-systems). This means the special tree-buffers of ECB display files managed by a VC-system with an appropriate image-icon[8] in front of the filename.

The following four options allow full control over this feature (see also ⟨undefined⟩ [ecb-version-control], page ⟨undefined⟩:

`ecb-vc-enable-support`
: Enable or disable this feature.

`ecb-vc-supported-backends`
: The most important option for this feature. Allows to specify how ECB should test if a directory is managed by a VC-system (how to identify the VC-backend of a directory) and - if yes - how it should check the VC-state of a certain file. The former ones are called *identify-backend-functions* and the latter ones *check-state-functions*.

`ecb-vc-directory-exclude-regexps`
: Allows excluding certain directories (on a regexp-basis) from the VC-support even if they are managed by a VC-system.

`ecb-vc-state-mapping`
: Defines the mapping between the state-values returned by a check-state-function (a function set in `ecb-vc-supported-backends` and used for getting the VC-state of a file, e.g. `vc-state`) and the allowed state-values ECB can understand.

Probably the default settings will fit your needs but to get sure you should carefully read the documentation of these options!

The following subsection give you important informations about identify-backend-functions, check-state-functions, about working with remote repositories.

10.8.1 How ECB identifies the VC-backend of a dir

ECB tries all functions added as identify-backend-funtions to the option `ecb-vc-supported-backends` until one of them returns not `nil` but a symbol which identifies the backend (e.g. `CVS`). After this check ECB stores the result of this check (i.e. either the identified backend or the fact that the directory is not managed by a VC-system) for that directory in a special cache, so the identify-backend-process will be performed only once per directory. If for a directory a VC-backend could be identified ECB stores not only the backend itself for that directory but also the associated check-state-function defined in `ecb-vc-supported-backends` (see ⟨undefined⟩ [Checking the state], page ⟨undefined⟩).

[8] Of course only when Emacs is capable to display images; otherwise a suitable ascii-icon will be displayed.

You can add arbitrary functions to this options as long as they get one directory-argument and return either nil or a backend-symbol. Per default ECB offers the following functions to identify the VC-backend CVS, RCS, SCCS or Subversion[9]:

`ecb-vc-dir-managed-by-CVS DIRECTORY`
: Return `CVS` if DIRECTORY is managed by CVS. nil if not.

  This function tries to be as smart as possible: First it checks if DIRECTORY is managed by CVS by checking if there is a subdir `CVS`. If no then nil is returned. If yes then for GNU Emacs it takes into account the value of `vc-cvs-stay-local`: If t then just return `CVS`. Otherwise ECB checks the root repository if it is a remote repository. If not just `CVS` is returned. If a remote repository it checks if the value of `vc-cvs-stay-local` is a string and matches the host of that repository. If yes then just `CVS` is returned. If not then ECB checks if that host is currently accessible by performing a ping. If accessible `CVS` is returned otherwise nil. This has the advantage that ECB will not be blocked by trying to get the state from a remote repository while the host is not accessible (e.g. because the user works offline).

  Special remark for XEmacs: XEmacs has a quite outdated VC-package which has no option `vc-cvs-stay-local` so the user can not work with remote CVS-repositories if working offline for example. So if there is no option `vc-cvs-stay-local` then ECB performs always the repository check mentioned above.

`ecb-vc-dir-managed-by-RCS DIRECTORY`
: Return `RCS` if DIRECTORY is managed by RCS. nil if not.

`ecb-vc-dir-managed-by-SCCS DIRECTORY`
: Return `SCCS` if DIRECTORY is managed by SCCS. nil if not.

`ecb-vc-dir-managed-by-SVN DIRECTORY`
: Return `SVN` if DIRECTORY is managed by Subversion. nil if not. Returns always nil if the library vc-svn.el can not be found.

If ECB should support another VC-backend than CVS, RCS, SCCS or Subversion you have to write your own identify-backend-funtion for the used VC-backend (e.g. Clearcase)!

10.8.1.1 Special remarks for XEmacs

XEmacs contains only a quite outdated VC-package, especially there is no backend-independent check-vc-state-function available (like `vc-state` for GNU Emacs). Only for CVS a check-vc-state-function is available: `vc-cvs-status`. Therefore ECB adds per default only support for CVS and uses `ecb-vc-managed-by-CVS` rsp. `vc-cvs-status`. See also ⟨undefined⟩ [Known VC-problems], page ⟨undefined⟩!

10.8.2 How ECB checks the VC-state of a file

After ECB has identified the VC-backend of a directory it will display the VC-state (e.g. up-to-date, edited, needs-mergs etc...) with a suitable image-icon in the tree-windows of the ECB-file-browser. To get this state for a certain file ECB uses that check-state-function

[9] For this the most recent version of the VC-package (incl. the library vc-svn.el) is needed - as contained in CVS Emacs

stored in the cache for the directory of that file (see ⟨undefined⟩ [Identifying backends], page ⟨undefined⟩).

You can add any arbitrary functions as check-state-function to `ecb-vc-supported-backends` as long as they get one filename-argument and return a state-symbol (e.g. `up-to-date`. ECB can understand a certain set of state-values which are then mapped to suitable image-icons which will in turn be displayed in front of the filename in the file-browser. Because the values a check-state-function return can differ from that state-values ECB understands, ECB offers an option to define a appropriate state-mapping. The name of this option is `ecb-vc-state-mapping`. See the documentation of this option to get a list of all state-value ECB understands.

Per default ECB uses - when running under GNU Emacs - the function `vc-state` of the VC-package[10] to check the state for the backends CVS, RCS, SCCS and Subversion. So the default-value of `ecb-vc-state-mapping` contains a mapping between these values `ecb-vc-state` can return and that state-values ECB understands.

If ECB should support other VC-backends than CVS, RCS, SCCS and Subversion (e.g. Clearcase) you should add that new backend to the VC-package (see the initial comments of vc.el how to do this) then ECB will automatically support that new backend. Alternatively it may be sufficient if you write your own check-state-function for this backend and add the needed mapping to `ecb-vc-state-mapping` if necessary.

10.8.2.1 Getting heuristic state-values or real ones for CVS

The interface of GNU Emacs' VC-package offers two different ways to get the VC-state of a file:

- The real, fresh and expensive approach VC has a function `vc-recompute-state` which always performs a command "cvs status" to get a fresh and real state for a file. As you can imagine this operation can be very expensive and long lasting depending on the location of the repository. But the CVS-backend of VC offers with the option `vc-cvs-stay-local` a way to tell Emacs to stay local even for the sake of getting a real state.
- The heuristic approach: The function `vc-state` always returns a "heuristic" state which should be used when a fresh and real state is not necessary. With `vc-state` the option `vc-cvs-stay-local` will never take effect.

VC/CVS actually does it this way (regardless if ECB is active or not): When you visit a file, it always uses just the heuristic to get the state (comparing file times), regardless of the setting of `vc-cvs-stay-local`. This is because the `"fresh-but-slow"` state is determined by calling `"cvs status"` on the file, and this was deemed unacceptably slow if done at visiting time under any conditions.

The state is updated by calling `vc-recompute-state` prior to `vc-next-action` (C-x v v) which either checks a file in or out. IF `vc-cvs-stay-local` is nil, then this does in fact call `"cvs status"` to get the `"fresh-but-slow-state"`, but if `vc-cvs-stay-local` is t, then it just compares the file times again.

But under certain conditions (e.g. if called for files not already visited or for files their VC-state has been changed from outside Emacs, e.g. by checking in the file via command

[10] The VC-package of Emacs offers a standardised and uniform interface for several backends; per default CVS, RCS, SCCS and Subversion are supported by the VC-package.

line) `vc-state` does not compute a new heuristic state but returns a cached one (cached by the VC-package itself not by ECB) which does not reflect the current VC-state. Example: if you have edited a file within Emacs and then checked in from outside Emacs `vc-state` returns a wrong state until you call `revert-buffer` for this file. Therefore ECB offers the check-state-function `ecb-vc-state` which does the same as `vc-state` but it clears the internal caches of the VC-package for that file before calling `vc-state`.

The bottom line for you is this: If you use `ecb-vc-state` in `ecb-vc-supported-backends` to get the version control state, then you get the same policy that VC uses and you get always a "correct" heuristic state (as correct as possible a heuristic state can be). There should no harm if you use `vc-recompute-state` as a replacement function if you want to get fresh and real state-values, but then (a) you must make sure to set `vc-cvs-stay-local` to nil, and (b) fetching the state over the network under all conditions was deemed unacceptably slow in VC.

10.8.3 Important informations about remote repositories

At least CVS can be used in a mode called "Client/Server" which means the root repository is located on a remote machine. We call a repository which can not being mounted by the client-machine (which contains the working directory) a *remote repository*. In most cases getting the fresh and real VC-state for such repositories will be unacceptable slow or often users will work offline means with no connection available to the remote host. To avoid problems like these ECB offers first an option `ecb-vc-directory-exclude-regexps` to exclude such directories with a remote repository from the VC-support of ECB and secondary the identify-backend-funtion `ecb-vc-dir-managed-by-CVS` behaves smart with that respect (see ⟨undefined⟩ [Identifying backends], page ⟨undefined⟩). See also `ecb-vc-xemacs-exclude-remote-cvs-repository`!

10.8.3.1 Remote paths and the VC-support of ECB

ECB supports working with remote directories like TRAMP- or EFS-directories (see ⟨undefined⟩ [Remote directories], page ⟨undefined⟩). Do not confuse remote directories with remote repositories. A local directory located on your disk and set in `ecb-source-path` can have a remote repository if managed by a VC-system. A remote directory means a path in the format of TRAMP, ANGE-FTP or EFS set in `ecb-source-path`. Its very likely that getting the VC-state of files contained in such a remote directory would be extremly expensive and therefore ECB would be blocked quite long even if the VC-check is performed stealthy (see ⟨undefined⟩ [Stealthy background tasks], page ⟨undefined⟩).

To avoid problems with such remote directories ECB prevents per default such directories from being processed by the VC-support of ECB. But if a user is dying to having the VC-state being displayed in the tree-buffers ECB offers two ways to switch on the VC-support - see the option `ecb-vc-enable-support`: This option is set per default to the value `unless-remote` which means remote paths will not be processed but it can be set to `t` which means process all directories regardless if remote or not. It's strongly recommended to use `unless-remote`!

10.8.4 How to refresh ECB-state-display when changed outside

If all actions concerning version controlling of a file are performed within Emacs with commands offeres by VC then the displayed state for such a file in the tree-buffers of ECB

will be always correct - in that sense that ECB will always display that state which the check-state-function for the file will return. At least with GNU Emacs for the backends CVS, RCS, SCCS and Subversion this will be true. With XEmacs only for CVS. For other backends see ⟨undefined⟩ [Adding new backends], page ⟨undefined⟩.

But if the VC-state of a file will be changed outside of Emacs (unfortunately PCL-CVS must be called "outside" too because PCL-CVS doesn't use the functions of the VC-package of Emacs for checking-in or -out) then ECB can not automatically recognize this and therefore it can not aurtomatically update the displayed state-image-icon. You have to tell ECB for which files in the tree-buffers the VC-state should be recomputed. This can be done via the popup-menus of the ECB-tree-buffers - The following popup-commands are offered in the submenu "Version Control":

ECB-directories-buffer (if sources are displayed within):
: "Recompute state for file" and "Recompute state for dir" whereas the latter one recomputes the VC-state for all files of that directory the file belongs.

ECB-sources-buffer
: "Recompute state for file" and "Recompute state for dir" whereas the latter one recomputes the VC-state for all files currently displayed in the sources-buffer.

ECB-history-buffer
: "Recompute state for file" and "Recompute state for whole history" whereas the latter one recomputes the VC-state for all file-entries currently displayed in the history-buffer.

Caution: The state will only recomputed right under all situations if you use either `ecb-vc-state` or `vc-recompute-state` as check-state-function in `ecb-vc-supported-backends` (see ⟨undefined⟩ [Checking the state], page ⟨undefined⟩).

Of course all these commands update the VC-state in all visible tree-buffers the file is currently displayed (e.g. often a file is displayed in the sources- and the history-buffer)!

For general informations about the usage of popup-menus in ECB see ⟨undefined⟩ [Using the mouse], page ⟨undefined⟩ (subsection "The right mouse button").

In addition to these popup-commands using the POWER- rsp. Shift-click (see ⟨undefined⟩ [Using the mouse], page ⟨undefined⟩) onto a directory in the directory-window of ECB refreshes the VC-state-values of all files contained in this directory too.

10.8.5 Necessary steps and informations for adding new backends

There are mainly three necessary steps for adding a new[11] backend BE which should be supported by ECB:

1. Adding an identify-backend-function to `ecb-vc-supported-backends` ECB needs a function how to identify the new backend BE for a certain directory. If there exists already a library (other then VC) supporting this backend then this library propably contains already such a function which can be used or can be used at least with a small elisp-wrapper. If no elisp-library for backend BE exists then you have probably write the full identify-backend-function for your self. This function has to be added to `ecb-vc-supported-backends`.

[11] i.e. not already supported by the VC-package because all these backends are automatically supported by ECB too!

2. Adding an check-state-function to `ecb-vc-supported-backends` Associated to the new identify-backend-function mentioned in step 1 a new check-state-function is needed which can be used by ECB to get the VC-state for a file. See ⟨undefined⟩ [Checking the state], page ⟨undefined⟩ for a description about the needed interface of such a function. In combinatio with the identify-backend-function from step 1 this function has to be added to `ecb-vc-supported-backends`.
3. Enabling automatic state-update after checkin/out

   This step is not essential if you do not need the displayed VC-state automatically updated after a checkin/out of a file via the commands available for backend BE (e.g. clearcase.el offers for the backend Clearcase elisp-commands to checkin and checkout a file which then should also update the displayed state in the ECB-tree-buffers. All you need is a way to tell these commands that they should clear the ECB-VC-cache for the file and then restart the ECB-VC-check-mechanism. This should be done after these commands have finished their original job.

   ECB enables this per default for all backends supported by the VC-package with the following code. Maybe this is a good starting point.

```
(defvar ecb-checkedin-file nil
  "Stored the filename of the most recent checked-in file. Is only set by the
after-advice of ‘vc-checkin’ and ‘ecb-vc-checkin-hook’ \(resets it to nil).
Evaluated only by ‘ecb-vc-checkin-hook’.

This is the communication-channel between ‘vc-checkin’ and
‘ecb-vc-checkin-hook’ so this hook-function gets the filename of the
checked-in file.")

(defadvice vc-checkin (after ecb)
  "Simply stores the filename of the checked-in file in ‘ecb-checkedin-file’
so it is available in the ‘vc-checkin-hook’."
  (setq ecb-checkedin-file (ecb-fix-filename (ad-get-arg 0))))

(defun ecb-vc-checkin-hook ()
  "Ensures that the ECB-cache is reset and the entry for the most recent
checkedin file is cleared. Uses ‘ecb-checkedin-file’ as last checked-in file."
  (when ecb-checkedin-file
    (ecb-vc-cache-remove ecb-checkedin-file)
    (ecb-vc-reset-vc-stealthy-checks)
    (setq ecb-checkedin-file nil)))
```

10.8.6 Currently know problems with the VC-support

10.8.6.1 Remote repositories and XEmacs

Currently there are mostly problems related to XEmacs - cause of its outdated VC-package which allows no heuristic state-computation but always runs “cvs status” to get the VC-state for a file (done by `vc-cvs-status`). This can be horrible slow for remote CVS-root-repositories. Now ECB performs the VC-check stealthy and only in idle-time of Emacs but even so XEmacs can be blocked espcially if the cygwin-build of XEmacs is used: This

XEmacs-version is substantially slower concering file-operations and has sometimes a very slow and delayed response-behavior for mouse- and keyboard interrupts - so even ECB let the user interrupt by using `input-pending-p` before getting the VC-state of a file XEmacs sometimes does not react to such user-interrupts and seems to be blocked.

Current solution: ECB offers the option `ecb-vc-xemacs-exclude-remote-cvs-repository` which excludes remote repositories from being checked. This option is per default t for XEmacs. Whenever XEmacs syncs up its VC-package with the Emacs one this option will automatically take no effect.

10.9 Optimal using of eshell in ECB

ECB offers a very smart integration of the "eshell" if you are using a compile window (see ⟨undefined⟩ [Temp- and compile-buffers], page ⟨undefined⟩)[12]

Here is a short summary of provided features:

- Ability to jump to the eshell buffer within the compilation window by simply call `eshell` (bound to `C-c . e`). If the eshell isn't running it will be started.
- Expands the compilation window when you run commands. So for example it allows you to view the eshell in minimized mode and then when you run "ls" the window automatically expands (but always depending on the output of the command you run).
- Synchronizes the current directory of the eshell with the current buffer in the current active edit-window of ECB.
- Provides smart window layout of the eshell buffer. This makes sure that the eshell is taking up the exact amount of space and that nothing is hidden.

