Linux Kernel Coding Style

Linus Torvalds

This is a short document describing the preferred coding style for the linux kernel. Coding style is very personal, and I won't *force* my views on anybody, but this is what goes for anything that I have to be able to maintain, and I'd prefer it for most other things too. Please at least consider the points made here.

First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture.

Anyway, here goes:

Chapter 1: Indentation

Tabs are 8 characters, and thus indentations are also 8 characters. There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of PI to be 3.

Rationale: The whole idea behind indentation is to clearly define where a block of control starts and ends. Especially when you've been looking at your screen for 20 straight hours, you'll find it a lot easier to see how the indentation works if you have large indentations.

Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.

In short, 8-char indents make things easier to read, and have the added benefit of warning you when you're nesting your functions too deep. Heed that warning.

Chapter 2: Placing Braces

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

```
if (x is true) {
        we do y
}
```

However, there is one special case, namely functions: they have the opening brace at the beginning of the next line, thus:

```
int function(int x)
{
        body of function
}
```

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

Note that the closing brace is empty on a line of its own, *except* in the cases where it is followed by a continuation of the same statement, i.e. a "while" in a do-statement or an "else" in an if-statement, like this:

```
do {
        body of do-loop
} while (condition);
```

and

```
if (x == y) {
        ..
} else if (x > y) {
        ...
} else {
```

```
            ....
    }
```

Rationale: K&R.

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

Chapter 3: Naming

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

However, while mixed-case names are frowned upon, descriptive names for global variables are a must. To call a global function `foo` is a shooting offense.

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

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

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

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

Chapter 4: Functions

Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80×24, as we all know), and do one thing and do that well.

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

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

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

Chapter 5: Commenting

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

Generally, you want your comments to tell WHAT your code does, not HOW. Also, try to avoid putting comments inside a function body: if the function is so complex that you need to separately comment parts of it, you should probably go back to chapter 4 for a while. You can make small comments to note or warn about something particularly clever (or ugly), but try to avoid excess. Instead, put the comments at the head of the

function, telling people what it does, and possibly WHY it does it.

Chapter 6: You've made a mess of it

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

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

```
(defun linux-c-mode ()
  "C mode with adjusted defaults for use with the Linux
kernel."
  (interactive)
  (c-mode)
  (setq c-indent-level 8)
  (setq c-brace-imaginary-offset 0)
  (setq c-brace-offset -8)
  (setq c-argdecl-indent 8)
  (setq c-label-offset -8)
  (setq c-continued-statement-offset 8)
  (setq indent-tabs-mode nil)
  (setq tab-width 8))
```

This will define the `M-x linux-c-mode` command. When hacking on a module, if you put the string `-*- linux-c -*-` somewhere on the first two lines, this mode will be automatically invoked. Also, you may want to add

```
(setq auto-mode-alist (cons '("/usr/src/linux.*/.*\\.[ch]
$" . linux-c-mode)
                            auto-mode-alist))
```

to your .emacs file if you want to have linux-c-mode switched on automagically when you edit source files under /usr/src/linux.

But even if you fail in getting emacs to do sane formatting, not everything is lost: use "indent".

Now, again, GNU indent has the same brain dead settings that GNU emacs has, which is why you need to give it a few command line options. However, that's not too bad, because even the makers of GNU indent recognize the authority of K&R (the GNU people aren't evil, they are just severely misguided in this matter), so you just give indent the options "-kr -i8" (stands for "K&R, 8 character indents").

"indent" has a lot of options, and especially when it comes to comment re-formatting you may want to take a look at the manual page. But remember: "indent" is not a fix for bad programming.