Documentation/CodingStyle and Beyond

Greg Kroah-Hartman
IBM Linux Technology Center
greg@kroah.com — gregkh@us.ibm.com

Abstract

With more companies starting to write Linux kernel code, an understanding of what is the acceptable kernel coding style and conventions is becoming a necessity. The goal of this paper is to explain both the written and unwritten Linux kernel programming style. It explains why a consistent coding style and rules are a requirement for the kernel. It discusses the basic kernel style rules as outlined in `Documentation/CodingStyle` and explains the large number of style rules that are not documented. Each of these rules is documented with existing code, and why the rule is considered a "good thing".

1 Why rules?

Why are there kernel programming style rules in the first place? Why not just let every author code in whatever style they want to, and let everyone live with it? After all, code formatting does not affect memory use, execution speed, or anything else a normal user of the kernel would see. The reason can be summed up with this quote from Elliot Soloway and Kate Ehrlich in 1984[1]

> It is not merely a matter of aesthetics that programs should be written in a particular style. Rather there is a psychological basis for writing programs in a conventional manner: programmers have strong expectations that other programmers will follow these discourse rules. If the rules are violated, then the utility afforded by the expectations that programmers have built up over time is effectively nullified.

A number of other studies and research has proven that if a large body of code is written in a common style, it directly affects how easy it is to quickly understand the code, review it, and revise it.

Since the number of developers that look at the Linux kernel code is very large, it is in the best interest for the project to have a consistent style guideline. This allows the code to be more easily understood either by someone reading it for the first time, or by someone revisiting their old code later. It also allows someone else to more easily read, understand, and potentially fix and enhance your code, which is one of the greatest strengths of open source code.

2 What are the rules?

Now that we have an understanding that there should be some rules, what are they? Linus Torvalds and other kernel programmers have written a short document that details some of the kernel programming rules. This document is located in the *Documentation/CodingStyle* file in the kernel source tree. It is required reading for anyone who wants to contribute to the Linux kernel. Here is a summary of these rules.

2.1 Indentation

All tabs are 8 characters, and will be the `<TAB>` character. This makes it easy to quickly locate where different blocks of code start and end. If you find your code is being indented too deeply, with more than three levels of indentation causing the code to shift off to the right of the screen, then you should fix the code. It is a good warning.

2.2 Placing Braces

The original authors of UNIX placed their braces with the opening brace last on the line, and the closing brace first on the line, like:

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

Because of this, the kernel shall be written in this style.

The exception to this rule are functions, which have the opening brace at the beginning of the line, like:

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

Again, this is how Kernighan and Ritchie wrote their code.

For good examples of the proper indentation and braces style, look at any of the *fs/*.c* files, or anything in the *kernel/*.c* files. Generally, most of the kernel is in the proper indentation and brace style, but there are some notable exceptions. The code in *fs/devfs/*.c* or *drivers/scsi/qla1280.** are good examples of how **not** to do indentation and braces.

There is a script that can be used to run the `indent(1)` program in the proper kernel indentation and braces style. It is useful if you have to convert a large amount of code to the correct format. This file is located at *scripts/Lindent* in the kernel source tree.

2.3 Naming

Your variables and functions should be declared descriptively and concisely. You should not use long flowery names like, `CommandAllocationGroupSize` or DAC960_V1_EnableMemoryMailboxInterface(), but rather, `cmd_group_size`, or `enable_mem_mailbox()`. Your names need to be descriptive, and easily recognized. Mixed case names are frowned upon and encoding the type of the variable or function in the name (like "Hungarian notation") is forbidden.

Global variables should be only used if they are absolutely necessary. Local variables should be short and to the point. `i` and `j` are valid local loop variable names, while `loop_counter` is too verbose. `tmp` is allowed to be used for any short-lived temporary variable.

Again, good examples of proper names can be found in *fs/*.c*. Lots of driver code has bad variable names, as they have been ported from other operating systems. *drivers/block/DAC960.** and *drivers/scsi/cpqfc** are examples of how to **not** name functions and variables.

2.4 Functions

Functions should only do one thing, and do it well. They should be short, and contain one or two screens of text. If you have a function that does lots of small things for different cases, it is acceptable to have a longer function. If you have a complex long function, it should be rewritten to be simpler.

If you have a large number of local variables within a function, it is also a measure of the complexity. If there are more than 10 local variables, it is too complex.

