VVDN Coding Indentation Style (INDx_TRNG_01)

Rev: A0-01

21 June 2013

VVDN Coding Indentation Style

'C' program indentation style for software projects

VVDN Contact:

praveen.raja@vvdntech.com

Revision History:

Date	Rev No.	Description	By
21-06-2013	A0-01	First draft	Praveen

"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 recommended guidelines for indentations of codes. This shall be followed for all projects executed at VVDN.

1.1 Need for Indentations

When people learn programming, they learn best when they attempt simple programs of minor utility. An unfortunate consequence of this is that the programs are rarely revisited by their authors. Try it sometime; dig out one of your old programs and see if you can easily figure out the details of its operation. If you used a poor indentation style, you will probably have trouble following the logic of the code. And it's your own code! Embarrassing, isn't it?

Now think about larger, more complex programs. They tend to be written to do useful chores, and because they are useful they tend to "hang around" for a long time. Then, eventually, someone decides that the program would be even more useful if it did a few more chores in addition to its current duties. Before long, an unfortunate programmer will be assigned to make the necessary additions to the program, and they will struggle with the task if the code is hard to comprehend.

What could be worse? YOU could be that unfortunate programmer!

The moral: When you write code, no matter how insignificant or trivial it may seem to you now, write it to be easy to read. In particular, adopt a standard indentation style for your code, and stick with it throughout the program. Other programmers will thank you for it.

1.2 Important guidelines for Indentations

1) Always use <8 spaces> for the blocks of code within a function/statement from the starting point of the corresponding function/statement.

```
/* See how there is a 8 space indentation within the function block
*/
static void write_to_alm_fifo (const char *buffer, int len)
{
<4 spaces> kfifo_put(buf, (unsigned char*)buffer, len);
          wake_up(&wqh);
}
```

2) Always use an empty line between declaration of header files & function prototypes and the starting of the main()/other functions & declarations.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define PI 21/7

int g_variable;
/*
 * See how the blank line seperates the header files ,
 * global variables and #defines from the main()
 */
int main()
{
<4> ......
    ......
}
```

3) Similarly use an empty line after defining all the local variables in your main().

```
#include "tdmoip_alarm.h"
#include "wdt_mgr_internal.h"

#ifdef DEBUG
#define WDT_TRACE(lvl, fmt...) printf("[WDT] " fmt);
#else
#define WDT_TRACE syslog
#endif
#define REMOTE_IP "127.0.0.1"
/* See how the blank line seperates the header files ,global
variables and #defines from the main() */

int main()
{
    int input1,input2,input3;
    char name;
    float avg;

    /*
    * blank line seperates the local
    * variables declarations from the
    * body of the main ()
    */
    led1_green_on();
    led2_green_on();
    init_dl_list(&client_list);
    proc_wdt_mgr_cmds_init();
    approve_pending = get_image_approval_status();
    return 0;
}
```

4) All functions, including main() and user defined functions have their opening braces "{" at the beginning of the next line. Here is an example to make you understand.

```
int main()
{                                   /* "{" starts here */
    led1_green_on();
    led2_green_on();
    init_dl_list(&client_list);
    proc_wdt_mgr_cmds_init();
    approve_pending = get_image_approval_status();
}                                   /* "}" it ends here */
```

5) Incase of other Control statements and looping statement, it is preferred to use the opening braces "{" immediatelynext to those statements after giving a single space.

Template:

```
if<1>(x > 0)<1> {            /* start "{" here */
<4> action:
    action;
}                            /* end "}" here */
```

Example:

```
if ((index = cmdopcode_to_index(cmdbuf->cmd_opcode)) >= 0) {
    err = proc_wdt_mgr_commands[index] (cmdbuf);
    exit(0);
}
```

Template:

```
while<1>(TRUE)<1>{
<4> action;
    action;
}
```

Example:

```
while (WDT_RESET!=0) {
    cmdhdr->module_id = MOD_ID_WDT_MGR;
    cmdhdr->cmd_seq_number = cmdin->cmd_seq_number;
    cmdhdr->cmd_exec_status = cmd_exec_status;
    cmdhdr->buff_length = sizeof(CMD_INTERFACE_T) + sizeof(*cmdbuf);
    WDT_TRACE(LOG_DEBUG,
              "sending %d bytes to module %04x\n",
              sizeof(*cmdhdr) + sizeof(*cmdbuf),
              cmdhdr->module_id);
    cmgr_send_buff_to_module( void );
}
```

Template:

```
for<1>(i=0; i<10; i++)<1>{
<4> action;
    action;
}
```

Example:

```
for (count=0;count<WDT_LIMIT;count++) {
    cmdhdr->module_id = MOD_ID_WDT_MGR;
    cmdhdr->cmd_seq_number = cmdin->cmd_seq_number;
    cmdhdr->cmd_exec_status = cmd_exec_status;
}
```

Template:

```
do<1>{
<4> action;
    action;
}<1>while<1>(1);
```

Example:

```
do {
    cmdhdr->module_id = MOD_ID_WDT_MGR;
    cmdhdr->cmd_seq_number = cmdin->cmd_seq_number;
    cmdhdr->cmd_exec_status = cmd_exec_status;
} while(WDT_REST);
```

Template:

```
switch <1>(case)<1>{
case 1:
<4> action 1;
case 2:
    action 2;
case 3:
    action 3;
default:
    default action;
}
```

Example:

```
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_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;
}
```

6) Do use "{" & "}" even for a single line of code after condition statements like if, if else and looping statements like for, while, do while etc.

Template:

```
if<1>(condition)<1>{
<4> do_that;
} else {
<4> do_this;
}
```

```
Example:
if (cli->cli.module_id == cmdin->module_id) {
    cli->dead_count = 0;
} else {
    return 0;
}
```

1.2 Breaking long lines

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.

```
Example 1:
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;
    }
}

Example 2:
very_long_function_name(
        very_long_type_identifier_1 * very_long_argument_name_1,
        very_long_type_identifier_2 * very_long_argument_name_2)
```

1.3 Usage of Space

1) 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 "structfileinfo info;" is declared).

   So use a space after these keywords:

   if, switch, case, for, do, while /* These conditions are explained above */

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

   Example:

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

2) Do not add spaces around (inside) parenthesized expressions. This example shows the wrong usage of spaces.

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

3) 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 longmemparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

4) Use one space around (on each side of) most binary and ternary operators, such as any of these:
= + - <> * / % | & ^ <= >= == != ? :

but no space after unary operators:

& * + - ~ ! sizeoftypeofalignof __attribute__ defined

no space after the prefix increment & decrement unary operators:

++ --

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