PATNI COMPUTER SYSTEMS LTD

Coding Standards for C

Compiled by Smart Programming CoP

DOCUMENT HISTORY

Reference no:		
Security classification:	**Patni internal confidential**	
Issue date:	**28-Mar-11**	
Contributed by	**PES**	
Date	**Version**	**Change Description**
24-Mar-11	**1.0**	**Baseline**

TABLE OF CONTENTS

TABLE OF CONTENTS 3

1 CODING STANDARDS 4

1.1 NAMING CONVENTIONS 4

1.1.1 Variable Types 4

1.1.2 Variables 4

1.1.3 Structures 5

1.1.4 Enums 6

1.1.5 User-defined Data-types 6

1.1.6 Constants 7

1.1.7 Functions 7

1.1.8 Macros 7

1.1.9 Block 7

1.1.10 Files 7

1.1.11 Loops and Conditions 8

1.2 GENERAL PROGRAMMING RULES 8

1.2.1 General 8

1.2.2 History 8

1.2.3 Variable 8

1.2.4 Functions 9

1.2.5 Constants 10

1.2.6 Pointers 10

1.2.7 Pointer handling 10

1.2.8 Stack size 10

1.2.9 Operators 10

1.3.10 Pre processor directives 10

1.3.11 Avoid dynamic memory allocation 10

1.3.12 No compiler warnings 10

1.3.13 Add Comments 11

1.3 CODE OPTIMIZATION 11

2 EXCEPTIONS 11

1 CODING STANDARDS

This section specifies the C language coding standards and code optimization techniques to be followed for the project.

1.1 NAMING CONVENTIONS

1.1.1 Variable Types

Each project will have its own typedef defined against each basic 'C' data types. These definitions should be defined through a conveniently named header file (for example; "SysTypes.h"). This file should be located in a common accessible directory in the project and all source files shall include this header file.

Table 1: Variable Types naming

Data types	Corresponding typedef
unsigned char	BOOL
unsigned char	UINT8
signed char	SINT8
unsigned short	UINT16
signed short	SINT16
unsigned long	UINT32
signed long	SINT32
float	FLT
double	DBL

1.1.2 Variables

1. Use general variable syntax defined in **Table 1** for naming the variable.
2. Use prefixes defined in **Table 2** for specifying the scope of the variable.
3. Use prefixes defined in **Table 3** for specifying the data type of the variable.
   e.g. a local pointer will be named as "l_pSignal".
   e.g. a local unsigned char will be named as "l_ucName".
4. Avoid similar looking names. e.g. l_nSysTst and l_nSysStst.
5. Names for similar but distinct entities will be distinct. That is, it is better to have names like l_nLion and l_nTiger instead of l_nAnimal1 and l_nAnimal2.
6. Names will not have an underscore at their beginning or end.
7. Names should be related to the purpose of the identifiers.
8. Underscore '_' should be used to separate scope of the variable while significant parts.
9. Avoid names that differ only in presence or absence of underscore '_'.
10. Avoid names that differ only in case, e.g. l_nfoo, l_nFoo.
11. Avoid names conflicting with standard library names, variables and keywords.
12. Unrelated variables should be defined at different line.

Table 2: General Syntax

Scope Prefix	Separator	Data Type Prefix	Name
Define Scope of variables. See table 2 for details.	Underscore'_'	See table 4 below for details.	Variable name starting with Capital letter. If the Name consists of more than one word, the starting character of each word will be in Capital letters. Others will be strictly lowercase.