There are lots of good examples of nice sized functions in the *fs/*.c* and other kernel core code. Some bad examples of functions can be found in *drivers/hotplug/ibmphp_res.c* where one function is 370 lines long, or *drivers/usb/usb-uhci.c* where one function has 18 local variables.

2.5 Comments

Comments are very good to have, if they are good comments. Bad comments explain how the code works, who wrote a specific function on a specific date, or other such useless things. Good comments explain what the file or function does, and why it does it. They should be at the beginning of the function, and not necessarily embedded within the function. You are writing small functions, right?

There is now a standard format for function comments. It is a variant of the documentation method used by the GNOME project for their code. If you write your function comments in this style, the information in them can be extracted by a tool and made into stand-alone documentation. This can be seen by running `make psdocs` or `make htmldocs` on the kernel tree to generate a *kernel-api.ps* or *kernel-api.html* file containing all of the public interfaces to the different kernel subsystems.

```
/**
 * function_name(:)? (- short description)?
(* @parameterx: (description of parameter x)?)*
(* a blank line)?
 * (Description:)? (Description of function)?
 * (section header: (section description)? )*
(*)?*/
```

Figure 1: The format of a block comment

This style is documented in the file *Documentation/kernel-doc-nano-HOWTO.txt* and *scripts/kernel-doc*. The basic format can be seen in Figure 1.

The short function description cannot be multi-line, but the other descriptions can be, and they can contain blank lines. All further descriptive text can contain the following markups:

`funcname()` - name of a function
`$ENVVAR` - name of a environment variable
`&struct_name` - name of a structure (up to two words including 'struct')
`@parameter` - name of a parameter
`%CONST` - name of a constant.

A simple example of a function comment with a single argument looks like:

```
/**
 * my_function - does my stuff
 * @my_arg: my argument
 *
 * Does my stuff explained.
 **/
void my_function (int my_arg)
{
        ...
}
```

Comments should be written for structures, unions and enums. The format for them is much like the function format:

```
/**
 * struct my_struct - short description
 * @a: first member
 * @b: second member
 *
 * Longer description
 */
struct my_struct {
        int a;
        int b;
};
```

Some good examples of well commented functions can be found in the *drivers/usb/usb.c* file, where all global functions are documented. The file *arch/i386/kernel/mtrr.c* is a good example of a file with a reasonable amount of comments, but they are in the incorrect format, so they can not be extracted by the documentation tools. *drivers/scsi/pci2220i.c* is also a good example of how **not** to create the comment blocks for your functions.

2.6 Data Structure requirements

The addition of a chapter on data structures, showed up in the 2.4.10-pre7 kernel release. It describes how every data structure that can exist outside of a single-threaded environment, needs to implement reference counting to properly handle the memory management issues. If you add reference counting to your structure, you can avoid lots of nasty locking issues and race conditions. Multiple threads can access the same structure without having to worry that a different thread will free the data from under it.

The last sentence in this chapter is required reading by any kernel developer:

> 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.

A good example of why reference counting is necessary can be found in the USB data structure, `struct urb`. This structure is created by a USB device driver, filled with data, sent to a USB host controller where it will be processed and eventually sent out over the wire. When the host controller is finished with the urb, the original device driver is notified. While a host controller driver is processing the urb, the original driver can try to cancel the urb, or even free it. This led to long detailed arguments on the *linux-usb-devel* mailing list about when in the life span of a urb it was allowed to be touched by either driver, and numerous bugs in the core USB subsystem and different USB drivers.

In the 2.5 kernel series, `struct urb` had a reference count added to it, and the USB core and USB host controller drivers had a small amount of code added to properly handle the reference count. Now whenever a driver wants to use the urb, a reference count is incremented. When it is finished, the reference count is decremented. If this was the last user, the memory is freed, and the urb disappears. This allowed the USB device drivers to vastly simplify their urb handling logic and fixed lots of different race condition bugs. It also made all of the developer's lives simpler by quieting all arguments about the topic.

3 Unwritten rules

If you follow the above set of rules, your code looks like good Linux kernel code. There are quite a few unwritten rules and style guidelines that good kernel code follows. Here are some of them.

3.1 Avoid NIH syndrome

There are a wide variety of well designed, well documented, and well debugged functions and data structures within the kernel. Take advantage of them rather than reinventing your own version. Among the most common of these are the string functions, the byte order functions, and the linked list data structure and functions.

