VVDN Coding Style (INDx_TRNG_01)

Rev: A0-01

21 June 2013

VVDN Coding Style

‘C’ coding standards for software projects

VVDN Contact:
manas.mohanta@vvdntech.com
sethuraman.thangavelu@vvdntech.com

Revision History:

Date	Rev No.	Description	By
21-06-2013	A0-01	First Draft	Manas & Sethu

"Difference that makes a professional"

- ***Professional can replicate performance***
- ***Professionals have a plan, and follow the plan***
- ***Professional do what they do better than almost everyone else. "If it is worth doing, it is worth doing well"***
- ***Professional know more about what they are doing than almost everyone else "It is the knowing & doing that counts"***
- ***Professionals will always try to improve by exerting continuous effort to learn more /practice more and get better at what they do***
- ***Professional don't let their feelings interfere with their performance. "The way they feel about doing something is separate from the importance of doing it or the actual doing of it"***

Table of Contents

1 Introduction

This document describes necessary guidelines for Coding Style. This should be followed by all projects executed at VVDN.

A coding standards document tells developers how they must write their code. Instead of each developer coding in their own preferred style, they will write all code to the standards outlined in the document. This makes sure that a large project is coded in a consistent style — parts are not written differently by different programmers.

"The best applications are coded properly. This sounds like an obvious statement, but by 'properly', I mean that the code not only does its job well, but is also easy to add to, maintain and debug." When you start sharing code, or start reading code shared by others, you begin to realize that not everybody writes their code they way you do. You see that other, ugly coding style, and think "everybody needs to write in the same style so that things are easier for me to understand."

Thus, it is natural that everybody wants their own habits turned into the standard, so they don't have to change their habits. They're used to reading code a certain way (their own way) and get irritated when they see the code in a different format. The thing about defining a coding style standard is that there is no objective means by which to judge one style as "better" or "more-right" than another. Sure, we can talk about "readability" and "consistency" but what is readable is different for each coder (depending on what they're used to) and consistency follows automatically because, well, why would you use another style.

If we make standard coding as our habit then it will be reliable for all to read, maintain, reuse and modify the program even when original programmer is absent. It will not only save time but also reduce unnecessary effort to understand while working with others code.

2 CODING DECISIONS

- For comparing only true or false and assign some values then instead using if-else, effective way is to use ternary operator.

```
Example:

Instead of using if-else as below:
if (x == TRUE) {
    result = 10;
} else {
    result = 20;
}

We can use the conditional operator as follows:
 result = (x == TRUE) ? 10 : 20;
```

- Instead of using 'while' with some iterative logic, we can use 'for' like that you collect all the types and represent the effective coding.

```
Example:

Instead of using while as follows:
i = 0;                      /* initialize */
while (i <= 10) {           /* condition */
    printf("i = %d \n",i);
    i++;                    /* increment */
}

We can use for:
for (i = 0;i <= 10;i++) {  /* initialize, condition, increment */
      printf("i=%d \n",i);
}
```

Following contain the effective usage of functions, macros, typedefs, conditional and looping statements.

2.1 FUNCTIONS

- Functions should be short and sweet, and do just one thing.
- They should fit on one or two screens full 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.
- Use helper functions with descriptive names
- Another measure of the function is the number of local variables. They shouldn't exceed 5-10, or you're doing something wrong. Rethink the function, and split it into smaller pieces.

NOTE: 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.

```
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.

2.1.1 FUNCTIONS 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.
- Lets follow the following standard
    - For SUCCESS, return 0
    - For ERROR, return < 0

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.

2.1.2 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;
}
```

2.2 MACROS

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 FUNCTION(x)         \
do {                        \
    if (FUNCTION(x) < 0){   \
        return -EBUGGERED;  \
    }                       \
} while(0)
```

It 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)
```

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

Example 1:

```
#include<stdio.h>
#define EVAL(x,y) ({(a > b ? 1: 0);0;})
int main(void)
{
    int a = 7,b = 6,ret;
    ret = EVAL(a,b);
    if (ret) {
        printf("Gratest : %d\n",((a > b) ? a : b));
    } else {
        printf("Sum :%d\n",(a + b));
    }
}
```

Example 2: (Usage of Bitwise operator in Macro)

```
/* the flag definitions*/
#define CAR_LOCKED    0x01  /* 0000 0001*/
#define CAR_PARKED    0x02  /* 0000 0010*/
#define CAR_RESET     0x00  /* 0000 0000*/

/* macros to manipulate the flags*/
#define RESET_CAR(x) (x = CAR_RESET)
#define SET_LOCKED(x)  (x |= CAR_LOCKED)
#define SET_PARKED(x)  (x |= CAR_PARKED)
#define UNSET_LOCKED(x)  (x &= (~CAR_LOCKED))
#define UNSET_PARKED(x)  (x &= (~CAR_PARKED))
#define TOGGLE_LOCKED(x)  (x ^= CAR_LOCKED)
#define TOGGLE_PARKED(x)  (x ^= CAR_PARKED)

/* these evaluate to non-zero if the flag is set*/
#define IS_LOCKED(x) (x & CAR_LOCKED)
#define IS_PARKED(x) (x & CAR_PARKED)

/* a short program that demonstrates how to use the macros*/
int main(void)
{
    unsigned char fMercedes, fCivic;

    RESET_CAR(fMercedes);
    RESET_CAR(fCivic);
    SET_LOCKED(fMercedes);
    if ( IS_LOCKED(fMercedes) != 0 ) {
        UNSET_PARKED(fCivic);
    }
    TOGGLE_LOCKED(fMercedes);
    return 0;
}
```

2.2.1 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.

