VVDN Variable Nomenclature (INDx_TRNG_01)

Rev: A0-01

21 June 2013

VVDN Variable Nomenclature

'C' software project variable naming standards

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 describesthe recommended guidelines fornaming a variable in codes. This shall be followed for all projects executed at VVDN.

1.1 Rules for naming a variable

The following is a set of rules that must be followed while naming a variable in C language.

1) A Variable name consists of any combination of alphabets, digits and underscores. Some compiler allows variable names whole length could be up to 247 characters. Still it would be safer to stick to the rule of 31 characters. Please avoid creating long variable name as it adds to your typing effort.

2) The first character of the variable name must either be alphabet or underscore. It should not start with the digit.

3) No commas and blanks are allowed in the variable name.

4) No special symbols other than underscore are allowed in the variable name.

The compiler would accept any variable name which satisfies the above criteria but to know how to name your variable smartly so that it adds meaning and readability to your code, read on.

```
char n[10];             /* wrong way */
char name[MAX_LEN];     /* a better way to do it */
```

1.2 Why do we need meaningful names?

"Meaningful" is the keyword when it comes to naming. By meaningful names, I mean concise names that accurately describe the variable, method or object. Giving a variable meaningful name goes a long way in making the code reusable and easy to understand. Naming a variable “input_num” instead of “a” gives more meaning and makes the purpose of that variable clear.

1.3 Naming Local variables

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

1.4 Naming pointer variables

Make sure that you always place the “*” right next to the variable name and not next to the data type.

```
char* name;           /* wrong way */
char *name;           /* right way */
```

1.5 Naming Global variables

Global variables should be prepended with a 'g_'. This is to ensure that the variable is interpreted as Global .

It's important to know the scope of a variable.

```
int  g_log;                    /*always prefix with " g_"  */
int *g_plog;                   /*For pointer types use "g_p" */
```

1.6 Naming Global constants

Global constants should be all caps with '_' separators.

It's tradition for global constants to named this way. You must be careful to not conflict with other global #defines and enum labels.

```
const int A_GLOBAL_CONSTANT= 5;
```

1.7 Naming #define and macros

Put #defines and macros in all upper using '_' separators. Macros are capitalized, parenthesized, and should avoid side-effects. Spacing before and after the macro name may be any whitespace, though the use of TABs should be consistent through a file. If they are an inline expansion of a function, the function is defined all in lower case, the macro has the same name all in uppercase. If the macro is an expression, wrap the expression in parenthesis. If the macro is more than a single statement, use ``do { ... } while (0)", so that a trailing semicolon works. Right-justify the backslashes; it makes it easier to read.

This makes it very clear that the value is not alterable and in the case of macros, makes it clear that you are using a construct that requires care.

```
#define MAX(a,b)
#define IS_ERR(err)
#define MACRO(v, w, x, y) \
 do {                     \
    v = (x) + (y);        \
    w = (y) + 2;          \
} while(0)
```

1.8 Enum names

1) Always use typedef,when defining a ENUM type variable.

2) Labels All Upper Case with '_' Word Separators .This is the standard rule for enum labels. No comma on the last element.

3) Finally use an "_t" when defining the Enum variable to indicate it is a typedef.

```
typedef enum pin_state_e { /*using typedef with a meaningful name */
    PIN_OFF,
    PIN_ON
    /* Always use capital letter to define labels */
} pin_state_t ;            /* suffix with "_t" to identify typedef */
```

1.9 Structure names

1) Use underbars ('_') to separate name components. Also make sure that the name given is meaningful and concise. One should be able to tell the purpose of the structure just be seeing its name. It is always recommended to use typedef when declaring a structure variable. The structure variable is suffixed with a "_t" to indicate that it is a typedefvariable.For example a structure having details of Employees can be named as "emp_info".

```
typedefstruct student{     /* Use typedef with meaningful name */
    structstudent *next;   /* List of active student */
    structmarks;           /* Comment for marks */
    int bar;
    unsigned intmark1:1    /* Bitfield; line up entries if desired */
             mark2:5,
             mark2;
} student_t;               /* "_t" represents a typedef variable */
student_t *head_ptr;       /* defining a structure variable */
```

2) When defining structure variables, use typedefvariable . And also add '_ptr to indicate the pointer variables.

```
typedef struct config{     /* Always use typedef for structures */
    ........
    ........
    ........
    ........
} cfg_t;          /* config structure is abbreviated as "cfg_t"
                   * indicating that its a typedef variable of structure
                   * config */
```

3) When declaring variables in structures, declare them organized by use in a manner to attempt to minimize memory wastagebecause of compiler alignment issues, then by size, and then by alphabetical order. E.g, don't use ``int a; char *b; int c; char *d''; use ``int a; int b; char *c; char *d''. Each variable gets its own type and line, although an exception can be made when declaring bitfields (to clarify that it's part of the one bitfield). Note that the use of bitfields in general is discouraged. Major structures should be declared at the top of the file in which they are used, or in separate header files, if they are used in multiple source files. Use of the structures should be by separate declarations and should be "extern" if they are declared in a header file.

```
cfg_t head;
cfg_t *head_ptr;     /* declaring the pointer structure variable */
```