



Cross-Referencing Linux

Linux/Documentation/CodingStyle

[[source navigation](#)]
 [[diff markup](#)]
 [[identifier search](#)]
 [[freetext search](#)]
 [[file search](#)]

Version: [[1.0.9](#)] [[1.2.13](#)] [[2.0.40](#)] [[2.2.26](#)] [[2.4.18](#)] [[2.4.20](#)] [[2.4.28](#)] [[2.6.10](#)] [[2.6.11](#)]
 [[2.6.17.13](#)] [[2.6.18](#)]
 Architecture: [[i386](#)] [[alpha](#)] [[arm](#)] [[ia64](#)] [[m68k](#)] [[mips](#)] [[mips64](#)] [[ppc](#)] [[s390](#)] [[sh](#)] [[sparc](#)]
 [[sparc64](#)] [[x86_64](#)]

[1](#)

[2](#)

Linux kernel coding style

[3](#)

[4](#)

[5](#)

[6](#)

[7](#)

[8](#)

[9](#)

[10](#)

[11](#)

[12](#)

[13](#)

Anyway, here goes:

[14](#)

[15](#)

[16](#)

Chapter 1: Indentation

[17](#)

[18](#)

[19](#)

[20](#)

[21](#)

[22](#)

[23](#)

[24](#)

[25](#)

[26](#)

[27](#)

[28](#)

[29](#)

[30](#)

[31](#)

[32](#)

[33](#)

[34](#)

[35](#)

[36](#)

[37](#)

[38](#)

[39](#)

[40](#)

[41](#)

```
if (condition) do_this;
    do_something_everytime;
```

[42](#)

[43](#)

[44](#)

Outside of comments, documentation and except in Kconfig, spaces are never

[45](#) used for indentation, and the above example is deliberately broken.

[46](#)

[47](#) Get a decent editor and don't leave whitespace at the end of lines.

[48](#)

[49](#)

[50](#) Chapter 2: Breaking long lines and strings

[51](#)

[52](#) Coding style is all about readability and maintainability using commonly
[53](#) available tools.

[54](#)

[55](#) The limit on the length of lines is 80 columns and this is a hard limit.

[56](#)

[57](#) Statements longer than 80 columns will be broken into sensible chunks.

[58](#) Descendants are always substantially shorter than the parent and are placed
[59](#) substantially to the right. The same applies to function headers with a long
[60](#) argument list. Long strings are as well broken into shorter strings.

[61](#)

```
62 void fun(int a, int b, int c)
```

```
63 {
```

```
64     if (condition)
```

```
65         printk(KERN_WARNING "Warning this is a long printk with "  
66                               "3 parameters a: %u b: %u "  
67                               "c: %u \n", a, b, c);
```

```
68     else
```

```
69         next_statement;
```

```
70 }
```

[71](#)

[72](#) Chapter 3: Placing Braces

[73](#)

[74](#) The other issue that always comes up in C styling is the placement of
[75](#) braces. Unlike the indent size, there are few technical reasons to
[76](#) choose one placement strategy over the other, but the preferred way, as
[77](#) shown to us by the prophets Kernighan and Ritchie, is to put the opening
[78](#) brace last on the line, and put the closing brace first, thusly:

[79](#)

```
80     if (x is true) {
```

```
81         we do y
```

```
82     }
```

[83](#)

[84](#) However, there is one special case, namely functions: they have the
[85](#) opening brace at the beginning of the next line, thus:

[86](#)

```
87     int function(int x)
```

```
88     {
```

```
89         body of function
```

```
90     }
```

[91](#)

[92](#) Heretic people all over the world have claimed that this inconsistency
[93](#) is ... well ... inconsistent, but all right-thinking people know that
[94](#) (a) K&R are right and (b) K&R are right. Besides, functions are
[95](#) special anyway (you can't nest them in C).