2.3 TYPEDEFs

Please don't use things like "vps_t". It's a mistake to use typedef for structures and pointers. When you see ‘var’

```
vps_t var;
```

in the source, what does it mean?

In contrast, if it says

```
struct virtual_container *var;
```

you can actually tell what "var" is.

Else name the typedef such that it describes the type of the variable , so for the above structure we can name the typedef as

```
s_vps_t var;
```

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

b) 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.

c) Clear integer types, where the abstraction _helps_ avoid confusion whether it is "int" or "long". uint8_t / uint16_t / uint32_t 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.

d) When you use sparse to literally create a _new_ type for type-checking.

e) 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 uint8_t / uint16_t / uint32_t types and their signed equivalents which are identical to standard types are permitted -- although they are not mandatory in the 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.

f) Types safe for use in user space. In certain structures which are visible to user space, 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 user space. 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.

2.4 CONDITIONAL AND LOOPING STATEMENTS

2.4.1 IF ELSE

Example 1:

```
if(cmd_inbuf->start > alm_history.count) {
    ALM_TRACE(LOG_WARNING,
              "Err. Alarm Start index [%d] > Alarm history size: [%d]\n",
              cmd_inbuf->start, alm_history.count);
    cmd_inbuf->count = 0;
    err = -EFAULT;
}
```

Example 2:

```
/* Board is of idu4 type */
if (((idu_brd_type == SIDU0)|| (idu_brd_type == MIDU0)||
     (idu_brd_type == SIDU2)|| (idu_brd_type == MIDU2))
    && (oneplusone_cfg)) {
    idu4_odu_led_cfg(msg);
} else if (((idu_brd_type == GSIDU8R1) || (idu_brd_type == GSIDU4R1) ||
            (idu_brd_type == GSIDU2R1) || (idu_brd_type == GSIDU1R1) ||
            (idu_brd_type == GMIDU8R1) || (idu_brd_type == GMIDU4R1) ||
            (idu_brd_type == GMIDU2R1) || (idu_brd_type == GMIDU1R1)) &&
           (oneplusone_cfg)) {
    du8_odu_led_cfg(msg);
}
```

2.4.2 SWITCH

```
switch (p_api_hdr->cmd_opcode) {
case RA_CM_CMD_CREATE_LINK:
    p_tx_api_hdr->cmd_opcode = CM_RA_RES_CREATE_LINK;
    break;
case RA_CM_CMD_DEL_LINK:
    p_tx_api_hdr->cmd_opcode = CM_RA_RES_DEL_LINK;
    break;
case RA_CM_CMD_STATE_CHNG_LINK:
    p_tx_api_hdr->cmd_opcode = CM_RA_RES_STATE_CHNG_LINK;
    break;
case RA_SM_CMD_GET_LINK_STAT:
    p_tx_api_hdr->cmd_opcode = SM_RA_RES_LINK_STAT;
    break;
case RA_CM_CMD_GET_LINK_CONFIG:
    p_tx_api_hdr->cmd_opcode = CM_RA_RES_GET_LINK_CONFIG;
    break;
default:
    break;
}
```

2.4.3 DO WHILE

```
do {
    (result)->tv_sec = (a)->tv_sec - (b)->tv_sec;
    (result)->tv_usec = (a)->tv_usec - (b)->tv_usec;
    if ((result)->tv_usec < 0) {
        --(result)->tv_sec;
        (result)->tv_usec += 1000000;
    }
} while(0);
```

3 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.

4 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 page cache. 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 compile time constant, and as a result of these constants 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.

```
Example 1:
static inline uint32_t image_get_header_size (void)
{
    return (sizeof (image_header_t));
}

Example 2:
static int inline is_all_healthy(node_info_t *node)
{
    int all = local_healthy | remote_healthy;
    if (is_stack_present(node)) {
        all |= stack_healthy;
    }
    return ((node->health & all) == all);
}
```

5 NEED OF DYNAMIC ALLOCATING

The kernel provides the following general purpose memory allocators:

```
kmalloc(), kzalloc(), kcalloc(), vmalloc(), and vzalloc().
```

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.

```
Example :
int get_sw_vlan_config()
{
    CMD_INTERFACE_T *p_cmd;    /* structure for the complete message
                                  including the header. */
    p_cmd = malloc(buff_size); /* allocating the total memory */
    if (!p_cmd) {
        printf("[GET_SYS_CFG]Malloc fails");
        exit(1);
    }
}
```