Linux kernel coding style

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.

The preferred way to ease multiple indentation levels in a switch statement is to align the "switch" and its subordinate "case" labels in the same column instead of "double-indenting" the "case" labels. E.g.:

```
	switch (suffix) {
	case 'G':
	case 'g':
		mem <<= 30;
		break;
	case 'M':
	case 'm':
		mem <<= 20;
		break;
	case 'K':
	case 'k':
		mem <<= 10;
		/* fall through */
	default:
		break;
	}
```

Don't put multiple statements on a single line unless you have something to hide:

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

```
          do_something_everytime;
```

Don't put multiple assignments on a single line either. Kernel coding style is super simple. Avoid tricky expressions.

Outside of comments, documentation and except in Kconfig, spaces are never used for indentation, and the above example is deliberately broken.

Get a decent editor and don't leave whitespace at the end of lines.

Chapter 2: Breaking long lines and strings

Coding style is all about readability and maintainability using commonly available tools.

The limit on the length of lines is 80 columns and this is a strongly preferred limit.

Statements longer than 80 columns will be broken into sensible chunks. Descendants are always substantially shorter than the parent and are placed substantially to the right. The same applies to function headers with a long argument list. Long strings are as well broken into shorter strings. The only exception to this is where exceeding 80 columns significantly increases readability and does not hide information.

```
void fun(int a, int b, int c)
{
        if (condition)
                printk(KERN_WARNING "Warning this is a long printk with "
                                                "3 parameters a: %u b: %u "
                                                "c: %u \n", a, b, c);
        else
                next_statement;
}
```

Chapter 3: Placing Braces and Spaces

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
        }
```

This applies to all non-function statement blocks (if, switch, for, while, do). E.g.:

```
        switch (action) {
        case KOBJ_ADD:
                return "add";
        case KOBJ_REMOVE:
                return "remove";
        case KOBJ_CHANGE:
                return "change";
        default:
                return NULL;
        }
```

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, ie 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.

Do not unnecessarily use braces where a single statement will do.

```
if (condition)
	action();
```

This does not apply if one branch of a conditional statement is a single statement. Use braces in both branches.

```
if (condition) {
	do_this();
	do_that();
} else {
	otherwise();
}
```

3.1: Spaces

Linux kernel style for use of spaces depends (mostly) on function–versus–keyword usage. Use a space after (most) keywords. The notable exceptions are sizeof, typeof, alignof, and __attribute__, which look somewhat like functions (and are usually used with parentheses in Linux, although they are not required in the language, as in: "sizeof info" after "struct fileinfo info;" is declared).

So use a space after these keywords:

```
		if, switch, case, for, do, while
```

but not with sizeof, typeof, alignof, or __attribute__. E.g.,

```
		s = sizeof(struct file);
```

Do not add spaces around (inside) parenthesized expressions. This example is *bad*:

```
		s = sizeof( struct file );
```

When declaring pointer data or a function that returns a pointer type, the preferred use of '*' is adjacent to the data name or function name and not adjacent to the type name. Examples:

```
		char *linux_banner;
		unsigned long long memparse(char *ptr, char **retptr);
		char *match_strdup(substring_t *s);
```

Use one space around (on each side of) most binary and ternary operators, such as any of these:

```
		=  +  -  <  >  *  /  %  |  &  ^  <=  >=  ==  !=  ?  :
```

but no space after unary operators:

```
		&  *  +  -  ~  !  sizeof  typeof  alignof  __attribute__  defined
```

no space before the postfix increment & decrement unary operators:

```
		++  --
```

no space after the prefix increment & decrement unary operators:

```
		++  --
```

and no space around the '.' and "->" structure member operators.

Do not leave trailing whitespace at the ends of lines. Some editors with "smart" indentation will insert whitespace at the beginning of new lines as appropriate, so you can start typing the next line of code right away. However, some such editors do not remove the whitespace if you end up not putting a line of code there, such as if you leave a blank line. As a result, you end up with lines containing trailing whitespace.

Git will warn you about patches that introduce trailing whitespace, and can optionally strip the trailing whitespace for you; however, if applying a series of patches, this may make later patches in the series fail by changing their context lines.

Chapter 4: 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 chapter 6 (Functions).

Chapter 5: Typedefs

Please don't use things like "vps_t".

It's a _mistake_ to use typedef for structures and pointers. When you see a