Table 3: Conventions for Scope Prefix

Scope/Qualifier	Prefix
Local Variable	l
Global/File Variable	g
Function Variable (Parameters/arguments)	f
Member variable(Structure member variable)	m
Const variable	c
Static variable	s
Register variable	r
Volatile variable	v

Table 4: Conventions for Data-Type prefix

Data Type	Prefix
Pointer	p
Array	a
Short	s
Integer	n
Long	l
Float	f
Double	d
Char	c
Bool	b
Unsigned	u
Structure and User defined data types	t
Enumeration	e

1.1.3 Structures

1. All Structure names should be in Uppercase with suffix _Type.

2. If the name consists of more than one word, they shall be separated by underscore '_'.
3. Member variable of the structure will have the prefix 'm_'.
4. For example
   Structure for coordinate can be defined as

```
typedef struct COORD_T
{
        int m_x;
        int m_y;
} COORD_Type;
```

1.1.4 Enums

1. Enum names should be in Uppercase with suffix _Type.
2. Enum elements names should be defined in Upper case.
3. If the name consists of more than one word, they shall be separated by underscore '_'.
4. The first element of enum should always be initialized.
5. For example
   Enum for different IO operations can be defined as

```
typedef enum IO_TYPE_E
{
        READ_TEXT = 0,
        WRITE_TEST,
        READ_BIN,
        WRITE_TEXT,
} IO_TYPE_Type;
```

1.1.5 User-defined Data-types

1. Names for user-defined data-types should be in Upper case with suffix _Type.
2. If the name consists of more than one word, they shall be separated by underscore '_'.
3. Structures and Enums should always be defined as typedefs
4. For Example:
   - Structure for coordinates can be defined as

```
typedef struct COORD_T
{
        int m_x;
        int m_y;
} COORD_Type;
```

   - Enum for different IO operations can be defined as

```
typedef enum IO_TYPE_E
{
        READ_TEXT = 0,
        WRITE_TEST,
        READ_BIN,
        WRITE_TEXT,
} IO_TYPE_Type;
```

1.1.6 Constants

1. All the symbolic constant names should be in capitals only. Underscore should be used between different word parts of the name.
   For e.g. const GLBL_ARR_BOUND 25
2. All symbolic constants should have 4 character prefix relevant to the module followed by underscore. For global constants prefix “GLBL” should be used.
   For e.g. const GLBL_ARR_BOUND 25

1.1.7 Functions

1. Use meaningful name for functions related to its functionality.

1.1.8 Macros

1. Macro name should be in all capitals with underscore between words in name.
   For e.g.
   #define MOD (a,b) (a) < (b) ? ((b) - (a)) : ((a) - (b))
2. Macro must have pair of parenthesis if it calls a function inside it, even if the macro does not have parameters.
   For e.g.
   #define CALLFUNC() CallOtherFunction()
3. A separate macros file shall be prepared with standard macros line ON, OFF, TRUE, FALSE etc. this header file shall be updated if macro looks common to the project.

1.1.9 Block

1. Block start and end will have same indentation.
   For e.g.
   ```
   if (condition)
   {
       Body
   }
   ```
2. The length of a single line of code should not exceed 80 characters. This rule has exceptions where there are tables; in this case readability has more importance than this rule.
3. Proper readability of code must be maintained. Make use of blank lines to improve readability where ever necessary.

1.1.10 Files

1. File name must be of the format: <base name>.<ext>.
2. Files belonging to a module should have a common prefix added to the base name, which would indicate the module to which the file belongs.
3. If a file contains only one function then the name of the file can be the same as the name of the function.
4. File name should never conflict with any system file names.
5. Spaces should not be used in the file name.
6. Function names should normally be formed from two parts: an action (verb) and an object (noun) of the action. Exceptions are query functions where the second

part is not a noun, but the name should form a “question”. Each word forming the function name is capitalized, except for the first which should be lowercase (this will be the module name). Examples of acceptable function names are playFile, setTime and driveIsActive.

7. If a file grows more than 1000 lines, always split the file into two or more files.

1.1.11 Loops and Conditions

1. All statements within for/if loops will be indented.
2. A break statement must always terminate the code following a case label.
3. For every ‘if’ and ‘else’ statement braces should be used.
4. A switch statement must always contain a default branch, which handles unexpected cases.
5. Never use ‘goto’ and ‘continue’ statement.
6. Each ‘if’ should have an else statement even if there is nothing to be done in else. Comment as /* do nothing */ in such cases. This does not add any code in the image as the linker generally removes such conditions.
7. There shall be no unreachable code.