[96](#)

[97](#) Note that the closing brace is empty on a line of its own, except in
[98](#) the cases where it is followed by a continuation of the same statement,
[99](#) ie a "while" in a do-statement or an "else" in an if-statement, like

```
100 this:
101
102     do {
103         body of do-loop
104     } while (condition);
105
```

106 and

```
107
108     if (x == y) {
109         ..
110     } else if (x > y) {
111         ...
112     } else {
113         ....
114     }
115
```

116 Rationale: K&R.

117

118 Also, note that this brace-placement also minimizes the number of empty
119 (or almost empty) lines, without any loss of readability. Thus, as the
120 supply of new-lines on your screen is not a renewable resource (think
121 25-line terminal screens here), you have more empty lines to put
122 comments on.

123

124

125 Chapter 4: Naming

126

127 C is a Spartan language, and so should your naming be. Unlike Modula-2
128 and Pascal programmers, C programmers do not use cute names like
129 `ThisVariableIsATemporaryCounter`. A C programmer would call that
130 variable "tmp", which is much easier to write, and not the least more
131 difficult to understand.

132

133 HOWEVER, while mixed-case names are frowned upon, descriptive names for
134 global variables are a must. To call a global function "foo" is a
135 shooting offense.

136

137 GLOBAL variables (to be used only if you really need them) need to
138 have descriptive names, as do global functions. If you have a function
139 that counts the number of active users, you should call that
140 `count_active_users()` or similar, you should not call it `cntusr()`.

141

142 Encoding the type of a function into the name (so-called Hungarian
143 notation) is brain damaged - the compiler knows the types anyway and can
144 check those, and it only confuses the programmer. No wonder MicroSoft
145 makes buggy programs.

146

147 LOCAL variable names should be short, and to the point. If you have
148 some random integer loop counter, it should probably be called "i".
149 Calling it `loop_counter` is non-productive, if there is no chance of it
150 being mis-understood. Similarly, "tmp" can be just about any type of
151 variable that is used to hold a temporary value.

152

153 If you are afraid to mix up your local variable names, you have another
154 problem, which is called the function-growth-hormone-imbalance syndrome.

[155](#) See next chapter.

[156](#)

[157](#)

[158](#)

Chapter 5: Functions

[159](#)

[160](#) Functions should be short and sweet, and do just one thing. They should
[161](#) fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24,
[162](#) as we all know), and do one thing and do that well.

[163](#)

[164](#) The maximum length of a function is inversely proportional to the
[165](#) complexity and indentation level of that function. So, if you have a
[166](#) conceptually simple function that is just one long (but simple)
[167](#) case-statement, where you have to do lots of small things for a lot of
[168](#) different cases, it's OK to have a longer function.

[169](#)

[170](#) However, if you have a complex function, and you suspect that a
[171](#) less-than-gifted first-year high-school student might not even
[172](#) understand what the function is all about, you should adhere to the
[173](#) maximum limits all the more closely. Use helper functions with
[174](#) descriptive names (you can ask the compiler to in-line them if you think
[175](#) it's performance-critical, and it will probably do a better job of it
[176](#) than you would have done).

[177](#)

[178](#) Another measure of the function is the number of local variables. They
[179](#) shouldn't exceed 5-10, or you're doing something wrong. Re-think the
[180](#) function, and split it into smaller pieces. A human brain can
[181](#) generally easily keep track of about 7 different things, anything more
[182](#) and it gets confused. You know you're brilliant, but maybe you'd like
[183](#) to understand what you did 2 weeks from now.

[184](#)

[185](#)

Chapter 6: Centralized exiting of functions

[186](#)

[188](#) Albeit deprecated by some people, the equivalent of the goto statement is
[189](#) used frequently by compilers in form of the unconditional jump instruction.

[190](#)

[191](#) The goto statement comes in handy when a function exits from multiple
[192](#) locations and some common work such as cleanup has to be done.

