

NKE Coding Style

Portability

Portability refers to how easily —if at all— code can move from one system architecture to another. As little code as possible is machine-specific. Assembly is kept to a minimum, and interfaces and features are sufficiently general and abstract so that they work on a wide range of architectures. The obvious benefit is the relative ease with which a new architecture can be supported. The downside is that architecturespecific features are not supported, and code cannot be hand-tuned for a specific machine. With this design choice, optimal code is traded for portable code.

Everytime is possible, code should be written in C.

Word

A word is the amount of data that a machine can process at one time. A word is an integer number of bytes—for example, one, two, four, or eight. When someone talks about the “n-bits” of a machine, they are generally talking about the machine’s word size.

Typically, the virtual memory address space is equal to the word size, although the physical address space is sometimes less. Consequently, the size of a pointer is equal to the word size.

C Data Types

- **char** is always 1 byte or 8 bits. The signedness or unsignedness of a char variable needs to be explicitly declared.
- **int** is always 4 bytes or 32 bits
- **short** is always 2 bytes 16 bits
- never assume the size of **pointers** or **long**

Some abbreviations are used to indicate the size of some data types on different architectures:

- **ILPX**: int, long and pointers of size X
- **LPX**: long and pointers of size X
- **LLPX**: long long pointers of size X

Explicitly Sized Data Types

☐ s8, u8, s16, u16, s32, u32, s64, u64

They are generally used to deal with hardware properties. Thus, they need to be differently declared on distinct architectures. E.g.: In a 64 bits LP64 machine, the u64 type would be declared as `"typedef unsigned long u64"`

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 split your function into simpler ones. 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.

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.
 */
```

This is how a function should be commented (according to the kernel-doc format):

```
/**
 * foobar() - short function description of foobar
 * @arg1: Describe the first argument to foobar.
 * @arg2: Describe the second argument to foobar.
 *
 * One can provide multiple line descriptions
 * for arguments.
 *
 * A longer description, with more discussion of the function foobar()
 * that might be useful to those using or modifying it. Begins with
 * empty comment line, and may include additional embedded empty
 * comment lines.
 *
 * The longer description can have multiple paragraphs.
 *
 * Return: Describe the return value of foobar.
 */
```

And this, how a structure should be commented:

```
/**
 * struct blah - the basic blah structure
 * @mem1: describe the first member of struct blah
```

```
* @mem2:  describe the second member of struct blah,  
*         perhaps with more lines and words.  
*  
* Longer description of this structure.  
*/
```

The opening comment mark `/**` – showed above – is reserved for kernel-doc comments. Only comments so marked will be considered by the kernel-doc scripts, and any comment so marked must be in kernel-doc format.

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.

Indentation

The whole idea behind indentation is to clearly define where a block of control starts and ends. Indentation must be done using tabular characters, not spaces. Thus, It's important to check whether your text editor, when you press the "TAB" key, effectively puts a *tab character*, or a sequence of space characters. Also, you also should notice how your text editor interprets *tab characters* that are already in the document (how many spaces they show to you). *Tabs are 8 characters*.

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.

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 or assignments on a single line.

Get a decent editor and don't leave whitespace at the end of lines; coding style is all about

readability and maintainability using commonly available tools.

Breaking long lines and strings

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, unless exceeding 80 columns significantly increases readability and does not hide information. 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. However, never break user-visible strings such as print messages, because that breaks the ability to *grep* for them.

Placing Braces and Spaces

```
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*/  
}
```

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 {  
    ....  
}
```

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();
```

and

```
if (condition)  
    do_this();  
else  
    do_that();
```

This does not apply if only one branch of a conditional statement is a single statement; in the latter case use braces in both branches:

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

Spaces

Usage 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 ); /* BAD */
```

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 typedef alignof __attribute__ defined
```

no space before the postfix, or 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.

Naming

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 "countactive_users()" or similar, you should *not* call it "cntusr()".

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.

Typedefs

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 ptet *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".**

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.

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

- **(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.**

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, 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.

In source files, separate functions with one blank line.

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.

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.

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.

Centralized Exiting of Functions


Although 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. If there is no cleanup needed then just return directly.

Choose label names which say what the *goto* does or why the *goto* exists. An example of a good name could be "out_buffer:" if the *goto* frees "buffer". Avoid using GW-BASIC names like "err1:" and "err2:". Also don't name them after the *goto* location like "err_kmalloc_failed:"

The rationale for using gotos 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;  
  
    buffer = kmalloc(SIZE, GFP_KERNEL);  
    if (!buffer)  
        return -ENOMEM;  
  
    if (condition1) {  
        while (loop1) {  
            ...  
        }  
        result = 1;  
        goto out_buffer;  
    }  
    ...  
out_buffer:  
    kfree(buffer);  
    return result;  
}
```

A common type of bug to be aware of it "one err bugs" which look like this:

```
err:
    kfree(foo->bar);
    kfree(foo);
    return ret;
```

The bug in this code is that on some exit paths "foo" is NULL. Normally the fix for this is to split it up into two error labels "err_bar:" and "err_foo:".