Here comes a detailed explanation of these features and how to use it (all these features are only available if you use a persistent compile-window, i.e. if `ecb-compile-window-height` is not nil):

You do not have to learn a new command for the eshell-start - just call `eshell` (for convenience also bound to `C-c . e`) and the eshell will displayed in the compile-window of ECB (if eshell is not already alive then it will be started automatically).

ECB tries to display the contents of the eshell-buffer as best as possible, means ECB can autom. enlarge and shrink the compile-window so the contents of the eshell are fitting the window. See option `ecb-eshell-enlarge-when-eshell` and `ecb-eshell-fit-window-to-command-output`. Normally this is done autom. but you can also you the standard compile-window enlarging-command of ECB: `ecb-toggle-compile-window-height`.

ECB tries also to recenter the eshell-buffer as best as possible. Normally this is done autom. but you can do it on demand with the command `ecb-eshell-recenter`.

If option `ecb-eshell-synchronize` is true then ECB always synchronizes the command prompt of eshell with the directory of current source-buffer of the current active edit-window.

With the option `ecb-eshell-auto-activate` you can start eshell autom. in the compile-window when ECB is started but of course if a compile-window exists.

[12] Of course you can use eshell also if there is no compile-window. Then it is just displayed in the edit-area and there is no special integration.

10.10 Grepping directories with ECB

ECB offers in the popup-menus in the directories- and sources-tree-buffer commands for easy (recursive) grepping the current directory under point (directory-buffer) rsp. the current-directory (sources-buffer). In every case just the function of the options `ecb-grep-function` rsp. `ecb-grep-find-function` is called and the `default-directory` is tempor. set to the chosen directory so the grep will performed in this directory regardless of the `default-directory` of current buffer in the edit-window.

Other smart things beyond that are not done by ECB, see also `ecb-grep-function`!

So, how to exclude some subdirectories or files from the grep?

Basically this has to be done with the "-prune" option of the find-utility: If the standard-grep facility of Emacs is used then this is not easy but with the library '`igrep.el`' there is a convenient way to exclude things like CVS- or RCS-directories from the find-call: See the variable `igrep-find-prune-clause` of the library '`igrep.el`'.

10.11 Working best with ECB and JDEE

ECB is completely language independent, i.e. it works with any language supported by semantic (e.g. C, C++, Java, Elisp etc.).

But there are some special integrations for the great Java-Development-Environment JDEE:

- Displaying contents of class under point

  With the command `ecb-jde-display-class-at-point` you can display the contents of the class which contains the definition of the "thing" at point (which can be a method, variable etc.).

- Creating new source-files

  The popup-menus in the directories- and the sources-buffer offer a command "Create Source" which allows easy creating new java-sources by calling the command `jde-gen-class-buffer`.

- Adding user-extensions to the popup-menus

  The options `ecb-directories-menu-user-extension` and `ecb-sources-menu-user-extension`[13] allow adding often used JDEE-commands to the popup-menus of the directories- or sources-buffer. One example is to add building the project of current directory. Here is a function which could be added to `ecb-directories-menu-user-extension`:

```
(defun ecb-dir-popup-jde-build (node)
  "Build project in directory."
  (let ((project-file
 (expand-file-name jde-ant-buildfile (tree-node-get-data node))))
    (jde-ant-build project-file "build")))
```

  Of course you can add entries to the option `ecb-methods-menu-user-extension` and `ecb-methods-menu-user-extension` too.

[13] If you need a dynamic way of menu-extension then you should have a look at the options `ecb-directories-menu-user-extension-function` and `ecb-sources-menu-user-extension-function`.

10.12 Displaying the compile-window on demand

If you like displaying all output of compile/grep/etc. an all temp-buffers like *Help*-buffers in an extra compile-window (see ⟨undefined⟩ [Temp- and compile-buffers], page ⟨undefined⟩) but you dislike wasting the space of this compile-window if you are just editing then you can get a compile-window "on demand". Just do the following:

1. Customize `ecb-compile-window-height` to not nil and save it for future sessions. This gives you an extra compile-window at the bottom.
2. Add the following to your .emacs:

```
(add-hook 'ecb-activate-hook
          (lambda ()
            (let ((compwin-buffer (ecb-get-compile-window-buffer)))
            (if (not (and compwin-buffer
                          (ecb-compilation-buffer-p compwin-buffer)))
                (ecb-toggle-compile-window -1)))))
```

   This hides the extra compile-window directly after the start of ECB because there is no need for a compile-window at this moment. But the hiding will not be done if there is a compile-window and if a "compile-buffer" in the sense of `ecb-compilation-buffer-p` is displayed in this compile-window. Without this additional check the compile-window would always be hidden after the ECB-start even when ECB is reactivated after a deactivation by the window-manager-support of ECB (see ⟨undefined⟩ [Window-managers and ECB], page ⟨undefined⟩); but in these cases we want to preserve the state before deactivation as good as possible (see also option `ecb-split-edit-window-after-start`).

This is all you have to do. Now if you run `compile` (or `grep` or other compile-modes) or display temp-buffers like *Help*-buffers then ECB autom. displays the compile-window at the bottom and display the output there.

If you have finished with using the compile- or temp-output (e.g. fixing errors) then you can throw away the compile-window just by `ecb-toggle-compile-window` - ECB will reactivate it autom. before next compilation or help-buffer-display.!

10.13 Parsing and displaying non-semantic sources

ECB is mostly designed to display parsing information for files supported by semantic. But beginning with version 1.94 it also supports other parsing engines like imenu and etags, so also files not supported by semantic but by imenu/etags can be displayed in the Method-buffer of ECB. See ⟨undefined⟩ [Definition of semantic- and non-semantic-sources], page ⟨undefined⟩ for a description of "semantic-sources" and "non-semantic-sources".

If support of non-semantic-sources is enabled then ECB will display the contents of all sources which can be displayed by speedbar too. This comes from the fact that ECB uses speedbar-logic to parse sources with imenu or etags.

In most cases imenu-parsing is preferable over etags-parsing because imenu operates on Emacs-buffers and needs no external tool and therefore parsing works also if current contents of a buffer are not saved to disk.

This section describes all important aspects about parsing and displaying file-contents of file-types not supported by semantic but by imenu and/or etags.

10.13.1 Enabling parsing and displaying of non-semantic-sources

Enabling is simply done with the option `ecb-process-non-semantic-files`.

ECB offers an option `ecb-non-semantic-parsing-function` to specify on a major-mode basis which parsing-method should be used: imenu or etags. Normally there should be no need to change this option but read the documentation of this option (see ⟨undefined⟩ [ecb-non-semantic], page ⟨undefined⟩) for further details.

IMPORTANT:

- If imenu-parsing should be used then the option `speedbar-use-imenu-flag` must be set to not `nil`!
- If some non-semantic-sources are not parsed (i.e. there is an empty Methods-buffer) and you think that they should then maybe they are neither supported by imenu nor by etags or you have to check the options `ecb-non-semantic-parsing-function` and `speedbar-dynamic-tags-function-list` and - especially for etags - `speedbar-fetch-etags-parse-list`, `speedbar-fetch-etags-arguments` and `speedbar-fetch-etags-command`.
- Even with support for semantic-, imenu- and etags-parsing there will remain some file-types rsp. `major-modes` which are not parse-able, neither by semantic, imenu nor etags. This is no problem because these files simply have an empty Methods-buffer. But nevertheless you will get a message "Sorry, no support for a file of that extension" which comes from the speedbar-library and can not switched off. Therefore if a `major-mode` is known as not parse-able by semantic, imenu or etags it can be added to the option `ecb-non-semantic-exclude-modes` and then it will be excluded from being tried to parsed and this (annoying) message will not occur.

10.13.2 Automatic rescanning/reparsing of non-semantic-sources

In contrast to semantic (see `global-semantic-auto-parse-mode`) there is no built-in mechanism for autom. reparsing non-semantic-sources and then updating the contents of the Methods-buffer.

For non-semantic-sources you have always at least to call `ecb-rebuild-methods-buffer` (bound to *`C-c . r`*) or saving the source-file (if `ecb-auto-update-methods-after-save` is true) to update the Method-buffer[14].

Depending on the parsing-mechanism the following options have to be switched on so ECB can rebuild the methods-buffer for non-semantic-sources:

- imenu:

  The imenu-option `imenu-auto-rescan` must be enabled and `imenu-auto-rescan-maxout` has to be set big enough to auto-parse big files too! But this results not directly in an autom. updated Method-buffer. This is first done after calling the command `ecb-rebuild-methods-buffer` or saving the source-file (if `ecb-auto-update-methods-after-save` is true).

- etags:

  Only if `ecb-auto-save-before-etags-methods-rebuild` is switched on the command `ecb-rebuild-methods-buffer` rebuilds the method-buffer with current source-contents. See description of this option for an explanation.

[14] Maybe future versions of ECB (> 1.94) will offer an autom. mechanism for this.

Tip: If you want to program your own real. automatic rescan/reparse/rebuild mechanism for non-semantic-sources you can do:

Adding to `after-change-functions` a function F which either runs itself `ecb-rebuild-methods-buffer-for-non-semantic` or which adds only another function FF to an idle-timer and the function FF runs `ecb-rebuild-methods-buffer-for-non-semantic`. The latter approach has the advantage that the reparse/rebuild is not performed immediately after every change but first after Emacs is idle for a senseful interval (e.g. 4 seconds) after last change. Of course the function FF has to cancel its own idle-timer at the end, so the next idle-timer is first started again after the next change (i.e. by function F which is still contained in `after-change-functions`.

10.13.3 Customizing the display of the tags

For non-semantic-sources ECB uses does no special organizing of tags in groups and sub-tags but it completely uses the tag-hierarchy the imenu- and etags-parsers of speedbar return. So the displayed tag hierarchy can only be customized with some options speedbar offers for this:

`speedbar-tag-hierarchy-method`, `speedbar-tag-group-name-minimum-length`, `speedbar-tag-split-minimum-length` and `speedbar-tag-regroup-maximum-length`. See the speedbar documentation for details about these options.

With the option `ecb-method-non-semantic-face` you can define the face used for displaying the tags in the Method-buffer for non-semantic-sources.

`ecb-non-semantic-methods-initial-expand` can be useful too.

10.14 Using hide-show from the methods-buffer-menu

The popup-menu of the Methods-buffer offer two entries for either hiding or showing the block which is related to the selected tag (that tag for which the popup-menu was opened):

- "Jump to tag and hide block": Jumps to the tag and calls `hs-hide-block` from the hideshow-library which is shipped with (X)Emacs. After that the block is hidden, i.e. only the header-line of that tag (method, variable etc.) is visible, the rest is hidden behind the "...".
- "Jump to tag and show block": Jumps to the tag and calls `hs-show-block`. This shows the related hidden block if the block was hidden via `hs-hide-block` or the menu-entry "Jump to tag and hide block" (s.a.).

For this feature the library '`hideshow.el`' is used which should normally being included in the (X)Emacs-distribution. If this library is not loaded into Emacs, ECB does this automatically before the first call to one of these menu-entries.

IMPORTANT: If in some `major-mode` hiding and showing does not work as you expect it to work then you must probably add an entry for this `major-mode` to the hideshow-variable `hs-special-modes-alist`. See the documentation of this variable for further details. One example of such a `major-mode` is `jde-mode` of the Java Development Environment JDEE; just add an entry for it like the already contained entries for `c++-mode` or `java-mode` and hiding and showing will work for you with JDEE too.

10.15 Support of several Emacs-window-managers

There are several window-managers available which offer an easy interface to jump between different window-configurations within the same frame. A window configuration is the layout of windows and associated buffers within a frame. There is always at least one configuration, the current configuration. You can create new configurations and cycle through the layouts in either direction. Window configurations are often named or numbered, and you can jump to and delete named rsp. numbered configurations.

Without special support by ECB these window-managers would not work in combination with ECB!

ECB currently supports the following managers:

- winring.el: Written by Barry A. Warsaw `bwarsaw@python.org`, available at `http://www.python.org/emacs/`
- escreen.el: Written by Noah Friedman `friedman@splode.com`, available at `http://www.splode.com/~friedman/software/emacs-lisp/`

IMPORTANT: With one of these window-managers installed and active you can run applications like Gnus, VM or BBDB in the same frame as ECB! Just use different window-configurations (winring.el) or escreens (escreen.el) for ECB and the other applications. Especially with winring.el you can give every configuration a descriptive name like "ECB" or "Gnus"; afterwards you can jump to a window-configuration by name!

When you go back to the ECB-window-configuration (winring.el) or the ECB-escreen (escreen.el) with any of the special window-manager-commands then the state of ECB will be restored exactly as you have left it when going to another window-configuration rsp. escreen. This includes the whole splitting state of the edit-area and the visibilty of the ecb-windows and of the compile-window!

The rest of this section describes how to enable the special ECB-support for these window-managers and how to use them.

10.15.1 Enabling of the support

Every support must be enabled explicitly:

- winring: Call `ecb-winman-winring-enable-support`. This **MUST** be done **BEFORE** the first call to any winring-command, so also before calling `winring-initialize`!
- escreen: Call `ecb-winman-escreen-enable-support`. This **MUST** be done **BEFORE** the first call to any escreen-command, so also before calling `escreen-install`!

If a window-manager-support should be enabled autom. after Emacs-start just put the following into your '`.emacs`':

```
(ecb-winman-winring-enable-support)
(winring-initialize)

;; or - if you like escreen more

(ecb-winman-escreen-enable-support)
(escreen-install)
```

10.15.2 Usage of a window-manager in combination with ECB

After enabling the support of one of the supported window-managers just go on as described in the commentary or introduction of the respective library-file(s) of the window-manager. Here is a short description:

- winring: First you have to define how to identify the ECB-window-configuration, i.e. the configuration with activated ECB. This done with the option `ecb-winman-winring-name`. There is always only one window-configurations with name `ecb-winman-winring-name`!

  Then run `winring-initialize`. If ECB is active then the resulting window-configuration is the ECB-window-configuration. Otherwise you can create the ECB-window-configuration when you first time call `winring-new-configuration` with name equal to `ecb-winman-winring-name`. In general you can run all commands of the winring-library. If you jump to the ECB-window-configuration then ECB will be autom. activated and if you leave the ECB-window-configuration then ECB will autom. deactivated.
- escreen: First you have to define how to identify the ECB-escreen i.e. that escreen with activated ECB. This done with the option `ecb-winman-escreen-number`. There is always only one escreen with number `ecb-winman-escreen-number`!

  Then run `escreen-install` (deactivates ECB if currently running). After that you can call `escreen-create-screen` and `escreen-goto-screen`[15]. These commands autom. activate ECB if creating or selecting the escreen with number `ecb-escreen-number` (default = 1) and autom. deactivate ECB if leaving the ECB-escreen.

10.15.3 Disabling the support

There is normally no need to do this but nevertheless it can be done by `ecb-winman-escreen-disable-support` rsp. `ecb-winman-winring-disable-support`.

10.16 Using semanticdb to jump to type-tags defined in other files

In OO-languages like CLOS, eieio and C++ there can be type-tags in the method-buffer which are somehow virtual because there is no definition in the current source-file. But such a virtual type collects all its outside defined members like methods in C++ which are defined in the '`*.cc`' file whereas the class-definition is defined in the associated header-file. ECB uses semanticdb to open the definition-file of such a tag and to jump to the definition of this tag. Same for parent-tags in the methods-buffer. This feature can only work correctly if semanticdb is well configured!

Here is a C++-example:

This class is defined in a file '`ParentClass.h`':

```
class ParentClass
{
protected:
  int p;
};
```

[15] And of course all other `escreen-goto-*` commands!

This class is defined in a file 'ClassWithExternals.h'

```
#include "ParentClass.h"

class ClassWithExternals : public ParentClass
{
private:
  int i;

public:
  ClassWithExternals();
  ~ClassWithExternals();
};
```

Both the constructor and the destructor of the class "ClassWithExternals" are defined in a file 'ClassWithExternals.cc':

```
#include "test.h"

ClassWithExternals::ClassWithExternals(int i,
                                       boolean b,
                                       char c)
{
  return;
}

void
ClassWithExternals::~ClassWithExternals()
{
  return;
}
```

ECB displays the contents of 'ClassWithExternals.cc' in its methods-buffer like follows:

```
[-] [Includes]
 `- test.h
[-] ClassWithExternals
 |  +ClassWithExternals (+i:int, +b:class boolean, +c:char):ClassWithExternals
 `- +~ClassWithExternals ():void
```

Both the constructor and the destructor of the class "ClassWithExternals" are grouped under their class-type. ECB now uses semanticdb to jump to the definition of class "ClassWithExternals" when you click onto the type-node "ClassWithExternals" in the methods-buffer.