[193](#)

[194](#) The rationale is:

[195](#)

[196](#) - unconditional statements are easier to understand and follow

[197](#) - nesting is reduced

[198](#) - errors by not updating individual exit points when making
[199](#) modifications are prevented

[200](#) - saves the compiler work to optimize redundant code away ;)

[201](#)

```
202 int fun(int a)
```

```
203 {
```

```
204     int result = 0;
```

```
205     char *buffer = kmalloc(SIZE);
```

```
206
```

```
207     if (buffer == NULL)
```

```
208         return -ENOMEM;
```

```
209 }
```

```
210     if (condition1) {
211         while (loop1) {
212             ...
213         }
214         result = 1;
215         goto out;
216     }
217     ...
218 out:
219     kfree(buffer);
220     return result;
221 }
```

222

223 Chapter 7: Commenting

224

225 Comments are good, but there is also a danger of over-commenting. NEVER
226 try to explain HOW your code works in a comment: it's much better to
227 write the code so that the `_working_` is obvious, and it's a waste of
228 time to explain badly written code.

229

230 Generally, you want your comments to tell WHAT your code does, not HOW.
231 Also, try to avoid putting comments inside a function body: if the
232 function is so complex that you need to separately comment parts of it,
233 you should probably go back to chapter 5 for a while. You can make
234 small comments to note or warn about something particularly clever (or
235 ugly), but try to avoid excess. Instead, put the comments at the head
236 of the function, telling people what it does, and possibly WHY it does
237 it.

238

239 When commenting the kernel API functions, please use the kernel-doc format.
240 See the files Documentation/kernel-doc-nano-HOWTO.txt and scripts/kernel-doc
241 for details.

242

243 Chapter 8: You've made a mess of it

244

245 That's OK, we all do. You've probably been told by your long-time Unix
246 user helper that "GNU emacs" automatically formats the C sources for
247 you, and you've noticed that yes, it does do that, but the defaults it
248 uses are less than desirable (in fact, they are worse than random
249 typing - an infinite number of monkeys typing into GNU emacs would never
250 make a good program).

251

252 So, you can either get rid of GNU emacs, or change it to use saner
253 values. To do the latter, you can stick the following in your .emacs file:

254

```
255 (defun linux-c-mode ()
256   "C mode with adjusted defaults for use with the Linux kernel."
257   (interactive)
258   (c-mode)
259   (c-set-style "K&R")
260   (setq tab-width 8)
261   (setq indent-tabs-mode t)
262   (setq c-basic-offset 8))
```

263

264 This will define the M-x linux-c-mode command. When hacking on a

[265](#) module, if you put the string `-*- linux-c -*-` somewhere on the first
[266](#) two lines, this mode will be automatically invoked. Also, you may want
[267](#) to add
[268](#)
[269](#) `(setq auto-mode-alist (cons '("/usr/src/linux.*/*\.[ch]" . linux-c-mode)`
[270](#) `auto-mode-alist))`
[271](#)
[272](#) to your `.emacs` file if you want to have `linux-c-mode` switched on
[273](#) automagically when you edit source files under `/usr/src/linux`.
[274](#)
[275](#) But even if you fail in getting emacs to do sane formatting, not
[276](#) everything is lost: use `"indent"`.
[277](#)
[278](#) Now, again, GNU indent has the same brain-dead settings that GNU emacs
[279](#) has, which is why you need to give it a few command line options.
[280](#) However, that's not too bad, because even the makers of GNU indent
[281](#) recognize the authority of K&R (the GNU people aren't evil, they are
[282](#) just severely misguided in this matter), so you just give indent the
[283](#) options `"-kr -i8"` (stands for "K&R, 8 character indents"), or use
[284](#) `"scripts/Lindent"`, which indents in the latest style.
[285](#)
[286](#) `"indent"` has a lot of options, and especially when it comes to comment
[287](#) re-formatting you may want to take a look at the man page. But
[288](#) remember: `"indent"` is not a fix for bad programming.
[289](#)

[290](#) Chapter 9: Configuration-files

[291](#)

[292](#) For configuration options (arch/xxx/Kconfig, and all the Kconfig files),
[293](#) somewhat different indentation is used.
[294](#)
[295](#) Help text is indented with 2 spaces.
[296](#)
[297](#)

```
298 if CONFIG_EXPERIMENTAL  
299     tristate CONFIG_BOOM  
300     default n  
301     help  
302         Apply nitroglycerine inside the keyboard (DANGEROUS)  
303     bool CONFIG_CHEER  
304     depends on CONFIG_BOOM  
305     default y  
306     help  
307         Output nice messages when you explode  
308 endif  
309
```