3.2 String functions

In the file, *include/linux/string.h*, a number of common string handling functions are defined. These include:

```
strpbrk
strtok
strsep
strspn
strcpy
strncpy
strcat
strncat
strcmp
strncmp
strnicmp
strchr
strrchr
strstr
strlen
strnlen
memset
memcpy
memove
memscan
memcmp
memchr
```

And in the file, *include/linux/kernel.h*, a number of "simple" string functions are defined:

```
simple_strtoul
simple_strtol
simple_strtoull
simple_strtoll
```

If you need any type of string functionality in your kernel code, use the built in functions. Do not try to rewrite the existing functions accidentally.

3.3 Byte order handling

Do not rewrite code to switch data between different endian representations. The file *include/asm/byteorder.h* (*asm* will point to the proper subdirectory, depending on your processor architecture) brings in a wide range of functions that allow you to do automatic conversions, no matter what the endian format of your processor or your data.

3.4 Linked Lists

If you need to create a linked list of any kind of data structure, use the code that is in *include/linux/list.h*. It contains a structure, `struct list_head`, that should be included within the structure for the new list. You can easily add, remove, or iterate over a list of data structures, without having to write new code.

Some good examples of code that uses the list structure can be found in *drivers/hotplug/pci_hotplug_core.c* and *drivers/ieee1394/nodemgr.c*. Some code in the kernel that should be using the list structure, can be found in the ATM core, within the `struct atm_vcc` data structure. Because the ATM code did not use `struct list_head`, every ATM driver needs to walk the lists of data structures by hand, duplicating lots of code.

3.5 `typedef` is evil

`typedef` should not be used in naming any of your structures. Almost all main kernel structures do not have a `typedef` to shorten their usage. This includes the following:

```
struct inode
struct dentry
struct file
struct buffer_head
struct user
struct task_struct
```

Using `typedef` tries to hide the real type of a variable. There have been records of some kernel code using typedefs nested up to 4 layers deep, preventing the programmer from telling what type of variable they are really using. This can easily cause very large structures to be accidentally declared on the stack, or to be returned from functions if the programmer does not realize the size of the structure.

`typedef` can also be used as a crutch to keep from typing long structure definitions. If this is the case, the structure names should be made shorter, according to the above listed naming rules.

Never define a `typedef` to just signify a pointer to a structure, as in the following example:

```
typedef struct foo {
        int bar;
        int baz;
} foo_t, *pfoo_t;
```

This again hides the true type of the variable, and is using the name of the variable type to define what is is (see the comment about Hungarian notation previously.)

Some examples of where `typedef` is badly used are in the *include/raid/md*.h* files where every structure has a `typedef` assigned to it, and in the *drivers/acpi/include/*.h* files, where a lot of the structures do not even have a name assigned to them, only a `typedef`.

The only place that using `typedef` is acceptable, is in declaring function prototypes. These can be difficult to type out every time, so declaring a typedef for these is nice to do. An example of this is the `bh_end_io_t typedef` which is used as a parameter in the `init_buffer()` call. This is defined in *include/fs.h* as:

```
typedef void (bh_end_io_t)(struct buffer_head *bh,
                           int uptodate);
```

3.6 No magic numbers

The Jargon file[2] describes a magic number within source code as:

> In source code, some non-obvious constant whose value is significant to the operation of a program and that is inserted inconspicuously in-line (hardcoded), rather than expanded in by a symbol set by a commented `#define`. Magic numbers in this sense are bad style.