The contents of 'ClassWithExternals.h' are displayed like follows:

```
[-] [Includes]
 `- ParentClass.h
[-] ClassWithExternals:class
 |  [-] [Parents]
 |   `- ParentClass
 |  [-] [Variables]
 |   `- -i:int
 |  +ClassWithExternals ():ClassWithExternals
 |  +~ClassWithExternals ():void
 `- [+] [Misc]
```

ECB uses semanticdb to jump to the definition of the class "ParentClass" when you click onto the node "ParentClass".

To enable this feature `global-semanticdb-minor-mode` must be enabled and semanticdb must be correctly configured. This means mainly that the option `semanticdb-project-roots` must be setup well. See the manual of semanticdb for further informations about this.

11 Entry points for Elisp programmers

This chapter describes how ECB can be used/programmed/driven by an Elisp-program. This contains:

11.1 Variables for Elisp-programs

Variables an Elisp-program can use beyond those ones mentioned in ⟨undefined⟩ [The layout-engine], page ⟨undefined⟩:

- `ecb-source-path-functions`

Look at the documentation of these variables to get a description.

11.2 Available hooks of ECB

The following hooks are available:

- `ecb-activate-before-new-frame-created-hook`
- `ecb-activate-before-layout-draw-hook`
- `ecb-activate-hook`
- `ecb-after-directory-change-hook`
- `ecb-before-activate-hook`
- `ecb-before-deactivate-hook`
- `ecb-common-tree-buffer-after-create-hook`
- `ecb-current-buffer-sync-hook`
- `ecb-deactivate-hook`
- `ecb-directories-buffer-after-create-hook`
- `ecb-hide-ecb-windows-after-hook`
- `ecb-hide-ecb-windows-before-hook`
- `ecb-history-buffer-after-create-hook`
- `ecb-methods-buffer-after-create-hook`
- `ecb-redraw-layout-after-hook`
- `ecb-redraw-layout-before-hook`
- `ecb-show-ecb-windows-after-hook`
- `ecb-show-ecb-windows-before-hook`
- `ecb-sources-buffer-after-create-hook`

Look at the documentation of these hooks to get a detailed description.

11.3 The library tree-buffer.el

The library tree-buffer.el is an ECB-independent library written completely in emacs lisp and can be used also by other applications than ECB. But the main purpose of tree-buffer.el is to offer a small but powerful API to create new tree-buffers for ECB, add new tree-nodes to a tree-buffer and thus use such a tree-buffer to display arbitrary information structured by a tree.

This chapter is for emacs-lisp-programmers and describes how to create a new tree-buffer, how to add new tree-nodes to a tree-buffer (includes removing and updating already existing tree-nodes) and how to use the offered tree-buffer- and tree-node-APIs.

11.3.1 General description of tree-buffers

This subchapter is a general introduction in the main concepts of a tree-buffer.

11.3.1.1 What is a tree-buffer?

A *tree-buffer* is meant to display certain informations (e.g. a directory-tree) in a tree-structure consisting of *tree-nodes*. Every line in a tree-buffer displays exactly one tree-node. Each node has exactly one parent-node and can have any arbitrary number of *children*-nodes. If a tree-node has no children then it is called a *leaf*. A tree-node contains several "slots" wheras the most important ones are the "name", "displayed-name" and "data". See ⟨undefined⟩ [A new tree-node], page ⟨undefined⟩ for a detailed explanation.

The difference between a natural tree like a fir and a tree-buffer is that the root(-node) of a tree-buffer is not visible but only its children. In the example below the nodes parent-node-1 and parent-node-2 are the children of the invisible root-node. Each tree-buffer has exactly one root-node which is created automatically by 'tree-buffer-create'.

If a tree-node contains at least one child it is displayed with a special expand/collapse-symbol (see the example below). This symbol allows expanding (rsp. collapsing) the tree-node wheras expanding means to display the children-nodes and collapsing means to hide the childrens of a tree-node.

Here is an example of a tree-buffer:

```
<root-node>  ------------------------[root-node (invisible)]
    [+] <parent-node-1>  -------.
    [-] <parent-node-2>  -------|
        [-] <expanded>  --------|
            <leaf-node-1>  -----|
            <leaf-node-2>  -----|-----[tree-nodes]
            <leaf-node-3>  -----|
            <leaf-node-4>  -----|
        [+] <collapsed>  -------
         |
         `-----------------[expand/collapse-symbol]
```

In most cases an action is triggered when clicking with the mouse onto a tree-node[1] (e.g. clicking onto "leaf-node-1" or "parent-node-1" in the example above). Which action is triggered by which key depends on what you specify at creation-time of the tree-buffer - see ⟨undefined⟩ [A new tree-buffer], page ⟨undefined⟩ for details.

The creation-interface of a tree-buffer allows defining special popup-menus when clicking with the right mouse-button (of course also possible via keyboard, see ⟨undefined⟩ [Tree-buffer keybindings], page ⟨undefined⟩) onto a tree-node (e.g. some senseful actions possible for directory-nodes like grepping this directory or performing version-control actions for this directory or something else).

[1] Of course using the keyboard is also possible, see ⟨undefined⟩ [Tree-buffer keybindings], page ⟨undefined⟩.

11.3.1.2 General recipe for a tree-buffer

The following sequence of tasks is the general recipe for a tree-buffer beginning from creation and ending with the display.

1. Create the tree-buffer Creating a new tree-buffer has to be done with `tree-buffer-create` for non ECB-tree-buffers and with the macro `defecb-tree-buffer-creator` when the tree-buffer should be used as an ECB-tree-buffer, so it is an ECB-interactor. See ⟨undefined⟩ [A new tree-buffer], page ⟨undefined⟩ for all details.
2. Add tree-nodes to the tree-buffer Adding nodes to the new tree-buffer (means make the new tree-buffer the current buffer and call `tree-node-new` for a new tree-node (note that a root-node for this tree-buffer has been autom. created by `tree-buffer-create`!). The first tree-node you add to a tree-buffer must have always the root-node (avaliable via `tree-buffer-get-root`) as parent-node. The next nodes can have either one of the fromerly added nodes or the root-node too. All tree-nodes haveing the root-node as parent will be displayed at the toplevel of the tree-buffer. See ⟨undefined⟩ [A new tree-node], page ⟨undefined⟩ for all details.
3. Display the tree-buffer with current nodes and state When you are finished building up the tree-node-structure call `tree-buffer-update` to display the current tree-structure (again after making the tree-buffer the current-buffer). See ⟨undefined⟩ [Updating a tree-buffer], page ⟨undefined⟩ for all details.

IMPORTANT: First a call of `tree-buffer-update` updates the **display** of a tree-buffer, means shows all the tree-nodes in an emacs-buffer! Neither creating a tree-buffer nor adding tree-nodes display anything; this just builds the internal tree-structure.

IMPORTANT: See ⟨undefined⟩ [Programming special windows], page ⟨undefined⟩ for details about programming interactors (special windows) regardless if they were build as tree or not. There you can find a.o. how to automatically synchronizing a special window with the current edit-buffer.

11.3.2 How to create a new tree-buffer

The creator-function for a new tree-buffer depends on the fact if the new tree-buffer should be used as an ECB-interactor or not. For a new ECB-interactor the macro `defecb-tree-buffer-creator` has to be used, otherwise the function `tree-buffer-create`. In the end both methods use `tree-buffer-create` because the BODY-argument of `defecb-tree-buffer-creator` must contain a call to this function!.

This section describes all arguments of `tree-buffer-create`.

Except the first argument *NAME* all arguments are key-arguments of the form :arg-name arg-value, so for example a call looks like

```
(tree-buffer-create <buffer-name> :frame <frame-object> ...).
```

These key-arguments (all except the first argument `NAME`) can be arranged in any arbitrary order but all of them are not-optional! The key-arg-name is always a : followed by the lowercase version of the mentioned argument below (e.g. `FRAME` –> :frame, `MOUSE-ACTION-TRIGGER` –> :mouse-action-trigger).

Here is a description of the arguments of `tree-buffer-create` - also available as docstring for this function (via *C-h f*). The description below contains also some examples for complex-arguments!

`NAME` Buffername of the new tree-buffer.

`FRAME` Frame in which the tree-buffer is displayed and valid. All key-bindings and interactive functions of the tree-buffer work only if called in *FRAME* otherwise nothing is done!

`MOUSE-ACTION-TRIGGER`
When a mouse-action is triggered. Allowed values: `button-release` and `button-press`.

`IS-CLICK-VALID-FN`
`tree-buffer-create` rebinds *`mouse-1`*, *`mouse-2`*, *`RET`* (and *`TAB`*) and also in combination with shift and control (not with *`TAB`*). *IS-CLICK-VALID-FN* is called first if a node or an expand-symbol is clicked. This function is called with five arguments:

- mouse-button: The clicked mouse-button or RET or TAB (0 = RET or TAB, 1 = mouse-1, 2 = mouse 2)
- shift-pressed: Non nil if the SHIFT-key was pressed during mouse-click or RET.
- control-pressed: Non nil if the CONTROL-key was pressed during mouse-click or RET.
- meta-pressed: Non nil if the META-key was pressed during mouse-click or RET.
- tree-buffer-name: The buffer-name of the tree-buffer where the node has been clicked.

The function must return not nil iff exactly this click/hit is accepted. If the function returns nil then really nothing is done by the tree-buffer after this click/hit!

Here is an example (call *`C-h f`* to see what it does) for this callback-function:

```
(defun ecb-interpret-mouse-click (mouse-button
                                  shift-pressed
                                  control-pressed
                                  meta-pressed
                                  tree-buffer-name)
  (if (eq mouse-button 0)
      (list (if control-pressed 2 1) shift-pressed meta-pressed
            'keyboard)
    (if (and (not (eq mouse-button 1)) (not (eq mouse-button 2)))
nil
      (case ecb-primary-secondary-mouse-buttons
        (mouse-1--mouse-2
         (if control-pressed
             nil
           (list mouse-button shift-pressed meta-pressed 'mouse)))
        (mouse-1--C-mouse-1
         (if (not (eq mouse-button 1))
             nil
           (list (if control-pressed 2 1) shift-pressed meta-pressed
                 'mouse)))
        (mouse-2--C-mouse-2
         (if (not (eq mouse-button 2))
             nil
           (list (if control-pressed 2 1) shift-pressed meta-pressed
                 'mouse)))
        (otherwise nil)))))
```

This example would be passed as parameter as follows:

```
(tree-buffer-create "myname"
                    :is-click-valid-fn 'ecb-interpret-mouse-click
                    ...)
```

NODE-SELECTED-FN

Function to call if a node has been selected. This function is called with the following parameters:

- node: The selected node
- mouse-button (0 = RET, 1 = mouse-1, 2 = mouse 2)
- shift-pressed
- control-pressed
- meta-pressed
- tree-buffer-name

For the last four arguments see the description above. This function has to ensure that the expandable- and expanded-state of the selected node is correct after returning.

Here is an example (call `C-h f` to see what it does) for this callback-function:

```
(defun ecb-tree-buffer-node-select-callback (node
     mouse-button
     shift-pressed
     control-pressed
                                                meta-pressed
     tree-buffer-name)
  (let* ((ecb-button-list (ecb-interpret-mouse-click mouse-button
     shift-pressed
     control-pressed
                                                        meta-pressed
     tree-buffer-name))
 (ecb-button (nth 0 ecb-button-list))
 (shift-mode (nth 1 ecb-button-list))
         (meta-mode (nth 2 ecb-button-list))
         (keyboard-p (equal (nth 3 ecb-button-list) 'keyboard))
         (maximized-p (ecb-buffer-is-maximized-p tree-buffer-name)))
    ;; now we dispatch to the right action
    (when ecb-button-list
      (cond ((ecb-string= tree-buffer-name ecb-directories-buffer-name)
       (ecb-directory-clicked node ecb-button nil shift-mode
                                    meta-mode))
      ((ecb-string= tree-buffer-name ecb-sources-buffer-name)
       (ecb-source-clicked node ecb-button nil shift-mode
                                  meta-mode))
      ((ecb-string= tree-buffer-name ecb-history-buffer-name)
       (ecb-history-clicked node ecb-button nil shift-mode
                                   meta-mode))
      ((ecb-string= tree-buffer-name ecb-methods-buffer-name)
       (ecb-method-clicked node ecb-button nil shift-mode
                                  meta-mode))
      ((ecb-string= tree-buffer-name ecb-analyse-buffer-name)
       (ecb-analyse-node-clicked node ecb-button nil shift-mode
                                       meta-mode))
      (t nil)))))
```

This example would be passed as parameter as follows:

```
(tree-buffer-create "myname"
                    :node-selected-fn 'ecb-tree-buffer-node-select-callba
                    ...)
```

IMPORTANT: This callback must not modify the slot *EXPANDED* of the passed node because this is done automatically by the tree-buffer-library!

NODE-EXPANDED-FN

Function to call if a node is expandable, point stays onto the expand-symbol and node is not already expanded. This function is called with the following parameters:

– node: The selected node

- mouse-button (0 = TAB, 1 = mouse-1, 2 = mouse 2)
- shift-pressed
- control-pressed
- meta-pressed
- tree-buffer-name

This function should add all children nodes to this node if not already done (if possible). This function has to ensure that the expandable- and expanded state of the selected node is correct after returning!

IMPORTANT: This callback must not modify the slot *EXPANDED* of the passed node because this is done automatically by the tree-buffer-library!

NODE-COLLAPSED-FN

Function to call if a node is expandable, point stays onto the expand-symbol and node is already expanded. This function is called with the following parameters:

- node: The selected node
- mouse-button (0 = TAB, 1 = mouse-1, 2 = mouse 2)
- shift-pressed
- control-pressed
- meta-pressed
- tree-buffer-name

This function is only a callback to inform the owner/user of this tree-buffer that this node has been collapsed. This function must not modify the expandable- or expanded state of the selected node!

Often a sensefull value for this parameter is the function `ignore`.

IMPORTANT: This callback must not modify the slot *EXPANDED* of the passed node because this is done automatically by the tree-buffer-library!

NODE-MOUSE-OVER-FN

Function to call when the mouse is moved over a node. This function is called with three arguments: NODE, WINDOW, NO-PRINT, each of them related to the current tree-buffer. If NO-PRINT is nil then the function must print the text itself in any manner. This function must always return the text which either is printed by the function itself or by the caller (if NO-PRINT is not nil). The current buffer for this function is the tree-buffer itself. With XEmacs this function is only called if the tree-buffer track-mouse mechanism is activated (see the function `tree-buffer-activate-follow-mouse`). With GNU Emacs >= 21 this function is called by the `help-echo` property added to each node.

Here is an example (call *C-h f* to see what it does) for this callback-function:

```
(defun ecb-mouse-over-analyse-node (node &optional window no-message
                                         click-force)
  (let ((str (when (or click-force
                       (ecb-show-minibuffer-info
                        node window
                        (car ecb-analyse-show-node-info)))
               (if (equal (cdr ecb-analyse-show-node-info)
                          'full-info)
                   (ecb-analyse-gen-tag-info
                    (car (tree-node->data node)))
                 (tree-node->name node)))))
    (prog1 str
      (unless no-message
        (ecb-nolog-message str)))))
```

MOUSE-HIGHLIGHT-FN

If nil then in this tree-buffer no node is highlighted when the mouse moves over it. If t then each node is highlighted when the mouse moves over it. If a function then it is called with the node as argument and if it returns not nil then the node will be highlighted when the mouse moves over it - otherwise no highlighting takes place.

NODE-DATA-EQUAL-FN

Function used by the tree-buffer to test if the data of two tree-nodes are equal. The function is called with two args: The DATA-slots of the two tree-nodes (see ⟨undefined⟩ [A new tree-node], page ⟨undefined⟩ for details about the data-slots).

Here is an example (call *C-h f* to see what it does) for this callback-function:

```
(defun ecb-analyse-compare-node-data (left right)
  "Return not nil when LEFT and RIGHT are identical node-datas."
  (and (equal (nth 2 left) (nth 2 right))
       (ecb-compare-methods-buffer-node-data (car left)
                                             (car right))))
```

Often a suitable value for this parameter is `equal`.