[310](#) Generally, `CONFIG_EXPERIMENTAL` should surround all options not considered
[311](#) stable. All options that are known to trash data (experimental write-
[312](#) support for file-systems, for instance) should be denoted (DANGEROUS), other
[313](#) experimental options should be denoted (EXPERIMENTAL).
[314](#)

[315](#) Chapter 10: Data structures

[316](#)

[317](#)

[318](#) Data structures that have visibility outside the single-threaded
[319](#) environment they are created and destroyed in should always have

320 reference counts. In the kernel, garbage collection doesn't exist (and
 321 outside the kernel garbage collection is slow and inefficient), which
 322 means that you absolutely have to reference count all your uses.

323
 324 Reference counting means that you can avoid locking, and allows multiple
 325 users to have access to the data structure in parallel - and not having
 326 to worry about the structure suddenly going away from under them just
 327 because they slept or did something else for a while.

328
 329 Note that locking is not a replacement for reference counting.
 330 Locking is used to keep data structures coherent, while reference
 331 counting is a memory management technique. Usually both are needed, and
 332 they are not to be confused with each other.

333
 334 Many data structures can indeed have two levels of reference counting,
 335 when there are users of different "classes". The subclass count counts
 336 the number of subclass users, and decrements the global count just once
 337 when the subclass count goes to zero.

338
 339 Examples of this kind of "multi-level-reference-counting" can be found in
 340 memory management ("struct mm_struct": mm_users and mm_count), and in
 341 filesystem code ("struct super_block": s_count and s_active).

342
 343 Remember: if another thread can find your data structure, and you don't
 344 have a reference count on it, you almost certainly have a bug.

345

346

347 Chapter 11: Macros, Enums and RTL

348

349 Names of macros defining constants and labels in enums are capitalized.

350

```
351 #define CONSTANT 0x12345
```

352

353 Enums are preferred when defining several related constants.

354

355 CAPITALIZED macro names are appreciated but macros resembling functions
 356 may be named in lower case.

357

358 Generally, inline functions are preferable to macros resembling functions.

359

360 Macros with multiple statements should be enclosed in a do - while block:

361

```
362 #define macrofun(a, b, c)          \
363     do {                          \
364         if (a == 5)               \
365             do_this(b, c);       \
366     } while (0)
```

367

368 Things to avoid when using macros:

369

370 1) macros that affect control flow:

371

```
372 #define F00(x)                    \
373     do {                          \
374         if (blah(x) < 0)         \
```

```
375         return -EBUGGERED;     \  
376     } while(0)  
377  
378 is a _very_ bad idea. It looks like a function call but exits the "calling"  
379 function; don't break the internal parsers of those who will read the code.  
380  
381 2) macros that depend on having a local variable with a magic name:  
382  
383 #define F00(val) bar(index, val)  
384  
385 might look like a good thing, but it's confusing as hell when one reads the  
386 code and it's prone to breakage from seemingly innocent changes.  
387  
388 3) macros with arguments that are used as l-values: F00(x) = y; will  
389 bite you if somebody e.g. turns F00 into an inline function.  
390  
391 4) forgetting about precedence: macros defining constants using expressions  
392 must enclose the expression in parentheses. Beware of similar issues with  
393 macros using parameters.  
394  
395 #define CONSTANT 0x4000  
396 #define CONSTEXP (CONSTANT | 3)  
397  
398 The cpp manual deals with macros exhaustively. The gcc internals manual also  
399 covers RTL which is used frequently with assembly language in the kernel.
```

400

401

Chapter 12: Printing kernel messages

402

403
404 Kernel developers like to be seen as literate. Do mind the spelling
405 of kernel messages to make a good impression. Do not use crippled
406 words like "dont" and use "do not" or "don't" instead.