```
vps_t a;
```

in the source, what does it mean?

In contrast, if it says

```
struct virtual_container *a;
```

you can actually tell what "a" is.

Lots of people think that typedefs "help readability". Not so. They are useful only for:

(a) totally opaque objects (where the typedef is actively used to _hide_ what the object is).

Example: "pte_t" etc. opaque objects that you can only access using the proper accessor functions.

NOTE! Opaqueness and "accessor functions" are not good in themselves. The reason we have them for things like pte_t etc. is that there really is absolutely _zero_ portably accessible information there.

(b) Clear integer types, where the abstraction _helps_ avoid confusion whether it is "int" or "long".

u8/u16/u32 are perfectly fine typedefs, although they fit into category (d) better than here.

NOTE! Again – there needs to be a _reason_ for this. If something is "unsigned long", then there's no reason to do

```
typedef unsigned long myflags_t;
```

but if there is a clear reason for why it under certain circumstances might be an "unsigned int" and under other configurations might be "unsigned long", then by all means go ahead and use a typedef.

(c) when you use sparse to literally create a _new_ type for type–checking.

(d) New types which are identical to standard C99 types, in certain exceptional circumstances.

Although it would only take a short amount of time for the eyes and brain to become accustomed to the standard types like 'uint32_t', some people object to their use anyway.

Therefore, the Linux–specific 'u8/u16/u32/u64' types and their signed equivalents which are identical to standard types are permitted –– although they are not mandatory in new code of your own.

When editing existing code which already uses one or the other set of types, you should conform to the existing choices in that code.

(e) Types safe for use in userspace.

In certain structures which are visible to userspace, we cannot require C99 types and cannot use the 'u32' form above. Thus, we use __u32 and similar types in all structures which are shared with userspace.

Maybe there are other cases too, but the rule should basically be to NEVER EVER use a typedef unless you can clearly match one of those rules.

In general, a pointer, or a struct that has elements that can reasonably be directly accessed should _never_ be a typedef.

Chapter 6: 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 80x24, 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 than 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.

In source files, separate functions with one blank line. If the function is exported, the EXPORT∗ macro for it should follow immediately after the closing function brace line. E.g.:

```
int system_is_up(void)
{
        return system_state == SYSTEM_RUNNING;
}
EXPORT_SYMBOL(system_is_up);
```

In function prototypes, include parameter names with their data types. Although this is not required by the C language, it is preferred in Linux because it is a simple way to add valuable information for the reader.

Chapter 7: Centralized exiting of functions

Albeit deprecated by some people, the equivalent of the goto statement is used frequently by compilers in form of the unconditional jump instruction.

The goto statement comes in handy when a function exits from multiple locations and some common work such as cleanup has to be done.

The rationale is:

- unconditional statements are easier to understand and follow
- nesting is reduced
- errors by not updating individual exit points when making modifications are prevented
- saves the compiler work to optimize redundant code away ;)

```
int fun(int a)
{
        int result = 0;
        char *buffer = kmalloc(SIZE);

        if (buffer == NULL)
                return -ENOMEM;

        if (condition1) {
                while (loop1) {
                        ...
                }
                result = 1;
                goto out;
        }
        ...
out:
        kfree(buffer);
        return result;
}
```

Chapter 8: 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 6 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.

When commenting the kernel API functions, please use the kernel-doc format.

See the files Documentation/kernel-doc-nano-HOWTO.txt and scripts/kernel-doc for details.

Linux style for comments is the C89 "/* ... */" style. Don't use C99-style "// ..." comments.

The preferred style for long (multi-line) comments is:

```
            /*
             * This is the preferred style for multi-line
             * comments in the Linux kernel source code.
             * Please use it consistently.
             *
             * Description: A column of asterisks on the left side,
             * with beginning and ending almost-blank lines.
             */
```

It's also important to comment data, whether they are basic types or derived types. To this end, use just one data declaration per line (no commas for multiple data declarations). This leaves you room for a small comment on each item, explaining its use.

Chapter 9: 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 - an 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 c-lineup-arglist-tabs-only (ignored)
 "Line up argument lists by tabs, not spaces"
 (let* ((anchor (c-langelem-pos c-syntactic-element))
             (column (c-langelem-2nd-pos c-syntactic-element))
             (offset (- (1+ column) anchor))
             (steps (floor offset c-basic-offset)))
      (* (max steps 1)
          c-basic-offset)))

(add-hook 'c-mode-common-hook
               (lambda ()
                  ;; Add kernel style
                  (c-add-style
                   "linux-tabs-only"
                   '("linux" (c-offsets-alist
                                     (arglist-cont-nonempty
                                      c-lineup-gcc-asm-reg
                                      c-lineup-arglist-tabs-only))))))

(add-hook 'c-mode-hook
               (lambda ()
                  (let ((filename (buffer-file-name)))
                     ;; Enable kernel mode for the appropriate files
                     (when (and filename
                                      (string-match (expand-file-name "~/src/linux-trees")
                                                           filename))
                        (setq indent-tabs-mode            t)
                        (c-set-style "linux-tabs-only")))))
```

This will make emacs go better with the kernel coding style for C files below ~/src/linux-trees.

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"), or use "scripts/Lindent", which indents in the latest style.

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

Chapter 10: Kconfig configuration files

For all of the Kconfig* configuration files throughout the source tree, the indentation is somewhat different. Lines under a "config" definition are indented with one tab, while help text is indented an additional two spaces. Example:

```
config AUDIT
	bool "Auditing support"
	depends on NET
	help
	  Enable auditing infrastructure that can be used with another
	  kernel subsystem, such as SELinux (which requires this for
	  logging of avc messages output).  Does not do system-call
	  auditing without CONFIG_AUDITSYSCALL.
```

Features that might still be considered unstable should be defined as dependent on "EXPERIMENTAL":

```
config SLUB
	depends on EXPERIMENTAL && !ARCH_USES_SLAB_PAGE_STRUCT
	bool "SLUB (Unqueued Allocator)"
	...
```

while seriously dangerous features (such as write support for certain filesystems) should advertise this prominently in their prompt string:

```
config ADFS_FS_RW
	bool "ADFS write support (DANGEROUS)"
	depends on ADFS_FS
	...
```

For full documentation on the configuration files, see the file Documentation/kbuild/kconfig-language.txt.

Chapter 11: Data structures

Data structures that have visibility outside the single-threaded environment they are created and destroyed in should always have reference counts. In the kernel, garbage collection doesn't exist (and outside the kernel garbage collection is slow and inefficient), which means that you absolutely _have_ to reference count all your uses.

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

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

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

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

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

Chapter 12: Macros, Enums and RTL

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

```
#define CONSTANT 0x12345
```

Enums are preferred when defining several related constants.

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

Generally, inline functions are preferable to macros resembling functions.

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

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

Things to avoid when using macros:

1) macros that affect control flow:

```
#define FOO(x)					\
	do {					\
		if (blah(x) < 0)		\
			return -EBUGGERED;	\
	} while(0)
```

is a _very_ bad idea. It looks like a function call but exits the "calling" function; don't break the internal parsers of those who will read the code.

2) macros that depend on having a local variable with a magic name:

```
#define FOO(val) bar(index, val)
```

might look like a good thing, but it's confusing as hell when one reads the code and it’s prone to breakage from seemingly innocent changes.

3) macros with arguments that are used as l–values: FOO(x) = y; will bite you if somebody e.g. turns FOO into an inline function.

4) forgetting about precedence: macros defining constants using expressions must enclose the expression in parentheses. Beware of similar issues with macros using parameters.

```
#define CONSTANT 0×4000
#define CONSTEXP (CONSTANT | 3)
```

The cpp manual deals with macros exhaustively. The gcc internals manual also covers RTL which is used frequently with assembly language in the kernel.

Chapter 13: Printing kernel messages

Kernel developers like to be seen as literate. Do mind the spelling of kernel messages to make a good impression. Do not use crippled words like "dont"; use "do not" or "don’t" instead. Make the messages concise, clear, and unambiguous.

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

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

There are a number of driver model diagnostic macros in <linux/device.h> which you should use to make sure messages are matched to the right device and driver, and are tagged with the right level: dev_err(), dev_warn(), dev_info(), and so forth. For messages that aren’t associated with a particular device, <linux/kernel.h> defines pr_debug() and pr_info().