MAYBE-EMPTY-NODE-TYPES

Nil or a list of node-types (a node-type is an integer which must be set with `tree-node-new`). Nodes with one of these types are treated as empty if they are not expandable (i.e. they have no children) and will be displayed with the empty-symbol ([x]); for other nodes see next argument.

LEAF-NODE-TYPES

Nil or a list of node-types (see above). Nodes with one of these types are treated as leafs and will be displayed with the leaf-symbol (*).

Summary for *MAYBE-EMPTY-NODE-TYPES* and *LEAF-NODE-TYPES*:

- Expandable nodes will always be displayed either with the open- or with the close-symbol.

- Not-expandable nodes with a node-type contained in *MAYBE-EMPTY-NODE-TYPES* will be displayed with the empty-symbol.
- Not-expandable nodes with a node-type contained in *LEAF-NODE-TYPES* will be displayed with the leaf-symbol.
- All other nodes will be displayed with no symbol just with correct indentation.

MENU-CREATOR

Nil or function which has to return nil or a list of conses, each cons for a known node-type of this tree-buffer (the node-type of a node is an integer). Example: ((0 . menu-for-type-0) (1 . menu-for-type-1)). The cdr of a cons must be a menu in the same format `tree-buffer-create-menu` expects as argument - see the documentation of this function for details. This function gets two arguments: The name of the tree-buffer and the node for which a popup-menu should be opened.

Here is an example for such a menu-creator-callback:

```
(defconst ecb-analyse-nodedata-tag-with-pos 0)
(defconst ecb-analyse-nodedata-tag-without-pos 1)
(defconst ecb-analyse-nodedata-no-tag 2)

(defconst ecb-analyse-nodetype-bucket 0)
(defconst ecb-analyse-nodetype-context 1)
(defconst ecb-analyse-nodetype-arguments 2)
(defconst ecb-analyse-nodetype-completions 3)
(defconst ecb-analyse-nodetype-localvars 4)
(defconst ecb-analyse-nodetype-prefix 5)
(defconst ecb-analyse-nodetype-assignee 6)
(defconst ecb-analyse-nodetype-function 7)
(defconst ecb-analyse-nodetype-function-arg 8)

(defun ecb-analyse-create-menu (node)
  "Return a popup-menu suitable for NODE."
  (let* ((data (tree-node->data node))
         (tag-p (not (equal (nth 1 data) ecb-analyse-nodedata-no-tag)))
         (tag-with-pos-p (equal (nth 1 data)
                                ecb-analyse-nodedata-tag-with-pos))
         (nodetype (nth 2 data)))
    (delq nil (list (if (equal nodetype ecb-analyse-nodetype-completions)
                        '(ecb-analyse-complete "Complete"))
                    (if tag-p
                        '(ecb-analyse-show-tag-info "Show tag info"))
                    (if tag-with-pos-p
                        '(ecb-analyse-jump-to-tag "Jump to tag"))))))

(defun ecb-analyse-menu-creator (tree-buffer-name node)
  "Creates the popup-menus for the analyse-buffer."
  (let ((nodetype (tree-node->type node)))
    (unless (equal nodetype ecb-analyse-nodetype-bucket)
      (mapcar (function (lambda (type)
                          (cons type (ecb-analyse-create-menu node))))
              `(,ecb-analyse-nodetype-context
                ,ecb-analyse-nodetype-arguments
                ,ecb-analyse-nodetype-completions
                ,ecb-analyse-nodetype-localvars
                ,ecb-analyse-nodetype-prefix
                ,ecb-analyse-nodetype-assignee
                ,ecb-analyse-nodetype-function
                ,ecb-analyse-nodetype-function-arg)))))
```

This example would be passed as parameter as follows:

```
(tree-buffer-create "myname"
                    :menu-creator 'ecb-analyse-menu-creator
                    ...)
```

MENU-TITLES
: Nil or a list conses, each cons for a node-type. See *MENU-CREATOR*. The cdr of a cons must be either a string or a function which will be called with current node under point and must return a string which is displayed as the menu-title.

MODELINE-MENU-CREATOR
: Nil or a function which has to return nil or a menu in the same format `tree-buffer-create-menu` expects as argument - see the documentation of this function for details. This function gets one argument: The name of the tree-buffer. If the function returns a menu then this menu will be displayed when the user clicks with mouse-button 3 at the modeline of the tree-buffer. The menu-title will be "Tree-buffer modeline-menu".

TRUNC-LINES
: Should lines in this tree buffer be truncated (not nil).

READ-ONLY
: Should the treebuffer be read-only (not nil).

TREE-INDENT
: Spaces subnodes should be indented. Ignored if *TREE-STYLE* is `image` (see below).

INCR-SEARCH-P
: Should the incremental search be enabled in the tree-buffer. Three choices: `prefix`, `substring`, `nil`. See the command `tree-buffer-incremental-node-search`.

INCR-SEARCH-ADDITIONAL-PATTERN
: Every search-pattern is prefixed with a regexp to jump over not important stuff of a displayed node-name at incr. search.. This is per default: beginning spaces and guide characters (|`-) and all expand/collapse-buttons [+], [x], rsp. [-]!

  If this argument is not nil then it must be a cons-cell where car is a string which should be a regexp-pattern which is added to the basic-prefix pattern (see above) and both of them prefix the incr-search-pattern. The cdr is the number of subexpressions in this additoonal pattern.

ARROW-NAVIGATION
: If not nil then a smart navigation with arrow keys is offered:

  - Left-arrow: If node is expanded then it will be collapsed otherwise point jumps to the next "higher" node in the hierarchical tree (higher means the next higher tree-level or - if no higher level available - the next higher node on the same level).
  - Right-arrow: If node is not expanded then it will be expanded.

    Onto a not expandable node the horizontal arrow-keys go one character in the senseful correct direction.
  - Up- and down-key: Point jumps to the first character of the previous (up) rsp. next node (down). "First" character means either the first character of the expand-symbol (in case *EXPAND-SYMBOL-BEFORE-P* is not

nil) or of the displayed node-name. Or with other words: The first non-indentation and non-guide-line (see *TREE-STYLE*) character of a node.

HOR-SCROLL-STEP
: Number of columns a hor. scroll in the tree-buffer should scroll. If not nil then *M-mouse-1* and *M-mouse-2* scroll left and right and also *M-<left-arrow>* and *M-<right-arrow>*. Ignored with XEmacs.

DEFAULT-IMAGES-DIR
: Full path where the default images for the tree-buffer can be found. It should contain an image for every name of `tree-buffer-tree-image-names`.

ADDITIONAL-IMAGES-DIR
: Additional image-dir which should be searched first for images needed for current tree-buffer. If the image can not be found in this directory then *DEFAULT-IMAGES-DIR* is searched. If the image can't even found here the related ascii-symbol is used.

IMAGE-FILE-PREFIX
: Common prefix for all image-files for this tree-buffer, e.g. "ecb-".

TREE-STYLE
: There are three different styles available: Image-style (value `image`): Very nice and modern because image-icons are used to display the tree-buffer. For this style the arguments *TREE-INDENT* and *EXPAND-SYMBOL-BEFORE-P* have no effect.

  Ascii-style with guide-lines (value `ascii-guides`) and ascii-style without guide-lines (value `ascii-no-guides`. See ⟨undefined⟩ [Tree-buffer styles], page ⟨undefined⟩ for details about the tree-styles.

  Both ascii-styles are affected by the args *TREE-INDENT* and *EXPAND-SYMBOL-BEFORE-P*.

ASCII-GUIDE-FACE
: If *TREE-STYLE* is `ascii-guides` then this defines the face the guides should be displayed with.

TYPE-FACER`:`
: Nil or a list of one or more conses, each cons for a node-type (a node-type is an integer which must be set with `tree-node-new`). The cdr of a cons can be:

  – a face-symbol
  – a function-symbol which gets two arguments (see `tree-buffer-insert-text`). This function can do anything, but normally it should face a tree-node.
  – the symbol t. Then the tree-buffer assumes that the node-text is already faced and therefore it does not face the node, means it does nothing then inserting the node-text, if the tree-buffer is updated.

EXPAND-SYMBOL-BEFORE-P
: If not nil then the expand-symbol is displayed before the node-text. Ignored when *TREE-STYLE* is `image` and Emacs can display images.

HIGHLIGHT-NODE-FACE
: Face used for highlighting current selected node in this tree-buffer.

GENERAL-FACE
: General face in which the whole tree-buffer should be displayed.

AFTER-CREATE-HOOK:
: A function or a list of functions (with no arguments) called directly after creating the tree-buffer and defining it's local keymap. For example such a function can add additional key-bindings for this tree-buffer local keymap (use `local-set-key` for this).

  Here is an example for such a hook:

```
(defun ecb-common-after-tree-buffer-create-actions ()
  "Things which should be performed after creating a tree-buffer.
The tree-buffer is the current buffer."
  (local-set-key (kbd "C-t")
                 'ecb-toggle-do-not-leave-window-after-select)
  (if ecb-running-xemacs
      (define-key modeline-map
        '(button2up)
        'ecb-toggle-maximize-ecb-window-with-mouse)
    (local-set-key [mode-line mouse-2]
                   'ecb-toggle-maximize-ecb-window-with-mouse)))
```

AFTER-UPDATE-HOOK:
: A function or a list of functions (with no arguments) called each time after the tree-buffer has been updated via `tree-buffer-update`.

Here is an example how to create a tree-buffer (if you want a tree-buffer not for ECB then just strip off the `defecb-tree-buffer-creator` and just call `tree-buffer-create`):

```
(defecb-tree-buffer-creator ecb-create-analyse-tree-buffer
    ecb-analyse-buffer-name
  "Create the tree-buffer for analyse-display."
  (tree-buffer-create
   ecb-analyse-buffer-name
   :frame ecb-frame
   :mouse-action-trigger ecb-tree-mouse-action-trigger
   :is-click-valid-fn 'ecb-interpret-mouse-click
   :node-selected-fn 'ecb-tree-buffer-node-select-callback
   :node-expanded-fn 'ecb-tree-buffer-node-expand-callback
   :node-collapsed-fn 'ecb-tree-buffer-node-collapsed-callback
   :node-mouse-over-fn 'ecb-mouse-over-analyse-node
   :mouse-highlight-fn 'ecb-analyse-node-mouse-highlighted-p
   :node-data-equal-fn 'ecb-analyse-compare-node-data
   :maybe-empty-node-types nil
   :leaf-node-types nil
   :menu-creator 'ecb-analyse-menu-creator
   :menu-titles (ecb-analyse-gen-menu-title-creator)
   :modeline-menu-creator 'ecb-common-tree-buffer-modeline-menu-creator
   :trunc-lines (ecb-member-of-symbol/value-list
                 ecb-analyse-buffer-name
                 ecb-tree-truncate-lines)
   :read-only t
   :tree-indent ecb-tree-indent
   :incr-search-p t
   :incr-search-additional-pattern nil
   :arrow-navigation ecb-tree-navigation-by-arrow
   :hor-scroll-step ecb-tree-easy-hor-scroll
   :default-images-dir (car ecb-tree-image-icons-directories)
   :additional-images-dir (ecb-member-of-symbol/value-list
                           ecb-analyse-buffer-name
                           (cdr ecb-tree-image-icons-directories)
                           'car 'cdr)
   :image-file-prefix "ecb-"
   :tree-style ecb-tree-buffer-style
   :ascii-guide-face ecb-tree-guide-line-face
   :type-facer nil
   :expand-symbol-before-p ecb-tree-expand-symbol-before
   :highlight-node-face ecb-analyse-face
   :general-face ecb-analyse-general-face
   :after-create-hook (append
                       (list (lambda ()
                               (ecb-common-after-create-actions)))
                       ecb-common-tree-buffer-after-create-hook
                       ecb-analyse-buffer-after-create-hook)
   :after-update-hook nil))
```

11.3.3 How to create a new tree-node

When a new tree-buffer has been created, then the most senseful programming-task is adding some tree-nodes to it.

11.3.3.1 Content of a tree-node

A tree-node is an object which stores in special *slots* several data necessary to link the node with other nodes, to display the node and to hold some associated node-data (e.g. a tag created by the semantic-library).

A tree-node can have the following slots:

NAME The name of the node. Regardless how the node is displayed; see *SHRINK-NAME* and *DISPLAYED-NAME*.

TYPE The type of the node; must currently be an interger! The type is used to classify the nodes, so for example all nodes of a certain type can display the same popup-menu - see `tree-buffer-create` or ⟨undefined⟩ [A new tree-buffer], page ⟨undefined⟩ which parts of a tree-buffer are distinguished by node-types.

DATA The data of the node; This can be any arbitrary emacs-lisp-object. This slots holds that data asscociated with the node and represented by the node in the tree-buffer. Example: Assume a tree-buffer displaying a directory-tree where each node just displays as its name the name of (sub)directories, but not the full path. The full path is stored in the *DATA*-slot of a node so when clicking onto this node the asscociated directory can be open for example in a dired-buffer.

EXPANDABLE
If not nil then the node is expandable means it has children.

EXPANDED If not nil then the node is currently expanded, means its children are visible in the tree-buffers as subnodes of the node.

PARENT The parent tree-node. This is the link to the father (rsp. mother ;-) of the node. It must be a object of type tree-node!

CHILDREN List of children tree-nodes. They must be all objects of type tree-node!

SHRINK-NAME
Decides if the *NAME* can be shortened when displayed in a narrow tree buffer window. The following values are valid:

- `beginning`: The *NAME* is truncated at the beginning so the end is always visible.
- `end`: The *NAME* is truncated at the end. If the tree-node is EXPANDABLE the name is truncated so that the expand symbol is visible.
- `nil`: The *NAME* is never truncated. In this case *DISPLAYED-NAME* is equal to *NAME*.

INDENTSTR
Containes the full indentation-string for the node. So a single node can easily be redrawn.

DISPLAYED-NAME
: Contains the current displayed name of the node. The displayed name can be different from the *NAME* according to the value of *SHRINK-NAME*.

11.3.3.2 Creating a new tree-node and adding it to the tree

A new tree-node has to be created with the function `tree-node-new`. This "constructor" accepts the following parameter: *NAME*, *TYPE*, *DATA*, *NOT-EXPANDABLE*, *PARENT* and *SHRINK-NAME*.

For all parameters except *NOT-EXPANDABLE* the description is available in the slot-description in the section above. If *NOT-EXPANDABLE* is set to not nil then the slot *EXPANDABLE* will be set to `nil`; otherwise to `t`.

`tree-node-new` returns a new tree-node.

The new node can either being added implicitely to the tree via the optional *PARENT*-parameter when calling `tree-buffer-new` or explicitely by first creating the new node without setting the parent-node but later setting the parent-node via the according accessor (see next section below). Children should only being added with `tree-node-add-children` - see next section.

11.3.3.3 Accessing the slots of a tree-node

The section above shows which slots a tree-node have.

A slot with name XXX is getable with the following piece of code:

```
(tree-node->xxx <a tree node>)
```

Here is an example how to get the value of the slot *DATA*:

```
(tree-node->data <a tree node>)
```

A slot with name XXX is setable with the following piece of code:

```
(setf (tree-node->xxx <a tree node>) <new value>)
```

Again an example with slot *DATA* which sets this slot to the string "~/a_subdir_of_HOME":

```
(setf (tree-node->data <a tree node>) "~/a_subdir_of_HOME")
```

IMPORTANT: Adding new children to a node should always being done with the function `tree-node-add-children` because this functions encapsulates all the necessary stuff needed to add children to a node (mainly adding the children itself and setting the node itself as parent for every children).

See ⟨undefined⟩ [The tree-buffer-API], page ⟨undefined⟩ for the rest of the API available for tree-nodes.

11.3.4 How to update a tree-buffer-display after changes

When finished with adding tree-nodes to the tree-structure you mostly want to display the current tree and its state in the buffer/window so a user can see the current tree and can use it.

Threre are two ways to update a tree-buffer for display:

1. Updating the whole tree-buffer:

   This is the most used way to update the tree-buffer display. It's quite simple, just call `tree-buffer-update`. In most cases you want to call it without arguments.

   If you want to display a certain expanded node and as much as possible subnodes of this node then pass this node-object as first argument to `tree-buffer-update`.

   If you do not have the need to display a completely new tree-structure but you want only to display a previously cached display-state then pass this cached-state as second argument to `tree-buffer-update`. See the documentation of this function and also ⟨undefined⟩ [Tree-buffer How to], page ⟨undefined⟩ for a detailled description how to do this.

2. Updating only a single node of the tree-buffer:

   Sometimes it can be useful to update only exactly one special node, e.g. when your application codes some node-state in the displayed node-name (e.g. ECB displays the version-control state of a file as part of the related node-name) then it is necessary to update only this node if the state has changed.

