Coding standards

From "AllStarLink Wiki"

Contents

Open Source

Allstar is an open-source project licensed under the GPL based on Asterisk and written in C. This means you get the source code and can make any modifications you wish provided you make your modifications available to the community.

Formatting source code

Keep the length of source lines to 79 characters or less, for maximum readability in the widest range of environments.

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

For more details : https://www.gnu.org/prep/standards/html_node/Formatting.html#Formatting

Comments

Every program should start with a comment saying briefly what it is for. Example: 'fmt - filter for simple filling of text'. This comment should be at the top of the source file containing the 'main' function of the program.

Also, 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.

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. If you do not write English well, please write comments in English as well as you can, then ask other people to help rewrite them. If you can't write comments in English, please find someone to work with you and translate your comments into English.

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.

Please put two spaces after the end of a sentence in your comments, so that the Emacs sentence commands will work. Also, please write complete sentences and capitalize the first word. If a lower-case identifier comes at the beginning of a sentence, don't capitalize it! Changing the spelling makes it a different identifier. If you don't like starting a sentence with a lower case letter, write the sentence differently (e.g., "The identifier lower-case is …").

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 is usually no purpose in restating the name of the function in the comment before it, because readers can see that for themselves. There might be an exception when the comment is so long that the function itself would be off the bottom of the screen.

See: https://www.gnu.org/prep/standards/html_node/Comments.html#Comments

Syntactic Conventions

- Clean use of C constructs.

Don't make the program ugly just to placate static analysis tools such as lint, clang, and GCC with extra warnings options such as -Wconversion and -Wundef. These tools can help find bugs and unclear code, but they can also generate so many false alarms that it hurts readability to silence them with unnecessary casts, wrappers, and other complications. For example, please don't insert casts to void or calls to do-nothing functions merely to pacify a lint checker.

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.

See: https://www.gnu.org/prep/standards/html_node/Syntactic-Conventions.html#Syntactic-Conventions

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.

https://www.gnu.org/prep/standards/html_node/Names.html#Names

Portability between System Types

In the Unix world, "portability" refers to porting to different Unix versions. For a GNU program, this kind of portability is desirable, but not paramount.

The primary purpose of GNU software is to run on top of the GNU kernel, compiled with the GNU C compiler, on various types of CPU. So the kinds of portability that are absolutely necessary are quite limited. But it is important to support Linux-based GNU systems, since they are the form of GNU that is popular.

Beyond that, it is good to support the other free operating systems (*BSD), and it is nice to support other Unix-like systems if you want to. Supporting a variety of Unix-like systems is desirable, although not paramount. It is usually not too hard, so you may as well do it. But you don't have to consider it an obligation, if it does turn out to be hard.

The easiest way to achieve portability to most Unix-like systems is to use Autoconf. It's unlikely that your program needs to know more information about the host platform than Autoconf can provide, simply because most of the programs that need such knowledge have already been written.

Avoid using the format of semi-internal data bases (e.g., directories) when there is a higher-level alternative (readdir).

As for systems that are not like Unix, such as MSDOS, Windows, VMS, MVS, and older Macintosh systems, supporting them is often a lot of work. When that is the case, it is better to spend your time adding features that will be useful on GNU and GNU/Linux, rather than on supporting other incompatible systems.

If you do support Windows, please do not abbreviate it as "win". In hacker terminology, calling something a "win" is a form of praise. You're free to praise Microsoft Windows on your own if you want, but please don't do this in GNU packages. Instead of abbreviating "Windows" to "win", you can write it in full or abbreviate it

to “woe” or “w”. In GNU Emacs, for instance, we use ‘w32’ in file names of Windows-specific files, but the macro for Windows conditionals is called WINDOWSNT.

It is a good idea to define the “feature test macro” _GNU_SOURCE when compiling your C files. When you compile on GNU or GNU/Linux, this will enable the declarations of GNU library extension functions, and that will usually give you a compiler error message if you define the same function names in some other way in your program. (You don’t have to actually use these functions, if you prefer to make the program more portable to other systems.)

But whether or not you use these GNU extensions, you should avoid using their names for any other meanings. Doing so would make it hard to move your code into other GNU programs.

https://www.gnu.org/prep/standards/html_node/System-Portability.html#System-Portability

- CPU Portability: Supporting the range of CPU types.
- System Functions: Portability and “standard” library functions.

Category: How to
