Version
****

The Coding Conventions of the Rabbit Project, last updated December 7, 2010.

This Document is abridged and modified from "GNU Coding Standards" by Mike Zou <zxhmike@gmail.com> to meet the requirements of the project.

You can obtain a copy of the original document at http://www.gnu.org/prep/standards/

0. Preamble

The members of the Rabbit project should read this document carefully. They should obey all the rules described in this document. This part was originally written for C Programmers. However, C++ programmers have to refer to these articles as well, though the commenting conventions may differ. See the section of commenting for more details.

1. Formatting Your Source Code

It is important to put the open-brace that starts the body of a C function in column one, so that they will start a defun. Several tools look for open-braces in column one to find the beginnings of C functions. These tools will not work on code not formatted that way.

Avoid putting open-brace, open-parenthesis or open-bracket in column one when they are inside a function, so that they won't start a defun. The open-brace that starts a struct body can go in column one if you find it useful to treat that definition as a defun.

It is also important for function definitions to start the name of the function in column one. This helps people to search for function definitions, and may also help certain tools recognize them. Thus, using Standard C syntax, the format is this:

```
static char *
concat (char *s1, char *s2)
{
  …
}
```

In Standard C, if the arguments don't fit nicely on one line, split it like this:

```
int
lots_of_args (int an_integer, long a_long, short a_short,
              double a_double, float a_float)
…
```

For the body of the function, our style looks like this:

```
if (x < foo (y, z))
  haha = bar[4] + 5;
else
  {
    while (z)
      {
        haha += foo (z, z);
        z--;
      }
    return ++x + bar ();
  }
```

Note where the spaces lie. We find it easier to read a program when it has spaces before the open-parentheses and after the commas. Especially after the commas.

When you split an expression into multiple lines, split it before an operator, not after one. Here is the right way:

```
if (foo_this_is_long && bar > win (x, y, z)
    && remaining_condition)
```

Try to avoid having two operators of different precedence at the same level of indentation. For example, don’t write this:

```
mode = (inmode[j] == VOIDmode
        || GET_MODE_SIZE (outmode[j]) > GET_MODE_SIZE (inmode[j])
        ? outmode[j] : inmode[j]);
```

Instead, use extra parentheses so that the indentation shows the nesting:

```
mode = ((inmode[j] == VOIDmode
         || (GET_MODE_SIZE (outmode[j]) > GET_MODE_SIZE (inmode[j])))
        ? outmode[j] : inmode[j]);
```

Format do-while statements like this:

```
do
  {
    a = foo (a);
  }
while (a > 0);
```

2 Commenting Your Work

Please write a brief comment at the start of each source file, with the file name and a line or two about the overall purpose of the file. This kind of comment of other files rather than source files, such as header files and configuration files is also welcomed.

Then after that of a source, attach a GNU GPL copyright statement. Files other than source files should not include this statement. All of above should look like this:

```
/*
<filename>: <purpose>
Copyright (C) <year> Rabbit Project of Fudan University

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/
```

Please write the comments in a GNU program in English, because English is the one language that nearly all programmers in all countries can read.

Please put a comment on each function saying what the function does, what sorts of arguments it gets, and what the possible values of arguments mean and are used for. It is not necessary to duplicate in words the meaning of the C argument declarations, if a C type is being used in its customary fashion. If there is anything nonstandard about its use (such as an argument of type char * which is really the address of the second character of a string, not the first), or any possible values that would not work the way one would expect (such as, that strings containing newlines are not guaranteed to work), be sure to say so.

Also explain the significance of the return value, if there is one.

The comment of the function should look like this:

```
/*----------------------------------------------------------------------------------------
 * (function: foo)
 *      This is the comment line one.
 *      This is the comment line two.
 * Return:
```

```
 *      1, if bar is true.
 *      -1, if bar is false.
 *----------------------------------------------------------------------------------------*/
```

Note that there is always a tab at the beginning of the function descriptions.

The comment on a function is much clearer if you use the argument names to speak about the argument values. The variable name itself should be lower case, but write it in upper case when you are speaking about the value rather than the variable itself. Thus, “the inode number NODE_NUM” rather than “an inode”.

There should be a comment on each static variable as well, like this:

```
/* Nonzero means truncate lines in the display;
   zero means continue them.  */
int truncate_lines;
```

Be aware that C Programmers should always use the '/* */' style of programming while C++ Programmers should use the '//' style.

3. Clean Use of C Constructs

Declarations of external functions and functions to appear later in the source file should all go in one place near the beginning of the file (somewhere before the first function definition in the file), or else should go in a header file. Don’t put extern declarations inside functions.

It used to be common practice to use the same local variables (with names like tem) over and over for different values within one function. Instead of doing this, it is better to declare a separate local variable for each distinct purpose, and give it a name which is meaningful. This not only makes programs easier to understand, it also facilitates optimization by good compilers. You can also move the declaration of each local variable into the smallest scope that includes all its uses. This makes the program even cleaner.

Don’t use local variables or parameters that shadow global identifiers. GCC’s ‘-Wshadow’ option can detect this problem.