   This can be done with the function `tree-buffer-update-node`. For this function the mentioning in this section can be misleading because this function can not only update the node-display but in general the slots *NAME*, *SHRINK-NAME*, *TYPE*, *DATA* and *EXPANDABLE*. Do `C-h f` to see the documentation of this function for all details!

11.3.5 Default and customizable keybindings of a tree-buffer

When creating a tree-buffer with `tree-buffer-create` the following keys will automatically being bound:

delete
backspace
home
end
a (and each other key bound to `self-insert-command`)
: All of these keys are bound to the command `tree-buffer-incremental-node-search` if the argument *INCR-SEARCH-P* of `tree-buffer-create` was set to not nil. See the documentation of `tree-buffer-incremental-node-search` for all details.

RET
C-RET
S-RET
M-RET
C-S-RET
mouse-1
C-mouse-1
S-mouse-1
M-mouse-1
mouse-2
C-mouse-2
S-mouse-2
M-mouse-2

All these keys are bound to an action-dispatcher which works as follows:

If the callback-function in slot *IS-CLICK-VALID-FN* of the tree-buffer (see ⟨undefined⟩ [A new tree-buffer], page ⟨undefined⟩) returns nil then nothing is done.

If either *RET* has been hitted or point is as the node-name (i.e. the user has clicked with the mouse-1/2 at the node-name) then the callback-function in slot *NODE-SELECTED-FN* is called with the needed arguments (see ⟨undefined⟩ [A new tree-buffer], page ⟨undefined⟩).

If point is at the expand/collape-button then depending on the expansion-state of the node either the callback in slot *NODE-EXPANDED-FN* or *NODE-COLLAPSED-FN* is called (for parameters see again ⟨undefined⟩ [A new tree-buffer], page ⟨undefined⟩).

IMPORTANT: None of these callbacks must modify the slot *EXPANDED* of the passed node because this is done automatically by the action-dispatcher!

At the end the dispatcher updates the tree-buffer-display with optimized display of the clicked/selected node - see ⟨undefined⟩ [Updating a tree-buffer], page ⟨undefined⟩. This means `tree-buffer-update` is called with that node as argument.

TAB Depending on the expansion-state of the node either the callback in slot *NODE-EXPANDED-FN* or *NODE-COLLAPSED-FN* is called (for parameters see again ⟨undefined⟩ [A new tree-buffer], page ⟨undefined⟩).

IMPORTANT: None of these callbacks must modify the slot *EXPANDED* of the passed node because this is done automatically by the action-dispatcher!

At the end the the tree-buffer-display is updated with optimized display of the clicked/selected node - see ⟨undefined⟩ [Updating a tree-buffer], page ⟨undefined⟩. This means `tree-buffer-update` is called with that node as argument.

mouse-3 Activates the popup-menu for the current tree-buffer for current node-type (if defined). See ⟨undefined⟩ [A new tree-buffer], page ⟨undefined⟩ at argument *MENU-CREATOR* and *MENU-TITLES*. These callbacks are called for getting the menu and the menu-title.

modeline-mouse-3
: Activates the popup-menu for the modeline of the current tree-buffer (if defined). See ⟨undefined⟩ [A new tree-buffer], page ⟨undefined⟩ at argument *MODELINE-MENU-CREATOR*. This callback is called for getting the modeline-menu.

M-m
: This key is bound to the command `tree-buffer-show-node-menu-keyboard`: Activates the popup-menu of current tree-buffer for current node-type via keyboard. If called with a prefix-arg then the library '`tmm.el`' is used for displaying the popup-menu - ignored with XEmacs.

<up>
<down>
<left>
<right>
: These keys are bound to the command `tree-buffer-arrow-pressed` which implements the smart arrow-key-navigation described in ⟨undefined⟩ [A new tree-buffer], page ⟨undefined⟩ at argument *ARROW-NAVIGATION*.

In addition to these automatically bound keys you can add further keybindings to the local-keymap of the tree-buffer with the parameter *AFTER-CREATE-HOOK* of `tree-buffer-create`. See ⟨undefined⟩ [A new tree-buffer], page ⟨undefined⟩ for an example which binds *C-t* in this hook.

11.3.6 All functions available for tree-buffers and tree-nodes

This chapter lists the complete AI available for tree-buffers and tree-nodes.

IMPORTANT: These are the only functions and macros of tree-buffer.el you are allowed to use for programming with tree-buffers and tree-nodes. If you use other - not here listed - functions, macros or variables of tree-buffer.el then you run the risk of unwanted side-effects or program-behaviors!

11.3.6.1 The API for a tree-buffer:

See the documentation of these functions (e.g. via *C-h f*) to get the details how to use it.

- `tree-buffer-add-image-icon-maybe`
- `tree-buffer-find-image`
- `tree-buffer-create`[2]
- `tree-buffer-defpopup-command`
- `tree-buffer-destroy`[3]
- `tree-buffer-empty-p`
- `tree-buffer-expand-node`
- `tree-buffer-get-node-at-point`
- `tree-buffer-node-data-equal-p`
- `tree-buffer-recenter`

[2] If the tree-buffer should be used by ECB then you must use `defecb-tree-buffer-creator` - see the documentation!

[3] Not needed when `defecb-tree-buffer-creator` has been used for creation.

- `tree-buffer-highlight-node-data`
- `tree-buffer-remove-highlight`
- `tree-buffer-remove-node`
- `tree-buffer-clear-tree`
- `tree-buffer-displayed-nodes-copy`
- `tree-buffer-search-displayed-node-list`
- `tree-buffer-number-of-displayed-nodes`
- `tree-buffer-get-data-store`
- `tree-buffer-set-data-store`
- `tree-buffer-get-root`
- `tree-buffer-set-root`
- `tree-buffer-update`
- `tree-buffer-update-node`

11.3.6.2 The API for a tree-node

See the documentation of these functions (e.g. via *C-h f*) to get the details how to use it.

- `tree-node-add-children`
- `tree-node-linelength`
- `tree-node-new`
- `tree-node-new-root`
- `tree-node-remove-child`
- `tree-node-remove-child-by-data`
- `tree-node-find-child-by-data`
- `tree-node-find-child-by-name`
- `tree-node-search-subtree-by-data`
- `tree-node-sort-children`
- `tree-node-toggle-expanded`

In addition to these functions the tree-node API contains all accessors for a tree-node which are described in ⟨undefined⟩ [A new tree-node], page ⟨undefined⟩.

11.3.7 Things which are strictly forbidden

Variable `tree-buffers`: Only for internal use! It contains all tree-buffers of current Emacs-instance, means **all** tree-buffers of **all** applications which uses currently tree-buffers. Every application must store its own collection of tree-buffers in an own variable! For example: ECB stores its tree-buffer set in `ecb-tree-buffers`!

Variable `tree-buffer-displayed-nodes`: Only for internal use! Contains all the current visible nodes of current tree-buffer in top-to-bottom order. This variable is buffer-local in each tree-buffer! Do not use it directly! When you want to cache the current display, then see ⟨undefined⟩ [Tree-buffer How to], page ⟨undefined⟩ how to do this.

IMPORTANT: An application may only use the API tree-buffer.el provides but no internal variables - see ⟨undefined⟩ [The tree-buffer-API], page ⟨undefined⟩!

11.3.8 How to deal with certain programming-requirements

This chapter describes in detail how to solve certain programming-challenges with tree-buffers.

11.3.8.1 Caching the current tree-buffer display

Sometimes it can be useful or important to cache the current display of a tree-buffer and display later exactly this cached display-state. Here is how to do this:

1. Caching the display: You have to do two tasks: First store the current internal structure of the tree-buffer; you must do this with the function `tree-buffer-displayed-nodes-copy`. Then store the buffer-contents of that tree-buffer you want to cache; you can do this for example with `buffer-substring`. For both tasks you must make the tree-buffer the current-buffer.
2. Displaying a previous tree-buffer-cache: Make the tree-buffer the current buffer, call `tree-buffer-update` and pass as second argument *CONTENT* the data you have stored in step 1. See the documentation of `tree-buffer-update` for details.

   Here is an example:

```
(tree-buffer-update
  nil
  (cons (nth 2 cache-elem) ;; the stored buffer-string
        (nth 1 cache-elem) ;; the stored tree-structure
  )))
```

11.4 How to deal with the adviced functions

ECB needs a bunch of advices so ECB can work correctly. ECB has a powerful advice-backbone which allows defining sets of adviced functions whereas a set means, that all advices of a certain set are always enabled or disabled together.

For this ECB contains three macros:

- `defecb-advice-set`
- `defecb-advice`
- `ecb-with-original-adviced-function-set`

For a detailed explanation of each macro read the documentation with `describe-function`!

An advice in ECB must not being defined by `defadvice` but only with `defecb-advice` which in turn needs a previously defined advice-set defined by `defecb-advice-set`.

So ECB has always full control of all advices. For example ECB automatically disables all advices of all advice-sets at load-time of ECB and also at deactivation time of ECB. So you can be sure that after deactivating ECB all ecb-advices are deactivated/disabled too.

In addition to the three macros above ECB offers two further macros for running code with disabled some ecb-advices:

- `ecb-with-original-basic-functions`
- `ecb-with-original-permanent-functions`

The advice set `ecb-basic-adviced-functions` contains most of the ecb-advices. See the contents of this variable to see which advices are contained in this set. Use `ecb-with-original-basic-functions` when you want evaluating elisp-code with disabled basic-advices. `ecb-with-original-basic-functions` is only a shortcut for `ecb-with-original-adviced-function-set` called for the `ecb-basic-adviced-functions`-set.

Same for `ecb-with-original-permanent-functions` which is a shortcut for the advice-set `ecb-permanent-adviced-functions`.

Last but not least ECB contains an advice-set `ecb-always-disabled-advices`. These advices are always disabled. This advice-set can not be enabled by `ecb-enable-advices` but such an advice has to be activated 'on demand' by the caller. Such an advice must be used with the macro `ecb-with-ecb-advice` (see the docstring).

ECB contains some more advice-sets but don't bother about it.

11.5 How to program new layouts and new special windows

There are two aspects concerning this topic:

1. Programming a new layout which contains several special ECB-windows like directories, sources, methods, history or other special windows and arranging them in a new outline.
2. Creating complete new special windows (e.g. a local-variable window for a graphical debugger like JDEbug of JDEE), adding them to a layout and synchronizing them with the current active edit-window.

The former one covers merely the layout-programming aspect which is explained in the first subsection of this chapter whereas the latter one covers all aspects of creating new special windows and what is necessary to synchronize it with the current active edit-window of ECB. This is explained in the second subsection which will refers to the first subsection.

11.5.1 How to program a new layout

If you just want creating a new layout with the standard ECB-windows like directories, sources, methods, history and speedbar it's is strongly recommended to define the new layout interactively with the command `ecb-create-new-layout` (see ⟨undefined⟩ [Creating a new ECB-layout], page ⟨undefined⟩).

If you want creating a new layout and if this layout should contain other special windows than the standard ECB-windows then it's still recommended to define this layout interactively with `ecb-create-new-layout` and using the option to give the created windows user-defined types. For every user defined type you have then just to program the necessary buffer-set function. For all the details see ⟨undefined⟩ [Creating a new ECB-layout], page ⟨undefined⟩.

But if you do not like the interactive way (because you are tough and brave) but you want programming the new layout with Elisp then use the macro `ecb-layout-define` (the following definition has stripped the prefix "ecb-" for better indexing this manual):

`layout-define` *name type doc* **&rest** *create-code* [Macro]

Creates a new ECB-layout with name *NAME*. *TYPE* is the type of the new layout and is literal, i.e. not evaluated. It can be left, right, top or left-right. *DOC* is the docstring for the new layout-function "ecb-layout-function-<name>". *CREATE-CODE* is all the lisp code which is necessary to define the ECB-windows/buffers.

This macro adds the layout with *NAME* and *TYPE*to the internal variable `ecb-available-layouts`.

Preconditions for *CREATE-CODE*:

1. Current frame is splitted at least in one edit-window and the "column" (for layout types left, right and left-right) rsp. "row" (for a top layout) for the special ECB-windows/buffers. The width of the "column" rsp. the height of the "row" is always defined with the option `ecb-windows-width` rsp. `ecb-windows-height`. Depending on the value of the option `ecb-compile-window-height` there is also a compile window at the bottom of the frame which is stored in `ecb-compile-window`.
2. All windows are not dedicated.
3. Neither the edit-window nor the compile-window (if there is one) are selected for types left, right and top. For type left-right the left column-window is selected
4. All ECB-advices of the advice-sets `ecb-basic-adviced-functions` and `ecb-permanent-adviced-functions` are disabled.

Things *CREATE-CODE* has to do:

1. Splitting the ECB-tree-windows-column(s)/row (s.a.) in all the ECB-windows the layout should contain (directories, sources, methods and history). The split must not be done with other functions than `ecb-split-hor` and `ecb-split-ver`! It is recommended not to to use a "hard" number of split-lines or -rows but using fractions between -0.9 and +0.9! Tip: It is recommended to spilt from right to left and from bottom to top or with other words: First create the right-most and bottom-most special windows!
2. Making each special ECB-window a dedicated window. This can be done with one of the following functions:
   - `ecb-set-directories-buffer`
   - `ecb-set-sources-buffer`
   - `ecb-set-methods-buffer`
   - `ecb-set-history-buffer`
   - `ecb-set-speedbar-buffer`

   Each layout can only contain one of each tree-buffer-type!

   In addition to these functions there is a general macro: `defecb-window-dedicator`: This macro defines a so called "window-dedicator" which is a function registered at ECB and called by ECB to perform any arbitrary code in current window and makes the window autom. dedicated at the end. This can be used by third party packages like JDEE to create arbitrary ECB-windows besides the standard tree-windows.

   To make a special ECB-window a dedicated window either one of the five functions above must be used or a new "window-dedicator"-function has to be defined with 'defecb-window-dedicator' and must be used within the layout-definition.
3. Every(!) special ECB-window must be dedicated as described in 2.
4. *CREATE-CODE* must work correctly regardless if there is already a compile-window (stored in `ecb-compile-window`) or not (`ecb-compile-window` is nil).

Things *CREATE-CODE* can do or can use:

1. The value of `ecb-compile-window` which contains the compile-window (if there is one). Using the values of `ecb-compile-window-height`, `ecb-windows-width`, `ecb-windows-height`.

Things *CREATE-CODE* must NOT do:

1. Splitting the edit-window
2. Creating a compile-window
3. Deleting the edit-window, the compile-window (if there is any) or the ECB-windows-column(s)/row (see Precondition 1.)
4. Referring to the value of `ecb-edit-window` because this is always nil during *CREATE-CODE*.

Postconditions for *CREATE-CODE*:

1. The edit-window must be the selected window and must not be dedicated and not be splitted.
2. Every window besides the edit-window (and the compile-window) must be a dedicated window (e.g. a ECB-tree-window).

Use this macro to program new layouts within your '`.emacs`' or any other file which is loaded into your Emacs. After loading the file(s) with all the new layout-definitions you can use it by customizing the option `ecb-layout-name` to the appropriate name or with the command `ecb-change-layout`.

With the function `ecb-layout-undefine` you can remove a layout from the list of available layouts:

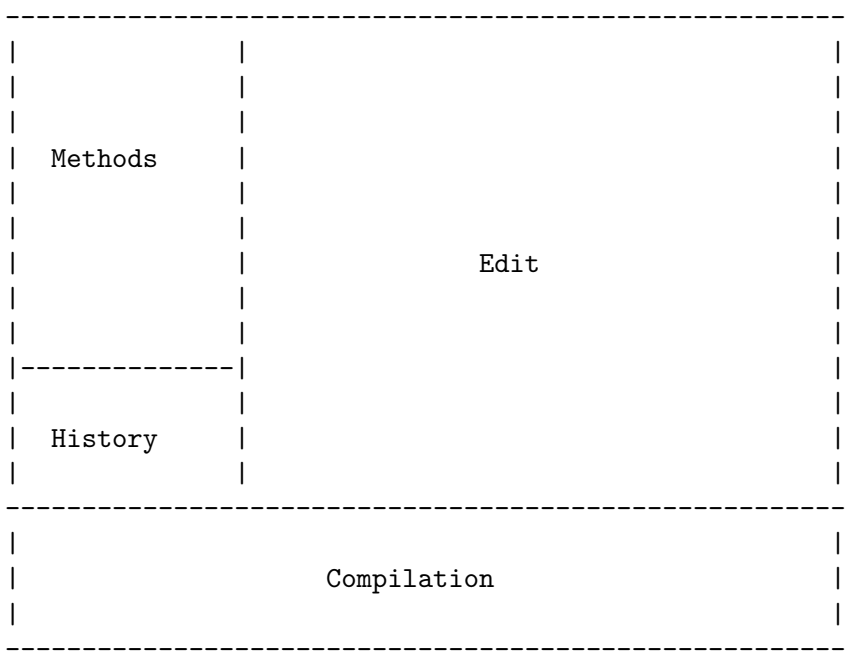
`layout-undefine` *name* [Function]

Unbind ecb-layout-function-<NAME> and ecb-delete-window-ecb-windows-<NAME> and remove `NAME` from `ecb-available-layouts`.