Coming up with good debugging messages can be quite a challenge; and once you have them, they can be a huge help for remote troubleshooting. Such messages should be compiled out when the DEBUG symbol is not defined (that is, by default they are not included). When you use dev_dbg() or pr_debug(), that’s automatic. Many subsystems have Kconfig options to turn on –DDEBUG. A related convention uses VERBOSE_DEBUG to add dev_vdbg() messages to the ones already enabled by DEBUG.

Chapter 14: Allocating memory

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

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

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

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

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

Chapter 15: The inline disease

There appears to be a common misperception that gcc has a magic "make me faster" speedup option called "inline". While the use of inlines can be appropriate (for example as a means of replacing macros, see Chapter 12), it very often is not. Abundant use of the inline keyword leads to a much bigger kernel, which in turn slows the system as a whole down, due to a bigger icache footprint for the CPU and simply because there is less memory available for the pagecache. Just think about it; a pagecache miss causes a disk seek, which easily takes 5 milliseconds. There are a LOT of cpu cycles that can go into these 5 milliseconds.

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

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

Chapter 16: Function return values and names

Functions can return values of many different kinds, and one of the most common is a value indicating whether the function succeeded or failed. Such a value can be represented as an error−code integer (−Exxx = failure, 0 = success) or a "succeeded" boolean (0 = failure, non−zero = success).

Mixing up these two sorts of representations is a fertile source of difficult−to−find bugs. If the C language included a strong distinction between integers and booleans then the compiler would find these mistakes for us... but it doesn't. To help prevent such bugs, always follow this convention:

> If the name of a function is an action or an imperative command, the function should return an error−code integer. If the name is a predicate, the function should return a "succeeded" boolean.

For example, "add work" is a command, and the add_work() function returns 0 for success or −EBUSY for failure. In the same way, "PCI device present" is a predicate, and the pci_dev_present() function returns 1 if it succeeds in finding a matching device or 0 if it doesn't.

All EXPORTed functions must respect this convention, and so should all public functions. Private (static) functions need not, but it is recommended that they do.

Functions whose return value is the actual result of a computation, rather than an indication of whether the computation succeeded, are not subject to this rule. Generally they indicate failure by returning some out−of−range result. Typical examples would be functions that return pointers; they use NULL or the ERR_PTR mechanism to report failure.

Chapter 17: Don't re−invent the kernel macros

The header file include/linux/kernel.h contains a number of macros that you should use, rather than explicitly coding some variant of them yourself.

For example, if you need to calculate the length of an array, take advantage of the macro

```
 #define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
```

Similarly, if you need to calculate the size of some structure member, use

```
 #define FIELD_SIZEOF(t, f) (sizeof(((t*)0)–>f))
```

There are also min() and max() macros that do strict type checking if you need them. Feel free to peruse that header file to see what else is already defined that you shouldn't reproduce in your code.

Chapter 18: Editor modelines and other cruft

Some editors can interpret configuration information embedded in source files, indicated with special markers. For example, emacs interprets lines marked like this:

```
–*– mode: c –*–
```

Or like this:

```
/*
Local Variables:
compile–command: "gcc –DMAGIC_DEBUG_FLAG foo.c"
End:
*/
```

Vim interprets markers that look like this:

```
/* vim:set sw=8 noet */
```

Do not include any of these in source files. People have their own personal editor configurations, and your source files should not override them. This includes markers for indentation and mode configuration. People may use their own custom mode, or may have some other magic method for making indentation work correctly.

Appendix I: References

The C Programming Language, Second Edition
by Brian W. Kernighan and Dennis M. Ritchie.
Prentice Hall, Inc., 1988.
ISBN 0–13–110362–8 (paperback), 0–13–110370–9 (hardback).
URL: http://cm.bell–labs.com/cm/cs/cbook/

The Practice of Programming
by Brian W. Kernighan and Rob Pike.
Addison–Wesley, Inc., 1999.
ISBN 0–201–61586–X.
URL: http://cm.bell–labs.com/cm/cs/tpop/

GNU manuals – where in compliance with K&R and this text – for cpp, gcc, gcc internals and indent, all available from http://www.gnu.org/manual/

WG14 is the international standardization working group for the programming language C, URL: http://www.open–std.org/JTC1/SC22/WG14/

Kernel CodingStyle, by greg@kroah.com at OLS 2002:

http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_talk/html/

––
Last updated on 2007–July–13.