Fortunately the kernel does not have many instances of code that uses magic numbers. The *drivers/usb/serial/pl2303.c* driver used to have the code shown in Figure 2 in the `open()` function. This code contains a lot of of different magic numbers. The current version of the file can be seen in Figure 3. Even with the odd use of the macros `FISH()` and `SOUP()`, some of the magic numbers have been replaced with the more descriptive `VENDOR_READ_REQUEST_TYPE`, `VENDOR_READ_REQUEST`, `VENDOR_WRITE_REQUEST_TYPE` and `VENDOR_WRITE_REQUEST`. This code could be cleaned up a lot more, detailing what the other magic numbers mean. Unfortunatly the driver was written by reverse engineering a protocol stream captured from a computer running a different operating system. Most of these numbers' true purpose are not known, only that they are necessary.

```
#define FISH(a,b,c,d) \
    i = usb_control_msg (serial->dev, \
        usb_rcvctrlpipe(serial->dev,0), \
        b, a, c, d, buf, 1, 100); \
    dbg("0x%x:0x%x:0x%x:0x%x  %d - %x", \
        a,b,c,d,i,buf[0]);

#define SOUP(a,b,c,d) \
    i = usb_control_msg(serial->dev, \
        usb_sndctrlpipe(serial->dev,0), \
        b, a, c, d, NULL, 0, 100); \
    dbg("0x%x:0x%x:0x%x:0x%x  %d", \
        a,b,c,d,i);

    FISH (0xc0, 1, 0x8484, 0);
    SOUP (0x40, 1, 0x0404, 0);
    FISH (0xc0, 1, 0x8484, 0);
    FISH (0xc0, 1, 0x8383, 0);
    FISH (0xc0, 1, 0x8484, 0);
    SOUP (0x40, 1, 0x0404, 1);
    FISH (0xc0, 1, 0x8484, 0);
    FISH (0xc0, 1, 0x8383, 0);
    SOUP (0x40, 1, 0, 1);
    SOUP (0x40, 1, 1, 0xc0);
    SOUP (0x40, 1, 2, 4);
```

Figure 2: Original version of *pl2303.c*

3.7 No `ifdef` in .c code

With the wide number of different processors, different configuration options, and variations of the same base hardware types that Linux runs on, it is very easy to start having a lot of `ifdef` statements in your code. This is not the proper thing to do. Instead, place the `ifdef` in a header file, and provide empty inline functions if the code is not to be included.

As an example, consider the code in *drivers/usb/hid-core.c* as shown in Figure 4.

Here the author does not want to call `hiddev_hid_event()` if a specific configuration option is not enabled. This is because that function is not present if the configuration option is not enabled.

To remove this `ifdef`, the changes shown in Figure 5 were made.

If `CONFIG_USB_HIDDEV` is not enabled, the compiler replaces the call to `hiddev_hid_event()` with a null function call, and then optimizes away the if statement entirely. This keeps the code readable and is much easier to maintain.

```
#define FISH(a,b,c,d) \
    i = usb_control_msg (serial->dev, \
        usb_rcvctrlpipe(serial->dev,0), \
        b, a, c, d, buf, 1, 100); \
    dbg("0x%x:0x%x:0x%x:0x%x  %d - %x", \
        a,b,c,d,i,buf[0]);

#define SOUP(a,b,c,d) \
    i = usb_control_msg(serial->dev, \
        usb_sndctrlpipe(serial->dev,0), \
        b, a, c, d, NULL, 0, 100); \
    dbg("0x%x:0x%x:0x%x:0x%x  %d", \
        a,b,c,d,i);

FISH (VENDOR_READ_REQUEST_TYPyE,
      VENDOR_READ_REQUEST, 0x8484, 0);
SOUP (VENDOR_WRITE_REQUEST_TYPE,
      VENDOR_WRITE_REQUEST, 0x0404, 0);
FISH (VENDOR_READ_REQUEST_TYPE,
      VENDOR_READ_REQUEST, 0x8484, 0);
FISH (VENDOR_READ_REQUEST_TYPE,
      VENDOR_READ_REQUEST, 0x8383, 0);
FISH (VENDOR_READ_REQUEST_TYPE,
      VENDOR_READ_REQUEST, 0x8484, 0);
SOUP (VENDOR_WRITE_REQUEST_TYPE,
      VENDOR_WRITE_REQUEST, 0x0404, 1);
FISH (VENDOR_READ_REQUEST_TYPE,
      VENDOR_READ_REQUEST, 0x8484, 0);
FISH (VENDOR_READ_REQUEST_TYPE,
      VENDOR_READ_REQUEST, 0x8383, 0);
SOUP (VENDOR_WRITE_REQUEST_TYPE,
      VENDOR_WRITE_REQUEST, 0, 1);
SOUP (VENDOR_WRITE_REQUEST_TYPE,
      VENDOR_WRITE_REQUEST, 1, 0xc0);
SOUP (VENDOR_WRITE_REQUEST_TYPE,
      VENDOR_WRITE_REQUEST, 2, 4);
```

Figure 3: Current version of *pl2303.c*

3.8 Labeled Elements in Initializers

`gcc` allows the use of labeled elements in initializers. This means that structures that are initialized at compile time can have the individual field names used to specify what fields to set. For example, if the `struct foo` structure was defined as:

```
static void hid_process_event (struct hid_device *hid, struct hid_field *field,
                               struct hid_usage *usage, __s32 value)
{
        hid_dump_input(usage, value);
        if (hid->claimed & HID_CLAIMED_INPUT)
                hidinput_hid_event(hid, field, usage, value);
#ifdef CONFIG_USB_HIDDEV
        if (hid->claimed & HID_CLAIMED_HIDDEV)
                hiddev_hid_event(hid, usage->hid, value);
#endif
}
```