Here is an example for a new layout programmed with `ecb-layout-define`:

```
(ecb-layout-define "my-own-layout" left nil
  ;; The frame is already splitted side-by-side and point stays in the
  ;; left window (= the ECB-tree-window-column)

  ;; Here is the creation code for the new layout

  ;; 1. Defining the current window/buffer as ECB-methods buffer
  (ecb-set-methods-buffer)
  ;; 2. Splitting the ECB-tree-windows-column in two windows
  (ecb-split-ver 0.75 t)
  ;; 3. Go to the second window
  (other-window 1)
  ;; 4. Defining the current window/buffer as ECB-history buffer
  (ecb-set-history-buffer)
  ;; 5. Make the ECB-edit-window current (see Postcondition above)
  (select-window (next-window)))
```

This layout definition defines a layout with name “my-own-layout” which looks like:



11.5.2 All aspects of programming special windows

ECB offers a flexible programmable layout-engine for other packages to display their own contents and informations in special ECB-windows. An example could be a graphical debugger which offers a special window for displaying local variables and another special window for messages from the debugger-process (like JDEbug of JDEE[4]).

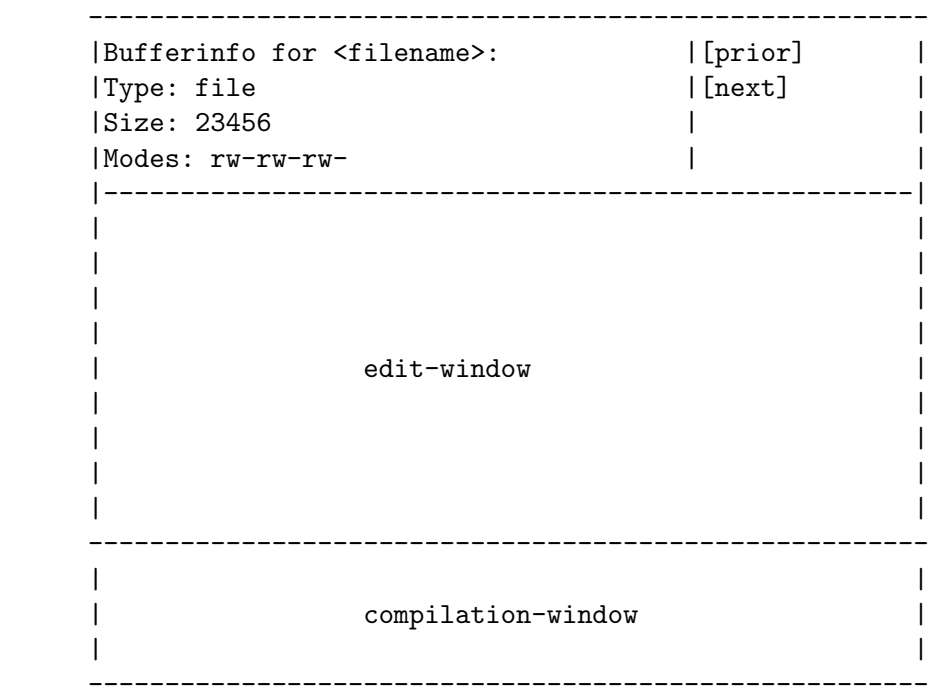
This section explains all aspects of programming new special windows, adding them to a new layout and synchronizing them with edit-window of ECB. This can be done best with an easy example which nevertheless covers all necessary aspects to be a good example and skeleton for complex tools (like a graphical debugger) which want to use the layout-engine of ECB do display their own information.

IMPORTANT: See ⟨undefined⟩ [tree-buffer], page ⟨undefined⟩ for a full documentation of the library tree-buffer.el which can be used for programming a new special window as a tree!

Here comes the example:

[4] JDEE is available at `http://jdee.sunsite.dk/`

11.5.2.1 The outline of the example layout:



11.5.2.2 The description of the layout-contents

The top-left window always displays informations about the current buffer in the selected edit-window. This window demonstrates how autom. synchronizing a special window/buffer of a layout with current edit-window.

The top-right window contains an read-only "action-buffer" and offers two buttons which can be used with the middle mouse-button to scroll the edit-window. This is not very senseful but it demonstrates how to control the edit-window with actions performed in a special window/buffer of a layout.

(If you have not set a compilation-window in `ecb-compile-window-height` then the layout contains no persistent compilation window and the other windows get a little more place).

11.5.2.3 The example code

Now let have us a look at the several parts of the Elisp-program needed to program this new example layout. ECB contains a library '`ecb-examples.el`' which contains the full working code of this example. To test this example and to play with it you can load this library into Emacs (with `load-library` for example) and then calling `ecb-change-layout` (bound to *`C-c . lc`*) and inserting "example-layout1" as layout-name. An alternative is calling `ecb-examples-activate` and `ecb-examples-deactivate`. For details see file '`ecb-examples.el`'.

The following steps only contain code-skeletons to demonstrate the principle. The full working code is available in '`ecb-examples.el`'.

11.5.2.4 The bufferinfo buffer of the example

The name of the bufferinfo buffer:

```
(defconst ecb-examples-bufferinfo-buffer-name " *ECB buffer info*")
```

Two helper functions for displaying infos in a special buffer:

```
(defun ecb-examples-print-file-attributes (buffer filename)
  (ecb-with-readonly-buffer buffer
    (erase-buffer)
    (insert (format "Bufferinfo for %s:\n\n" filename))
    ;; insert with the function 'file-attributes' some
    ;; informations about FILENAME.
    ))

(defun ecb-examples-print-non-filebuffer (buffer buffer-name)
  (ecb-with-readonly-buffer buffer
    (erase-buffer)
    ;; analogous to 'ecb-examples-print-file-attributes'
    ))
```

The main synchronizing function added to `ecb-current-buffer-sync-hook` for autom. evaluation by `ecb-current-buffer-sync` which runs dependent on the values of `ecb-window-sync` and `ecb-window-sync-delay`. This function synchronizes the bufferinfo buffer with the current buffer of the edit-window if that buffer has changed.

```
(defun ecb-examples-bufferinfo-sync ()
  (ecb-do-if-buffer-visible-in-ecb-frame
      'ecb-examples-bufferinfo-buffer-name

    ;; here we can be sure that the buffer with name
    ;; `ecb-examples-bufferinfo-buffer-name' is displayed in a
    ;; window of `ecb-frame'

    ;; The macro `ecb-do-if-buffer-visible-in-ecb-frame' locally
    ;; binds the variables visible-buffer and visible-window!! See
    ;; documentation of this macro!

    (let ((filename (buffer-file-name (current-buffer))))

      (if (and filename (file-readable-p filename))

          ;; real filebuffers
          ;; here we could add a smarter mechanism;
          ;; see ecb-examples.el
          (ecb-examples-print-file-attributes visible-buffer
                                              filename)

        ;; non file buffers like help-buffers etc...
        (setq ecb-examples-bufferinfo-last-file nil)
        (ecb-examples-print-non-filebuffer visible-buffer
                                           (buffer-name
                                             (current-buffer)))
        ))))
```

Two conveniance commands for the user:

```
(defun ecb-maximize-bufferinfo-window ()
  "Maximize the bufferinfo-window.
I.e. delete all other ECB-windows, so only one ECB-window and the
edit-window\(s) are visible \(and maybe a compile-window). Works
also if the ECB-analyse-window is not visible in current layout."
  (interactive)
  (ecb-display-one-ecb-buffer ecb-examples-bufferinfo-buffer-name))

(defun ecb-goto-bufferinfo-window ()
  "Make the bufferinfo-window the current window."
  (interactive)
  (ecb-goto-ecb-window ecb-examples-bufferinfo-buffer-name))
```

The function which makes the bufferinfo-buffer dedicated to a window and registers the new special window/buffer at ECB.

```
(defecb-window-dedicator ecb-examples-set-bufferinfo-buffer
    ecb-examples-bufferinfo-buffer-name
  "Set the buffer in the current window to the bufferinfo-buffer
and make this window dedicated for this buffer."
  (switch-to-buffer (get-buffer-create
                     ecb-examples-bufferinfo-buffer-name))
  (setq buffer-read-only t))
```

This is all what we need for the special bufferinfo buffer. We have demonstrated already three of the important functions/macros of the layout-engine API of ECB: `ecb-with-readonly-buffer`, `ecb-do-if-buffer-visible-in-ecb-frame` and `defecb-window-dedicator` (see ⟨undefined⟩ [The layout-engine API], page ⟨undefined⟩. Especially the second macro is strongly recommended for programming good synchronizing functions which do not waste CPU!

11.5.2.5 The action buffer of the example

The name of the action-buffer:

```
(defconst ecb-examples-action-buffer-name " *ECB action buffer*")
```

Two helper functions for creating a readonly action-buffer with a special local key-map for the middle-mouse-button and two buttons [prior] and [next]:

```
(defun ecb-examples-insert-text-in-action-buffer (text)
  (let ((p (point)))
    (insert text)
    (put-text-property p (+ p (length text)) 'mouse-face
                                             'highlight)))

(defun ecb-examples-action-buffer-create ()
  (save-excursion
    (if (get-buffer ecb-examples-action-buffer-name)
        (get-buffer ecb-examples-action-buffer-name)

      (set-buffer (get-buffer-create
                   ecb-examples-action-buffer-name))

      ;; we setup a local key-map and bind middle-mouse-button
      ;; see ecb-examples.el for the full code

      ;; insert the action buttons [prior] and [next] and
      ;; make it read-only

      (ecb-with-readonly-buffer (current-buffer)
        (erase-buffer)
        (ecb-examples-insert-text-in-action-buffer "[prior]")
        ;; analogous for the [next] button
        )

      (current-buffer))))
```

The function which performs the actions in the action-buffer if clicked with the middle-mouse button onto a button [next] or [prior].

```
(defun ecb-examples-action-buffer-clicked (e)
  (interactive "e")
  (mouse-set-point e)
  (let ((line (buffer-substring (ecb-line-beginning-pos)
                                (ecb-line-end-pos))))
    (cond ((string-match "prior" line)
           (ecb-select-edit-window)
           (call-interactively 'scroll-down))
          ((string-match "next" line)
           ;; analogous for [next]
           ))))
```

Two conveniance-commands for the user:

```
(defun ecb-maximize-action-window ()
  "Maximize the action-window.
I.e. delete all other ECB-windows, so only one ECB-window and the
edit-window\(s) are visible \(and maybe a compile-window). Works
also if the ECB-analyse-window is not visible in current layout."
  (interactive)
  (ecb-display-one-ecb-buffer ecb-examples-action-buffer-name))

(defun ecb-goto-action-window ()
  "Make the action-window the current window."
  (interactive)
  (ecb-goto-ecb-window ecb-examples-action-buffer-name))
```

The function which makes the action-buffer dedicated to a window and registers it at ECB.

```
(defecb-window-dedicator ecb-examples-set-action-buffer
    (buffer-name (ecb-examples-action-buffer-create))
  "Set the buffer in the current window to the action-buffer
and make this window dedicated for this buffer."
  (switch-to-buffer (buffer-name (ecb-examples-action-buffer-create))))
```

We do not need more code for the action buffer. All of the code is standard emacs-lisp which would also needed if used without ECB. You see that you can use any arbitrary code as second argument for `defecb-window-dedicator` as long it returns a buffer-name.

11.5.2.6 Adding the bufferinfo- and action-buffer to a new layout

Now we add the bufferinfo- and the action-buffer to a new layout of type top with name “example-layout1”:

```
(ecb-layout-define "example-layout1" top

  ;; dedicating the bufferinfo window to the bufferinfo-buffer
  (ecb-examples-set-bufferinfo-buffer)

  ;; creating the action-window
  (ecb-split-hor 0.75)

  ;; dedicate the action window to the action-buffer
  (ecb-examples-set-action-buffer)

  ;; select the edit-window
  (select-window (next-window)))
```

This all what we need to define the new layout. See ⟨undefined⟩ [Programming a new layout], page ⟨undefined⟩ for more details of the pure layout-programming task.

11.5.2.7 Synchronizing the bufferinfo-buffer automatically

The last thing we have to do is to synchronize the bufferinfo-buffer with current edit-window. We do this by adding `ecb-examples-bufferinfo-sync` to the hook `ecb-current-buffer-sync-hook'` (The file '`ecb-examples.el`' shows a smarter mechanism for (de)activating the new layout and the synchronization but this works also very well).

```
(add-hook 'ecb-current-buffer-sync-hook 'ecb-examples-bufferinfo-sync)
```

11.5.2.8 Activating and deactivating new layouts

Because a set of new special windows integrated in a new layout is often just the GUI of a complete tool (like a graphical debugger) we demonstrate here the complete activation and deactivation of such a tool or at least of the tool-GUI. We decide that the GUI of our example "tool" needs a compile-window with height 5 lines and the height of the special windows "row" on top should be exactly 6 lines (normally width and height of the special windows should be a fraction of the frame, but here we use 6 lines[5]

Here comes the (de)activation code.

The code for saving and restoring the state before activation (the full code is available in '`ecb-examples.el`':

```
(defun ecb-examples-preactivation-state(action)
  (cond ((equal action 'save)
         ;; code for saving the state
         )
        ((equal action 'restore)
        ;; code for restoring the state
        )))
```

The following function activates the GUI of our example tool:

[5] You can change the code in the file '`ecb-examples.el`' to use a frame-fraction of 0.2 instead of 6 hard lines, just try it!

```
(defun ecb-examples-activate ()
  (interactive)

  ;; activating the synchronization of the bufferinfo-window
  (add-hook 'ecb-current-buffer-sync-hook
            'ecb-examples-bufferinfo-sync)

  ;; saving the state
  (ecb-examples-preactivation-state 'save)

  ;; switch to our preferred layout
  (setq ecb-windows-height 6)
  (setq ecb-compile-window-height 5)
  (ecb-layout-switch "example-layout1"))
```

This function deactivates the GUI of our example-tool and restores the state as before activation:

```
(defun ecb-examples-deactivate ()
  (interactive)

  (remove-hook 'ecb-current-buffer-sync-hook
               'ecb-examples-bufferinfo-sync)
  (ecb-examples-preactivation-state 'restore)
  (ecb-layout-switch ecb-layout-name))
```

Now we have all code for the new layout and the new layout-buffers. The example is ready for use; just load '`ecb-examples.el`' (s.a.).

11.5.3 The wide range of possible layout-outlines

In the two previous sections ⟨undefined⟩ [Programming a new layout], page ⟨undefined⟩ and ⟨undefined⟩ [Programming special windows], page ⟨undefined⟩ we have explained in detail how to program new layouts and how to program new special windows/buffers and adding them to a new layout.

The intention of this section is to be a summary what are the real restrictions for a new layout-outline programmed with `ecb-layout-define`. This is necessary because until now we just programmed "obvious" layouts, means layout which are in principle very similar to the standard ones which means one big edit-window and some special windows "around" this edit-window. This section will show you that a layout can have also very different outlines.