Don’t declare multiple variables in one declaration that spans lines. Start a new declaration on each line, instead. For example, instead of this:

```
int    foo,
       bar;
```

write either this:

```
int foo, bar;
```

or this:

```
int foo;
int bar;
```

(If they are global variables, each should have a comment preceding it anyway.)

When you have an if-else statement nested in another if statement, always put braces around the if-else. Thus, never write like this:

```
if (foo)
  if (bar)
    win ();
  else
    lose ();
```

always like this:

```
if (foo)
  {
    if (bar)
      win ();
    else
      lose ();
  }
```

If you have an if statement nested inside of an else statement, either write else if on one line, like this,

```
if (foo)
  …
else if (bar)
  …
```

with its then-part indented like the preceding then-part, or write the nested if within braces like this:

```
if (foo)
  …
else
  {
    if (bar)
      …
  }
```

Don’t declare both a structure tag and variables or typedefs in the same declaration. Instead, declare the structure tag separately and then use it to declare the variables or typedefs.

Try to avoid assignments inside if-conditions (assignments inside while-conditions are ok). For example, don’t write this:

```
if ((foo = (char *) malloc (sizeof *foo)) == 0)
  fatal ("virtual memory exhausted");
```

instead, write this:

```
foo = (char *) malloc (sizeof *foo);
if (foo == 0)
  fatal ("virtual memory exhausted");
```

This example uses zero without a cast as a null pointer constant. This is perfectly fine, except that a cast is needed when calling a varargs function or when using sizeof.

4 Naming Variables, Functions, and Files

The names of global variables and functions in a program serve as comments of a sort. So don’t choose terse names—instead, look for names that give useful information about the meaning of the variable or function. In a GNU program, names should be English, like other comments.

Local variable names can be shorter, because they are used only within one context, where (presumably) comments explain their purpose.

Try to limit your use of abbreviations in symbol names. It is ok to make a few abbreviations, explain what they mean, and then use them frequently, but don’t use lots of obscure abbreviations.

Please use underscores to separate words in a name, so that the Emacs word commands can be useful within them. Stick to lower case; reserve upper case for macros and enum constants, and for name-prefixes that follow a uniform convention.

For example, you should use names like ignore_space_change_flag; don’t use names like iCantReadThis.

Variables that indicate whether command-line options have been specified should be named after the meaning of the option, not after the option-letter. A comment should state both the exact meaning of the option and its letter. For example,

```
/* Ignore changes in horizontal whitespace (-b).  */
int ignore_space_change_flag;
```

When you want to define names with constant integer values, use enum rather than ‘#define’. GDB knows about enumeration constants.

You might want to make sure that none of the file names would conflict if the files were loaded onto an MS-DOS file system which shortens the names. You can use the program doschk to test for this.

Some GNU programs were designed to limit themselves to file names of 14 characters or less, to avoid file name conflicts if they are read into older System V systems. Please preserve this feature in the existing GNU programs that have it, but there is no need to do this in new GNU programs. doschk also reports file names longer than 14 characters.

5. Internationalization

This part may not be useful for Qt programmers, for Qt has its own internationalization mechanism. However, they can refer some of the conventions here.

GNU has a library called GNU gettext that makes it easy to translate the messages in a program into various languages. You should use this library in every program. Use English for the messages as they appear in the program, and let gettext provide the way to translate them into other languages.

Using GNU gettext involves putting a call to the gettext macro around each string that might need translation—like this:

```
printf (gettext ("Processing file `%s'..."));
```

This permits GNU gettext to replace the string "Processing file `%s'..." with a translated version.

Once a program uses gettext, please make a point of writing calls to gettext when you add new strings that call for translation.

Using GNU gettext in a package involves specifying a text domain name for the package. The text domain name is used to separate the translations for this package from the translations for other packages. Normally, the text domain name should be the same as the name of the package—for example, ‘coreutils’ for the GNU core utilities.

To enable gettext to work well, avoid writing code that makes assumptions about the structure of words or sentences. When you want the precise text of a sentence to vary depending on the data, use two or more alternative string constants each containing a complete sentences, rather than inserting conditionalized words or phrases into a single sentence framework.

Here is an example of what not to do:

```
printf ("%s is full", capacity > 5000000 ? "disk" : "floppy disk");
```

If you apply gettext to all strings, like this,

```
printf (gettext ("%s is full"),
        capacity > 5000000 ? gettext ("disk") : gettext ("floppy disk"));
```

the translator will hardly know that "disk" and "floppy disk" are meant to be substituted in the other string. Worse, in some languages (like French) the construction will not work: the translation of the word "full" depends on the gender of the first part of the sentence; it happens to be not the same for "disk" as for "floppy disk".

Complete sentences can be translated without problems:

```
printf (capacity > 5000000 ? gettext ("disk is full")
        : gettext ("floppy disk is full"));
```

A similar problem appears at the level of sentence structure with this code:

```
printf ("#  Implicit rule search has%s been done.\n",
        f->tried_implicit ? "" : " not");
```

Adding gettext calls to this code cannot give correct results for all languages, because negation in some languages requires adding words at more than one place in the sentence. By contrast, adding gettext calls does the job straightforwardly if the code starts out like this:

```
printf (f->tried_implicit
        ? "#  Implicit rule search has been done.\n",
        : "#  Implicit rule search has not been done.\n");
```