407

408 Kernel messages do not have to be terminated with a period.

409

410 Printing numbers in parentheses (`%d`) adds no value and should be avoided.

411

412

Chapter 13: Allocating memory

413

414
415 The kernel provides the following general purpose memory allocators:
416 `kmalloc()`, `kzalloc()`, `kcalloc()`, and `vmalloc()`. Please refer to the API
417 documentation for further information about them.

418

419 The preferred form for passing a size of a struct is the following:

420

```
421     p = kmalloc(sizeof(*p), ...);
```

422

423 The alternative form where struct name is spelled out hurts readability and
424 introduces an opportunity for a bug when the pointer variable type is changed
425 but the corresponding `sizeof` that is passed to a memory allocator is not.

426

427 Casting the return value which is a void pointer is redundant. The conversion
428 from void pointer to any other pointer type is guaranteed by the C programming
429 language.

[430](#)

[431](#)

[432](#)

Chapter 14: The inline disease

[433](#)

[434](#) There appears to be a common misperception that gcc has a magic "make me
[435](#) faster" speedup option called "inline". While the use of inlines can be
[436](#) appropriate (for example as a means of replacing macros, see Chapter 11), it
[437](#) very often is not. Abundant use of the inline keyword leads to a much bigger
[438](#) kernel, which in turn slows the system as a whole down, due to a bigger
[439](#) icache footprint for the CPU and simply because there is less memory
[440](#) available for the pagecache. Just think about it; a pagecache miss causes a
[441](#) disk seek, which easily takes 5 miliseconds. There are a LOT of cpu cycles
[442](#) that can go into these 5 miliseconds.

[443](#)

[444](#) A reasonable rule of thumb is to not put inline at functions that have more
[445](#) than 3 lines of code in them. An exception to this rule are the cases where
[446](#) a parameter is known to be a compiletime constant, and as a result of this
[447](#) constantness you *know* the compiler will be able to optimize most of your
[448](#) function away at compile time. For a good example of this later case, see
[449](#) the kmalloc() inline function.

[450](#)

[451](#) Often people argue that adding inline to functions that are static and used
[452](#) only once is always a win since there is no space tradeoff. While this is
[453](#) technically correct, gcc is capable of inlining these automatically without
[454](#) help, and the maintenance issue of removing the inline when a second user
[455](#) appears outweighs the potential value of the hint that tells gcc to do
[456](#) something it would have done anyway.

[457](#)

[458](#)

[459](#)

Chapter 15: References

[460](#)

[461](#)

[462](#) The C Programming Language, Second Edition
[463](#) by Brian W. Kernighan and Dennis M. Ritchie.
[464](#) Prentice Hall, Inc., 1988.
[465](#) ISBN 0-13-110362-8 (paperback), 0-13-110370-9 (hardback).
[466](#) URL: <http://cm.bell-labs.com/cm/cs/cbook/>

[467](#)

[468](#) The Practice of Programming
[469](#) by Brian W. Kernighan and Rob Pike.
[470](#) Addison-Wesley, Inc., 1999.
[471](#) ISBN 0-201-61586-X.
[472](#) URL: <http://cm.bell-labs.com/cm/cs/tpop/>

[473](#)

[474](#) GNU manuals - where in compliance with K&R and this text - for cpp, gcc,
[475](#) gcc internals and indent, all available from <http://www.gnu.org/manual/>

[476](#)

[477](#) WG14 is the international standardization working group for the programming
[478](#) language C, URL: <http://www.open-std.org/JTC1/SC22/WG14/>

[479](#)

[480](#) Kernel CodingStyle, by greg@kroah.com at OLS 2002:
[481](#) http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_talk/html/

[482](#)

[483](#) --

[484](#) Last updated on 30 December 2005 by a community effort on LKML.

[\[*source navigation* \]](#)[\[diff markup \]](#)[\[identifier search \]](#)[\[freetext search \]](#)[\[file search \]](#)

This page was automatically generated by the [LXR engine](#).
[Arne Georg Gleditsch and Per Kristian Gjermshus <lxr@linux.no>](#)