OK, here are the real restrictions and conditions for a layout programmed with `ecb-layout-define`:

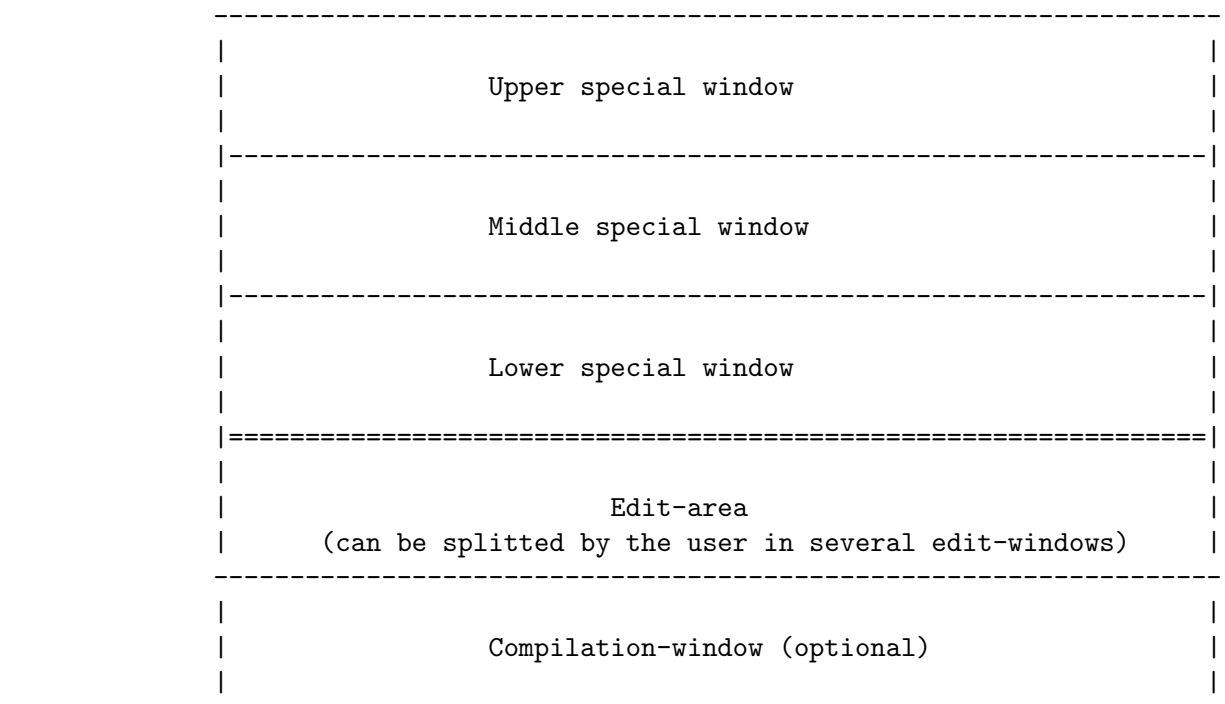
1. It must have exactly one edit-window regardless of its size. The user of this layout can later split this edit-window in as many edit-windows as he like.
2. All other windows created within the *CREATE-CODE* body of `ecb-layout-define` (see ⟨undefined⟩ [Programming a new layout], page ⟨undefined⟩) must be dedicated to their buffers.

3. All the dedicated windows must (exclusive!) either reside on the left, right, top or left-and-right side of the edit-window. This will be defined with the *TYPE*-argument of `ecb-layout-define` (see ⟨undefined⟩ [Programming a new layout], page ⟨undefined⟩).

You see, there are only three restrictions/conditions. These and only these must be fulfilled at layout-programming.

Demonstrating what this really means and how flexible the layout-engine of ECB really is, can be done best with some "pathological" layout-outlines. All the following are correct layouts (working code is added below each layout):

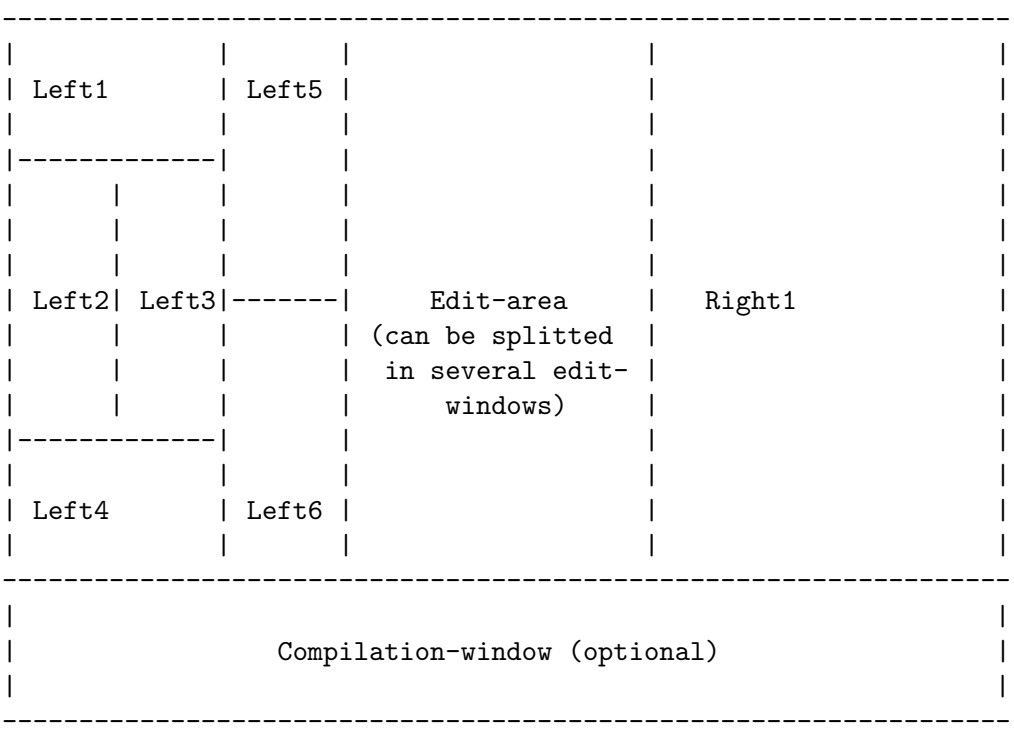
The following is a top layout with three vertical layered special windows.



Here is the code for that top layout (all buffers are dummy-buffers):

```
;; The "window dedicator" functions:

(defecb-window-dedicator ecb-set-usw-buffer "Upper special window"
  (switch-to-buffer (get-buffer-create "Upper special window")))

(defecb-window-dedicator ecb-set-msw-buffer "Middle special window"
  (switch-to-buffer (get-buffer-create "Middle special window")))

(defecb-window-dedicator ecb-set-lsw-buffer "Lower special window"
  (switch-to-buffer (get-buffer-create "Lower special window")))

;; The layout itself:

(ecb-layout-define "example-layout3" top
  nil

  ;; here we have an edit-window and above one top window which we can
  ;; now split in several other windows. Dependent on the value of
  ;; 'ecb-compile-window-height' we have also a compile-window at the
  ;; bottom.

  (ecb-set-usw-buffer)
  (ecb-split-ver 0.33)
  (ecb-set-msw-buffer)
  (ecb-split-ver 0.5)
  (ecb-set-lsw-buffer)

  ;; select the edit-window.
  (select-window (next-window)))
```

The following is a left-right layout which has six special windows in the left-"column" and one big special window in the right-"column". For left-right layouts the left-"column" and the right-"column" have always the same width.



Here is the code for that left-right layout, again with dummy-buffers (depending to your screen-resolution you will need a quite big value for `ecb-windows-width`, e.g. 0.4):

Here is one of the "window dedicator"-functions[6]:

```
(defecb-window-dedicator ecb-set-left1-buffer "Left1"
  (switch-to-buffer (get-buffer-create "Left1")))
```

Here is the layout-definition itself:

[6] The "window dedicators" for all these ecb-windows/buffers are not explicitly described - they look all like `ecb-set-left1-buffer` - of course with different buffer-names!

```
(ecb-layout-define "example-layout2" left-right
  nil

  ;; here we have an edit-window and left and right two windows each
  ;; with width 'ecb-windows-width'. Dependent to the value of
  ;; 'ecb-compile-window-height' we have also a compile-window at the
  ;; bottom.

  (ecb-set-left1-buffer)
  (ecb-split-hor 0.66 t)
  (ecb-split-ver 0.75)
  (ecb-set-left4-buffer)
  (select-window (previous-window (selected-window) 0))
  (ecb-split-ver 0.25 nil t)
  (ecb-set-left2-buffer)
  (ecb-split-hor 0.5)
  (ecb-set-left3-buffer)
  (select-window (next-window (next-window)))
  (ecb-set-left5-buffer)
  (ecb-split-ver 0.5)
  (ecb-set-left6-buffer)
  (select-window (next-window (next-window)))
  (ecb-set-right1-buffer))

  ;; select the edit-window
  (select-window (previous-window (selected-window) 0)))
```

Especially the last example should demonstrate that even very complicated layouts are easy to program with `ecb-layout-define`. If such layouts are senseful is another topic ;-)

11.5.4 The complete layout-engine API of ECB

This section lists all functions, macros, variables and user-options the layout-engine API of ECB offers foreign packages. Call `describe-function` rsp. `describe-variable` to get a detailed description.

Functions and macros for programming with layouts and special ecb-windows:

- `defecb-window-dedicator`
- `ecb-available-layouts-member-p`
- `ecb-canonical-ecb-windows-list`
- `ecb-canonical-edit-windows-list`
- `ecb-compile-window-live-p`
- `ecb-compile-window-state`
- `ecb-do-if-buffer-visible-in-ecb-frame`
- `ecb-exec-in-window`
- `ecb-get-current-visible-ecb-buffers`
- `ecb-layout-define`

- `ecb-layout-switch`
- `ecb-layout-undefine`
- `ecb-point-in-compile-window`
- `ecb-point-in-ecb-window`
- `ecb-point-in-edit-window`
- `ecb-select-edit-window`
- `ecb-split-hor`
- `ecb-split-ver`
- `ecb-where-is-point`
- `ecb-with-dedicated-window`[7]

Utility functions/macros:

- `ecb-display-one-ecb-buffer`
- `ecb-enlarge-window`
- `ecb-fix-filename`
- `ecb-goto-ecb-window`
- `ecb-window-live-p`
- `ecb-with-readonly-buffer`

Some other maybe useful functions/macros:

- `ecb-with-adviced-functions`
- `ecb-with-original-functions`
- `ecb-with-some-adviced-functions`

Some useful **READONLY** variables:

- `ecb-compile-window`
- `ecb-last-edit-window-with-point`
- `ecb-last-source-buffer`

Caution: DO NOT USE THE VARIABLE `ecb-edit-window` IN YOUR PROGRAMS!

User-options and hooks related to the layout-engine API:

- `ecb-current-buffer-sync-hook`
- `ecb-hide-ecb-windows-after-hook`
- `ecb-hide-ecb-windows-before-hook`
- `ecb-redraw-layout-after-hook`
- `ecb-redraw-layout-before-hook`
- `ecb-show-ecb-windows-after-hook`
- `ecb-show-ecb-windows-before-hook`
- `ecb-windows-height`
- `ecb-windows-width`
- `ecb-compile-window-height`

[7] Normally not needed because `defecb-window-dedicator` does all necessary.

12 Conflicts and bugs of ECB

This chapter describes what to do when there are conflicts with other packages and also the known (and currently unfixed) bugs of ECB. If possible (and in most cases it is possible ;-) then a practicable solution or workaround is described.

12.1 Conflicts with other packages

This chapter contains a list of already known conflict between ECB and other packages and how to solve them - in most cases ECB already contains a suitable workaround.

That is followed by a general recipe what you can do when you have detected a conflict between ECB and a package is not listed in the know-conflicts-section.

12.1.1 Proved workarounds or recommendations for other packages

Here is a list of packages which are proved to work properly with ECB and if not (i.e. there are conflicts) then helpful solutions/hints/workarounds are offered:

12.1.1.1 Package bs.el

The package bs.el offers a nifty buffer-selection buffer. The main command of this package is `bs-show`. With ECB < 2.20 this command does not really working well within activated ECB. But as of version 2.20 of ECB there should be no problems using this package.

If you add "*buffer-selection*" as buffer-name to the option `ecb-compilation-buffer-names` then ECB will always display the buffer-selection buffer of bs in the compile-window (if there is one). Otherwise bs will use the edit-area to do its job.

12.1.1.2 Package BBDB

As of ECB 2.21 there should be no conflicts between BBDB and ECB, so BBDB can be used even when the ECB-windows are visible.

But if you encounter problems then it is recommened to use one of the window-managers escreen.el or winring.el (see ⟨undefined⟩ [Window-managers and ECB], page ⟨undefined⟩). With such a window-manager ECB and BBDB should work together very well under all circumstances!

12.1.1.3 Package calendar.el

With activated ECB `calendar` does not shrink its window to the small size but splits the window equally. But if you add this to your '`.emacs`' it works:

```
(add-hook 'initial-calendar-window-hook
          (function (lambda ()
                      (when (and ecb-minor-mode
                                 (ecb-point-in-edit-window))
                        ;; if no horizontal split then nothing
                        ;; special to do
                        (or (= (frame-width) (window-width))
                            (shrink-window (- (window-height) 9))))
                    )))
```

12.1.1.4 Package cygwin-mount.el

There can be a conflict between ECB and cygwin-mount.el if the following conditions are true:

- You are working with cygwin-mount.el (sounds clear :-)
- You have set `cygwin-mount-build-mount-table-asynch` to not nil
- ECB is automatically started after starting Emacs (e.g. with `ecb-auto-activate` or calling `ecb-activate` in `window-setup-hook`)
- Your Emacs-setup contains a call of `cygwin-mount-activate`.

Under these circumstances Emacs 21.X sometimes eats up the whole CPU (at least with Windows XP) and the cygwin-mount-table is never build.

But there is an easy work-around: Call `cygwin-mount-activate` first *AFTER* ECB is activated. This can be done with the hook `ecb-activate-hook`:

```
(add-hook 'ecb-activate-hook
          (function (lambda ()
                      (require 'cygwin-mount)
                      (setq cygwin-mount-build-mount-table-asynch t)
                      (cygwin-mount-activate))))
```

12.1.1.5 Package desktop.el

ECB works perfectly with the desktop-saver desktop.el. But to ensure this the option `desktop-minor-mode-table` **MUST** contain the following entry:

```
(ecb-minor-mode nil)
```

Without this entry desktop.el tries for each buffer it loads after Emacs-start to enable `ecb-minor-mode` and therefore to start ECB. This conflicts with ECB! Therefore you must add the entry above to `desktop-minor-mode-table`!

Further it is strongly recommended to add entries for all the minor-mode of the semantic-package to `desktop-minor-mode-table`, so for example add also:

```
(semantic-show-unmatched-syntax-mode nil)
(semantic-stickyfunc-mode nil)
(senator-minor-mode nil)
(semantic-idle-scheduler-mode nil)
```

Which modes you have to add depends on which modes of semantic you use. But to get sure you should add all minor-modes of the semantic-package because these modes are normally activated by the related "global" command (e.g. `global-semantic-show-unmatched-syntax-mode`) or by adding the minor-mode to the related major-mode-hook.

It has also been reported that just disabling the Tip-Of-The-Day (option: `ecb-tip-of-the-day`) fixes the compatibility-problems with desktop.el. Just try it out!

12.1.1.6 Package edebug (Lisp Debugger)

It is strongly recommended to run edebug only when the ECB-windows are hidden. With visible ECB-windows there will probably serious conflicts between the ECB-layout and the edebug-window-manager.

12.1.1.7 Package ediff.el

In most cases ECB works very well with ediff (see option `ecb-run-ediff-in-ecb-frame`). But currently suspending ediff with `ediff-suspend` and restoring the ediff-session (e.g. with command `eregistry`) does confuse the window-management of ECB.

If you often use ediff in a scenario where you suspend ediff and reactivate it later then it is recommended to exit ECB first (`ecb-deactivate` or `ecb-minor-mode`)!

12.1.1.8 Package func-menu.el

This package has been reported to produce some conflicts under some circumstances when ECB is activated. Some of them could be reproduced by the ECB-maintainer. So the recommendation is to disable func-menu-support when using ECB. Normally using func-menu makes no sense in combination with ECB because ECB provides the same and even more informations as func-menu - so func-menu is redundant ;-)

12.1.1.9 Package Gnus (Newsreader)

As of ECB 2.21 there should be no conflicts between Gnus and ECB, so Gnus can be used even when the ECB-windows are visible.

But if you encounter problems then it is recommened to use one of the window-managers escreen.el or winring.el (see ⟨undefined⟩ [Window-managers and ECB], page ⟨undefined⟩). With such a window-manager ECB and Gnus should work together very well under all circumstances!

12.1.1.10 Package JDEE (Java Development Environment)

JDEE has a lot of "dialogs" where the user can select among several choices. An example is importing classes via the command `jde-import-find-and-import`. These dialogs are strongly designed to work in an environment where a new temporary window is created, the contents of the dialog are displayed in the new window, the user select his choice and hits [OK]. After that the new window is deleted and the selection is performed (for example the chosen import statement are inserted in the source-buffer.

Caution: ECB can work very well with this dialogs but only if the buffer-name of these dialog-buffers (normally "Dialog") is not contained in the option `ecb-compilation-buffer-names`. So do not add the string "Dialog" to this option!

Please Note: Regardless if a persistent compile-window is used (i.e. `ecb-compile-window-height` is not nil) or not, these JDEE-dialogs will always being displayed by splitting the edit-window of ECB and not within the compile-window.