Figure 4: Original version of `drivers/usb/hid-core.c`

include/linux/hiddev.h:

```
#ifdef CONFIG_USB_HIDDEV
        extern void hiddev_hid_event (struct hid_device *, unsigned int usage, int
value);
#else
        static inline void hiddev_hid_event (struct hid_device *hid, unsigned int usage,
int value) { }
#endif
```

drivers/usb/hid-core.c:

```
static void hid_process_event (struct hid_device *hid, struct hid_field *field,
                               struct hid_usage *usage, __s32 value)
{
        hid_dump_input(usage, value);
        if (hid->claimed & HID_CLAIMED_INPUT)
                hidinput_hid_event(hid, field, usage, value);
        if (hid->claimed & HID_CLAIMED_HIDDEV)
                hiddev_hid_event(hid, usage->hid, value);
}
```

Figure 5: After removal of `ifdef` in *drivers/usb/hid-core.c*

```
struct foo {
        int a;
        int b;
        int c;
};
```

any static definition of a variable of this type would traditionally be written as:

```
static struct foo bar =
        {A_INIT, B_INIT, C_INIT};
```

With the `gcc` extension, this initialization could also be written as:

```
static struct foo bar = {
        a: A_INIT,
        b: B_INIT,
        c: C_INIT,
};
```

which is a lot more descriptive. If a field is not specified with a specific value, the compiler sets that field to zero.

The kernel is filled with large structures, and lots of them are initialized at compile time. Previously, if someone added a new field in a structure, any variables that were declared like the previous example would break. For example, if the `struct foo` structure was changed to be:

```
struct foo {
        int a;
        char a1;
        int b;
        int c;
};
```

Any place that did not use labeled elements would break.

A good example of this, is any file that declares a `struct file_operations` variable. Generally, you do not want to define all fields of this structure, but rely on the VFS core to handle the majority of operations. The file *drivers/char/raw.c* has two good examples of named initializers:

```
static struct file_operations raw_fops = {
        read:           raw_read,
        write:          raw_write,
        open:           raw_open,
        release:        raw_release,
        ioctl:          raw_ioctl,
};

static struct file_operations
    raw_ctl_fops = {
        ioctl:          raw_ctl_ioctl,
        open:           raw_open,
};
```

The code in Figure 6 from *arch/ia64/sn/io/hcl.c* is a good example of how much overhead is involved if you do not use this style of code.

This file will have to be updated every time the `struct file_operations` structure changes in the future.

```
struct file_operations hcl_fops = {
        (struct module *)0,
        NULL,           /* lseek - default */
        NULL,           /* read */
        NULL,           /* write */
        NULL,           /* readdir - bad */
        NULL,           /* poll */
        hcl_ioctl,      /* ioctl */
        NULL,           /* mmap */
        hcl_open,       /* open */
        NULL,           /* flush */
        hcl_close,      /* release */
        NULL,           /* fsync */
        NULL,           /* fasync */
        NULL,           /* lock */
        NULL,           /* readv */
        NULL,           /* writev */
};
```

Figure 6: *arch/ia64/sn/io/hcl.c*

4 Conclusion

The Linux kernel consists of a very large amount of source code, contributed by hundreds of developers over many years. Since the majority of this code follows some simple and basic style and formatting rules, the ability for people to quickly understand new code has been greatly enhanced. If you want to contribute to this code, please read the *Documentation/CodingStyle* guidelines and follow them in your patches and new code. The "unwritten" rules can be just as important as the written ones, when you are trying to convince people to accept your contributions, and should be followed just as closely.

5 Trademarks

IBM is a trademark of International Business Machines Corporation.

Linux is a trademark of Linus Torvalds.

Other company, product or service names may be trademarks or service marks of others.

This work represents the view of the author and does not necessarily represent the view of IBM.

References

[1] Soloway, Elliot, and Kate Ehrlich. 1984. "Empirical Studies of Programming Knowledge", IEEE Transactions on Software Engineering SE-10, no. 5 (September): 595-609

[2] `http://www.tuxedo.org/ esr/jargon/`