Coding Rules

Section A: Linux kernel style based coding for C programs

Coding style for C is based on Linux Kernel coding style. The following excerpts in this section are mostly taken *as is* from articles written by Greg Kroah-Hartman on Linux coding style. It is required that the relevant parts of the articles are clearly understood. Given below is the condensed set of rules:

Indentation

Tabs are 8 characters, and thus indentations are also 8 characters.

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 most likely made a mistake 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.

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. All non-function blocks are thus to be written as:

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

```
while (x != y) {
        ...
}
```

```
struct node {
        ...
};
```

Functions are written as in K&R style as below. ' ' represents a single blank space.

```
int function(int x)
{
        ...
}
```

Naming

Mixed-case names (like `ThisVariableIsATemporaryCounter`) 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 forbidden - it only confuses the programmer.
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 misunderstood. Similarly, "`tmp`" can be just about any type of variable that is used to hold a temporary value.

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), 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.

White spaces and general guidelines

Blank spaces are to be used sensibly. The following are the general guidelines.

1. Conditions should be written clearly and preferably using positive check, instead of preferring not operator. The following is not a good approach, and probably is a mistake difficult to spot unless one correctly understands the implications.
   ```
   if (!rec->data = b && c | d)
   ```
2. While assigning values, use single blank space on both sides.
   ```
   a = ((float) b + c *d)/z;
   ```
3. No blank spaces should be placed before the terminating semi-colon.
4. Assigning values during declaration should be avoided.
5. Blank lines are to be placed when it would clearly separate tasks.
6. No space should be placed before comma but should be followed by a single space.

Commenting

Comments are good, but there is also a danger of over-commenting.

1. **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.
2. Generally, you want your comments to tell **what** your code does, not **how**.
3. 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 **Functions** section for a while.
4. 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.

The short function description cannot run multiple lines, but the other descriptions can (and they can contain blank lines). All further descriptive text can contain the following markups:

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

So a simple example of a function comment with a single argument would be:

```
/**
 * my_function - does my stuff
 * @my_arg: my argument
 *
 * Does my stuff explained.
 */
```

Comments can and 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;
};
```

`typedef`

`typedef` should **not** be used in naming any of your structures. Most main kernel structures have no `typedef`. Using a `typedef` only hides the real type of a variable. There are records of some kernel code using `typedefs` nested up to four layers deep, preventing the programmer from easily telling what type of variable they are actually using. If the programmer does not realize the size of the structure it can cause very large structures.

Section B: Building and testing of a program

1. Never begin to start keying-in the program unless pseudocode is written.

2. If possible, make a dry run over the pseudocode and see if it at least **seems** to do what is required of it.

3. Divide your program into units that can be compiled or checked **while** building the code to ensure things going smoothly. It is a good practice to check that at the end of each unit, one can verify if **progress** in the **correct direction** is being made. The identification of units, hence, themselves is a subjective matter and can be done easily when one has complete idea about the entire problem in the form of a pseudocode.

An example of units for the construction of a program to print a number in Fibonacci series is given below:

1. Build the minimal valid program (*ie,* a program that returns '0').
2. Declare the function template, say Fib(int i), and the function Fib that returns '0'.
3. Write the function and check it with a value, say '3'. It is expected to return '2'.
4. Take a number from user and verify the output using the function.

Section C: Sample code

```
#include <stdio.h>

/**
 * struct rec - stores marks obtained in various subjects
 * @icp: marks obtained in ICP
 * @dfs: marks obtained in DFS
 * @total: total marks
 */
struct rec {
        int icp;
        int dfs;
        int total;
};

int add_num(int *, int *);
void add_marks(struct rec *);

int main()
{
        int b;
        int c;
        struct rec b2020020;

        b = 100;
        c = 200;
        /* assign marks */
        b2020020.icp = 60;
        b2020020.dfs = 70;

        printf("add_num: %d + %d = %d\n\n", b, c, add_num(&b, &c));

        add_marks(&b2020020);
        printf("rec.total = %d\n\n", b2020020.total);

        return 0;
}

int add_num(int *x, int *y)
{
        return (*x + *y);
}

/**
 * add_marks - finds rec.total
 * @record: &struct rec
 *
 * We take the student record and add the marks to total
 */
void add_marks(struct rec *record)
{
        record->total = record->icp + record->dfs;
}
```

Section D: Pseudocode

The following are the rules for writing pseudocodes. Leave a constant amount of space for indentation. Inner block of statements are always pushed to left by a constant amount of white space.

`if` statement: The `if` statement may or may not have '`else if`' and `else` cases.

```
if (condition)
        block of statements
else if (condition)
        block of statements
else if (condition)
        block of statements
else
        block of statements
```

`while` loop: The `while` loop should be written as:

```
while (condition)
      block of statements
```

`for` loop: The for loop could be used either in increasing or in decreasing order for a single counter. The value of the counter changes by one.

```
for i = min to max
      block of statements
```

```
for i = max downto min
      block of statements
```

`struct`: A **`struct`** should be written as:

```
struct struct_name
      type members
```

Section E: Dry run

Dry run should have three columns, *viz.*, the pseudocode, memory state showing the values of various constants and variables and output. The following illustrates the example. Each change is written from top-to-bottom sequentially.

Pseudocode	**Memory state**	**Output**
`x = 2`	`x = 2`	
`for i = 1 to 5`	`i = 1`	
`  if ((i % x) == 0)`		`Odd`
`    print x`	`i = 2`	
`  else`		`2`
`    print "Odd"`	`i = 3`	
		`Odd`
	`i = 4`	
		`4`
	`i = 5`	
		`Odd`

Section F: Sample code snippets for C++ specific code

The following are the code snippets for writing parts of code specific to C++.

1. Creating templates

```
template<typename T1, typename T2>
T1 add(T1 x, T2 y)
{
      T1 z = x + (T1) z;
      return z;
}
```

2. Using templates

```
vector<int> v1;
list<vector<int> > li;
map<string, string>::iterator mit;
```