12.1.1.11 Package scroll-all.el (scroll-all-mode)

ECB advices `scroll-all-mode` so it is working correct during running ECB. This means if point stays in an edit-window and the edit-window is splitted then all edit-windows are scrolled by `scroll-all-mode` and no other window! If point stays in any other window just this selected window is scrolled. This is the only senseful behavior of `scroll-all-mode` with ECB.

12.1.1.12 Package VC (Version Control)

The variable `vc-delete-logbuf-window` must be set to nil during active ECB. This can be done with the hooks mentioned in ⟨undefined⟩ [Elisp programming], page ⟨undefined⟩.

12.1.1.13 Package VM (Emacs Mail-Client)

As of ECB 2.21 there should be no conflicts between VM and ECB, so VM can be used even when the ECB-windows are visible.

But if you encounter problems then it is recommened to use one of the window-managers escreen.el or winring.el (see ⟨undefined⟩ [Window-managers and ECB], page ⟨undefined⟩). With such a window-manager ECB and VM should work together very well under all circumstances!

12.1.1.14 Package winner.el (winner-mode)

`winner-mode` is autom. disabled as long as ECB is running. ECB has its own window-management which is completely incompatible with `winner-mode`! But `winner-mode` makes also not really sense with ECB.

12.1.1.15 Package wb-line-number.el

Do not use the package wb-line-number.el in combination with ECB - it will not work and it will not work under any circumstances and there is no way to make it work together and there will be no way in the future!

The reason behind that is: wb-line-number.el uses additional dedicated windows to display the line-numbers. And ECB can not work if there there are additional dedicated windows - additional to that ones created by ECB.

12.1.1.16 Application xrefactory

Xrefactory (also known as Xref, X-ref and Xref-Speller), the refactoring browser for (X)Emacs[1], can be used during running ECB regardless if the ECB-windows are visible or not. There should be no conflicts as of ECB versions >= 2.21.

If there are conflicts with the Xref-browser then the most recommended way is to use one of the window-manager escreen.el or winring.el (and then use different escreens or window-configurations for ECB and Xrefactory-browsing - ⟨undefined⟩ [Window-managers and ECB], page ⟨undefined⟩).

12.1.2 What to do for unknown conflicts with other packages

As of version 2.20 the layout-engine of ECB is so flexible that normally there should be no conflicts with other packages unless these packages have their own complete window-layout-management (e.g. Gnus, BBDB, Xrefactory). But these packages can and should be handled very well with the window-manager-support of ECB (see ⟨undefined⟩ [Window-managers and ECB], page ⟨undefined⟩).

So if you detect an unknown (i.e. not listed in the conflicts-list in the next subsection) conflict with a small package and some of its commands and you have installed an ECB-version < 2.20 the first task you have to do is to upgrade to a version >= 2.20!

If this doesn't solve the problem a very probable reason for the conflict is that the command fails if called from another window than an edit-window of the ecb-frame. So please check if the problem disappears if you call the failing command from an edit-window of ECB. If this is true then you you can add the following code to your .emacs (and of course replace the XXX with the failing command):

[1] Xrefactory is available at `http://www.xref-tech.com`

```
(defadvice XXX (before ecb activate)
  "Ensures 'XXX' works well when called from another window
as an edit-window. Does nothing if called in another frame
as the 'ecb-frame'."
  (when (equal (selected-frame) ecb-frame)
    (unless (ecb-point-in-edit-window)
      (ecb-select-edit-window))))
```

This before-advice runs before the command XXX and ensures that the XXX is called from within an edit-window if the current selected window is not an edit-window. It does nothing if called for another frame as the ecb-frame.

If such an advice solves the problem then please send a note with the solution to the ECB-mailing-list or direct to the ECB-maintainer so the solution can be integrated in the next ECB-release

If calling from an edit-window fails too then please file a complete bug-report to the ECB-mailing-list (see ⟨undefined⟩ [Submitting problem report], page ⟨undefined⟩). This report should contain a detailed description which command of which package fails under which circumstances!

12.2 Known bugs

This section describes all currently known bugs of ECB. The maintainers of ECB try to fix these bugs as soon as possible.

12.2.1 Following the source-file link in a help-buffer

The following bug occurs only in ECB-versions < 1.96 and is fixed since ECB 1.96!!

This bug only occurs if a compile-window is used and visible!

If you call functions like `describe-function` which displays a help-buffer in the compile-window, then you will often get an output like this in the compile-window:

```
ecb-activate is an interactive compiled Lisp function in 'ecb'.
(ecb-activate)

Activates the ECB...
```

The link to 'ecb' is normally a click-able link, means if you click with the middle-mouse button onto it the file is opened (in our example '`ecb.el`' would be opened.

If you click onto it when the help-buffer is already the current buffer (i.e. the compile-window is already selected before the click!) then all is working fine (i.e. the file is opened in the edit-window), but if you click onto the link without selecting the compile-window before (i.e. the edit-window is the current selected window) then the file is opened in the compile-window which is probably not what you want. Not a big problem but annoying.

The only available workaround is, first selecting the compile-window and then clicking onto the link!

12.2.2 Extra history-entries for JDEE source-buffers

ECB on occasions creates an extra edit buffer for entries in the history window. For example, let say there are three entries in the history window:

```
Test1
Test2
Test3
```

In the edit window Test1 file is edited. When clicked on Test2 entry in history, on occasion instead of switching to the existing buffer for Test2, a new edit buffer is opened for Test2 file. At this point, there are four entries in the history as follows:

```
Test2
Test2<2>
Test1
Test3
```

13 Frequently asked questions

This is the Emacs Code Browser FAQ.

Question	Answer
What is the first step i should do if i have problems with ECB?	Read carefully the related sections of the online-help of ECB.
What should i do, if a have a problem which can not be solved even after reading the online-help?	Send a problem-report to the ECB-mailing-list with the command `ecb-submit-problem-report`. See ⟨undefined⟩ [Submitting problem report], page ⟨undefined⟩.
What should i do, if another package seems not to work correct with ECB?	Take a look into ⟨undefined⟩ [Conflicts], page ⟨undefined⟩. If your package is not listed there then submit a problem-report.
Can ECB parse and display source-contents not supported by semantic?	Yes, in all version >= 1.94. ECB can now parse and display all source-contents supported by semantic, imenu or etags - same as speedbar. See ⟨undefined⟩ [Non-semantic sources], page ⟨undefined⟩.
Why are the lines in the ECB-, temp- and compilation-buffers not wrapped but truncated?	Check the variable `truncate-partial-width-windows` and set it to nil.
Why doesn't ECB work correct with VC?	The variable `vc-delete-logbuf-window` must be set to nil during active ECB. This can be done with the hooks of ECB.
Does ECB support C++ as well as Java?	This depends strongly on the used semantic-version, but all semantic-versions >= semantic-1.4.3 support C++ really well.
Does ECB support Perl?	If perl can be parsed either by imenu, etags or semantic then ECB supports perl. Of course ECB would support perl best if perl is supported by semantic.

Does ECB support language XYZ?

See question "Does ECB support Perl?" and replace "Perl" with "XYZ" in the answer.

How to add new languages to ECB?

Add the language XYZ to semantic (perform all necessary steps described in the semantic-manual) and ECB will automatically support language XYZ! There is nothing to do in ECB itself! Same when you write an imenu- or etags-support for language XYZ.

Why does ECB not recognize my source-files for C++?

Your C++-files have probably an extension which is not mapped to c++-mode in `auto-mode-alist` and/or your own Emacs-setup has "destroyed" the correct value of the hook-variable `c++-mode-hook`. See ⟨undefined⟩ [Setting up Emacs], page ⟨undefined⟩.

Why doesn't ECB display the node name in the echo area if mouse moves over it?

There can be several reasons: First the value of the option `ecb-show-node-name-in-minibuffer` must be either `always` or `if-too-long`. If this is OK, then maybe you have turned on follow-mouse AFTER activating ECB; follow-mouse must be turned on BEFORE ECB is activated, e.g. in the `ecb-activate-hook`! But with Emacs 21.X and XEmacs there are no problems with this feature, just activate it.

Is it possible to make the history of ECB persistent?

You can use the library "desktop.el" which works very well with ECB. Then all files of your recent Emacs-session will be opened automatically after next Emacs-start and will be added automatically to the ECB-history after ECB-start.

Is there an "Intellisense"-mechanism like with other IDEs?

For Java the JDEE[1] has this feature and for all other languages semantic offer something similar, see `http://cedet.sourceforge.net/intellisense.shtml`

[1] `http://jdee.sunsite.dk/`

Can i use ECB in combination with Gnus within one frame?	You can, but for ECB-versions < 1.96 it is not recommended because each of them has it's own window-management and probably there will be conflicts, so use different frames for ECB and Gnus! But beginning with ECB 1.96 you can use either escreen.el or winring.el as "window-manager" which allows you in consequence to use ECB and applications like Gnus in one frame! See ⟨undefined⟩ [Window-managers and ECB], page ⟨undefined⟩.
Can i speed up displaying the contents of big-size directories?	Yes, see the option `ecb-cache-directory-contents`. Read the section ⟨undefined⟩ [Large directories], page ⟨undefined⟩.
Is it possible to create/use other layouts than the built-in ones?	Yes. ⟨undefined⟩ [Creating a new ECB-layout], page ⟨undefined⟩ and ⟨undefined⟩ [The layout-engine], page ⟨undefined⟩ are the relevant sections. The former one describes how to create interactively new layouts where the latter one is for Elisp-programmers.
Can i use speedbar as directory-browser within ECB?	Yes, see ⟨undefined⟩ [Integrating speedbar], page ⟨undefined⟩.
Can i exclude subdirectories from the recursive grep in the directories buffer?	Yes, see ⟨undefined⟩ [Grepping directories], page ⟨undefined⟩.
How can i prevent contaminating each directory with a file '`semantic-cache`'?	Set `semanticdb-default-save-directory` to a directory.
Why ECB displays large portions of current source-file with dark background?	This comes from semantic; see ⟨undefined⟩ [Setting up Emacs], page ⟨undefined⟩.
Why ECB underlines some parts of current source-file?	This comes from semantic; see ⟨undefined⟩ [Setting up Emacs], page ⟨undefined⟩.

Can i add my own commands to the popup-menus of tree-buffers?	Yes, see ⟨undefined⟩ [Using the mouse], page ⟨undefined⟩.
Can ECB display the compile-window “on demand”?	Yes, see ⟨undefined⟩ [Tips and tricks], page ⟨undefined⟩.
Which buffers are treated as compilation-buffers by ECB?	See the docstring of the function `ecb-compilation-buffer-p`.
How can i change the modeline of an ECB-tree-buffer?	You can change it with the options `ecb-mode-line-prefixes`, `ecb-mode-line-data` and `ecb-mode-line-display-window-number`.
Can the tree-buffers being selected faster than with the standard-keybindings of ECB?	Yes, see option `ecb-mode-line-display-window-number`.
Can ECB display the window-number in the modeline of the special windows?	Yes, see option `ecb-mode-line-display-window-number`.
How can i change the keybindings of ECB?	You can do this with option `ecb-key-map` (see ⟨undefined⟩ [ecb-general], page ⟨undefined⟩).
What can i do if hiding/showing from the methods-buffer does not work?	Either the current `major-modes` is not supported by hideshow or you have to add an entry to `hs-special-modes-alist` (see ⟨undefined⟩ [Hideshow], page ⟨undefined⟩).
Can i maximize one of the ECB-windows for better overlook?	Yes, see ⟨undefined⟩ [Maximizing the ECB windows], page ⟨undefined⟩.
Can i hide the ECB-windows for getting more editing-space?	Yes, see ⟨undefined⟩ [Hiding the ECB windows], page ⟨undefined⟩.
Can i define the actions ECB performs after visiting a tag?	Yes, see ⟨undefined⟩ [Visiting tags], page ⟨undefined⟩.

Buffers are not displayed correctly in the compile-window?	See ⟨undefined⟩ [Problems with the compile window], page ⟨undefined⟩.
Can ECB work together with window-managers like escreen.el?	Yes, see ⟨undefined⟩ [Window-managers and ECB], page ⟨undefined⟩.
Can i remove these "ugly" vertical lines from a tree-buffer?	Yes, see option `ecb-tree-buffer-style`.
ECB does not display images in the tree-buffers - what can i do?	Customize `ecb-tree-buffer-style` and restart ECB. But note: GNU Emacs <= 21.3.X for Windows does not support image-display so ECB uses always ascii-guide-lines even when here the image-style is set in `ecb-tree-buffer-style`.
Do `special-display-function` et. al. work with ECB.	Yes, see ⟨undefined⟩ [Using special-display with ECB], page ⟨undefined⟩.
Can i activate the popup-menu of a tree-buffer from keyboard?	Yes, see ⟨undefined⟩ [Using popup-menus], page ⟨undefined⟩.
Can i display the popup-menu of a tree-buffer with tmm?	Yes, see ⟨undefined⟩ [Using popup-menus], page ⟨undefined⟩.
Does ECB disable all advices after deactivation?	"Nes"[2], see remarks in the documentation of the option `ecb-split-edit-window-after-start`.
Can ECB preserve the full state of ECB between deactivation and next activation?	Yes, see the option `ecb-split-edit-window-after-start`.
Can i change the behavior how ECB chooses another window for selecting it or scrolling it.	Yes, see ⟨undefined⟩ [The other window], page ⟨undefined⟩.

[2] Nes is a combination of No and Yes :-)

Can i increase the allowed depth of nested submenus.	Yes, see the docstring of the option `ecb-directories-menu-user-extension`.
Can i apply some filters to the Tree-buffers.	Yes, see ⟨undefined⟩ [Filtering the tree-buffers], page ⟨undefined⟩
With XEmacs i get sometimes an error "Wrong number of arguments: widen (1)". What can i do?	Disable the func-menu support in your XEmacs-setup. See ⟨undefined⟩ [Conflicts], page ⟨undefined⟩.
Can i use desktop.el in combination with ECB?	Yes, see ⟨undefined⟩ [Conflicts], page ⟨undefined⟩.
Opening directories takes a long time - what can i do?	Read ⟨undefined⟩ [Large directories], page ⟨undefined⟩.
ECB seems to be blocked sometimes - what is the reason?	ECB performs some stealthy tasks when idle - this can cause sometimes a blocked Emacs but this tasks will be immetiatelly interrupted by any user-event so there should be normally no problems. But especially for mounted net-drives some of the stealthy tasks can take time up to some seconds for each file - and during one file-operation it can not be interrupted. See also `ecb-stealthy-tasks-delay`.
Can i exclude certain directories from being checked for emptyness?	Yes, see option `ecb-prescan-directories-exclude-regexps`.
Can i exclude certain directories from checking the VC-state of the contained sources?	Yes, see option `ecb-vc-directory-exclude-regexps`.
Can i exclude certain directories from checking the read-only-state of the contained sources?	Yes, see option `ecb-read-only-check-exclude-regexps`.

ECB ignores the remote-paths i have added to `ecb-source-path`.	Maybe you have to check the option `ecb-ping-options`. Ensure that this option contains a value suitable for your ping-program (see `ecb-ping-program`). See also ⟨undefined⟩ [Remote directories], page ⟨undefined⟩.
ECB seems to be blocked a long time.	Maybe you use cygwin-XEmacs. Then either the empty-dir-check (see option `ecb-prescan-directories-for-emptyness`) or the VC-support (see `ecb-vc-enable-support`) can block ECB. For the latter one see ⟨undefined⟩ [Known VC-problems], page ⟨undefined⟩.
ECB seems to be blocked during the VC-state update in the tree-windows.	Maybe the root repository for the current directory is a remote-repository. This can result in a long lasting check-time per file. See also ⟨undefined⟩ [Version-control support], page ⟨undefined⟩ for hints what you can do.
I have encountered some problems with the display of the VC-state in the tree-buffers.	See also ⟨undefined⟩ [Version-control support], page ⟨undefined⟩ for hints what you can do.
I get errors when trying to download new ECB with `ecb-download-ecb`.	Ensure that the ECB-configuration of these tools is correct for your system (see `ecb-wget-setup`, `ecb-gzip-setup` and `ecb-tar-setup`).

Command Index

This index contains all user commands of ECB.

Please note: The commands in this index are listed without the prefix “ecb-” (e.g. the command `ecb-activate` is listed with name “activate”).

(Index is nonexistent)

Option Index

This index contains all customizable options of ECB.

Please note: All options in this index are listed without the prefix “ecb-” (e.g. the option `ecb-layout-name` is listed with name “layout-name”).

(Index is nonexistent)

Concept Index

(Index is nonexistent)

Table of Contents