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:

- macros that affect control flow:

```
#define F00(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.

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

```
#define F00(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.

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

```
#define CONSTANT 0x4000
#define CONSTEXP (CONSTANT | 3)
```

- namespace collisions when defining local variables in macros resembling functions:

```
#define FOO(x) \
({ \
    typeof(x) ret; \
    ret = calc_ret(x); \
    (ret); \
})
```

`ret` is a common name for a local variable – `__foo_ret` is less likely to collide with an existing variable.

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.

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.

Allocating memory

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.

The preferred form for allocating an array is the following:

```
❏ p = kmalloc_array(n, sizeof(...), ...);
```

The preferred form for allocating a zeroed array is the following:

```
❏ p = kzalloc(n, sizeof(...), ...);
```

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), 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.

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.

Inline assembly

In architecture-specific code, you may need to use inline assembly to interface with CPU or platform functionality. Don't hesitate to do so when necessary. However, don't use inline assembly gratuitously when C can do the job. You can and should poke hardware from C when possible.

Consider writing simple helper functions that wrap common bits of inline assembly, rather than repeatedly writing them with slight variations. Remember that inline assembly can use C parameters.

Large, non-trivial assembly functions should go in .S files, with corresponding C prototypes defined in C header files. The C prototypes for assembly functions should use "asm linkage".

You may need to mark your asm statement as volatile, to prevent GCC from removing it if GCC doesn't notice any side effects. You don't always need to do so, though, and doing so unnecessarily can limit optimization.

When writing a single inline assembly statement containing multiple instructions, put each

instruction on a separate line in a separate quoted string, and end each string except the last with `\n\t` to properly indent the next instruction in the assembly output:

```
asm ("magic %reg1, #42\n\t"  
    "more_magic %reg2, %reg3"  
    : /* outputs */ : /* inputs */ : /* clobbers */);
```

Conditional Compilation

Wherever possible, don't use preprocessor conditionals (`#if`, `#ifdef`) in `.c` files; doing so makes code harder to read and logic harder to follow. Instead, use such conditionals in a header file defining functions for use in those `.c` files, providing no-op stub versions in the `#else` case, and then call those functions unconditionally from `.c` files. The compiler will avoid generating any code for the stub calls, producing identical results, but the logic will remain easy to follow.

Prefer to compile out entire functions, rather than portions of functions or portions of expressions. Rather than putting an `ifdef` in an expression, factor out part or all of the expression into a separate helper function and apply the conditional to that function.

If you have a function or variable which may potentially go unused in a particular configuration, and the compiler would warn about its definition going unused, mark the definition as `__maybe_unused` rather than wrapping it in a preprocessor conditional. (However, if a function or variable *always* goes unused, delete it.)

Within code, where possible, use the `IS_ENABLED` macro to convert a Kconfig symbol into a C boolean expression, and use it in a normal C conditional:

```
if (IS_ENABLED(CONFIG_SOMETHING)) {  
    ...  
}
```

The compiler will constant-fold the conditional away, and include or exclude the block of code just as with an `#ifdef`, so this will not add any runtime overhead. However, this approach still allows the C compiler to see the code inside the block, and check it for correctness (syntax, types, symbol references, etc). Thus, you still have to use an `#ifdef` if the code inside the block references symbols that will not exist if the condition is not met.

At the end of any non-trivial `#if` or `#ifdef` block (more than a few lines), place a comment after the `#endif` on the same line, noting the conditional expression used. For instance:

```
#ifdef CONFIG_SOMETHING  
    ...  
#endif /* CONFIG_SOMETHING */
```

References

"Linux kernel coding style", Linus Torvaldis :

github.com/torvalds/linux/blob/master/Documentation/CodingStyle

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

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

"How to format kernel-doc comments" :

[github.com/torvalds/linux/blob/masterDocumentation/kernel-doc-nano-HOWTO.txt](https://github.com/torvalds/linux/blob/master/Documentation/kernel-doc-nano-HOWTO.txt)