1.2 General Programming Rules

1.2.1 General

1. Use of “Spaces” for indentation is mandatory. No tabs are allowed.
2. Strictly use 4 spaces for indentation.

1.2.2 History

Every modification to a file needs to be documented as modification history within the file. This is mandatory for *.c, *.cpp, *.h. A minimum of one comment shall be provided per change/modification.

The modification comment must be placed in the file modification comments header block. The modification comment must include

- Date
- Name
- Comment

This is illustrated in example below

```
/*
  16-Jul-2010    ShaileshM    Added General programming guidelines
*/
```

The personal names used in the comments should be easily recognizable.

1.2.3 Variable

1. Avoid the use of numeric values in code; use symbolic values instead.

2. Also ‘const’ and ‘enum’ should be used over ‘#define’ as both ‘const’ and ‘enum’ provide type checking.
3. Variables are to be declared with the smallest possible scope.
4. Every variable that is declared is to be given a value before it is used.
5. Always define a variable within say Init function and explicitly define the same; do not use declaration and definition at the same place.
6. Consider limitations such as allowable length and special characters of compiler/linker on local machines.
7. Never use floating point variable for equality or inequality.

1.2.4 Functions

1. The names of formal arguments to functions are to be specified and are to be the same both in the function declaration and in the function definition.
2. Do not write code, which depends on functions that use implicit type conversions.
3. Always specify the return type of a function explicitly.
4. Function should have only one return statement as far as possible.
5. All functions, which have a relevant success or failure condition, should return int value, which would be one of the members in global enumeration defined for all the collection of success and failure conditions.
6. Never call Function themselves.
7. Function header :
   Format

```
/* <function declaration>
 * Brief: <brief description of the function>

 * Params: <parameter-name> {parameter description}
 * Return: {description of the return value}
 */
```

Example

For Example the function int read(int f_nFd,char *f_pcBuf,size_t f_tCount) can be documented as

```
/* int read(int f_nFd,char *f_pcBuf,size_t f_tCount)
 * Brief: Read bytes from a file descriptor.

 * Params:  f_nFd The descriptor to read from.
 * f_pcBuf The buffer to read into.
 * f_tCount The number of bytes to read.
 * Return: returns 0 if fails and 1 for success.
```

1.2.5 Constants

1. Never convert a const to a non-const.
2. Never use octal constants and octal escape sequence.

1.2.6 Pointers

1. For pointer definitions attach "*" to variable names and not to the types. For e.g. Use char *l_pcAddress; instead of char* l_pcAddress;
2. When a pointer is deleted, set that pointer to NULL.
3. To check the validity of a pointer, compare it to a typecast NULL. Do not use the integer 0.
   Note that stdio.h must be included to define NULL.

1.2.7 Pointer handling

1. Verify if the memory is allocated before using pointer variables in the code.
2. Verify that every pointer is initialized before being used.
3. Never use Conversion between pointer to function and any type other than an integer type.
4. Never use cast conversion between a pointer type and an integer type.

1.2.8 Stack size

1. Check for stack size & stack pointer value while writing assembly routine. Make sure that stack is not getting overflow.

1.2.9 Operators

1. Never use **sizeof** operator on expressions.
2. Never use comma operator.

1.3.10 Pre processor directives

1. All #pragma directives should be documented and explained.

1.3.11 Avoid dynamic memory allocation

1. Dynamic memory allocation should be avoided, to prevent memory leaks and fragmentation unless it is guaranteed that the system will not go short of memory in any scenario. Never allocate or free memory in an ISR.

1.3.12 No compiler warnings

1. Code that compiles with warnings will not be accepted for integration. Also while developing code eliminate the warnings as-soon-as-possible, as the warnings tend to be the source of logic and run-time errors.
   NOTE: You should always compile with the compiler switch that treats warnings as errors.
   Use as many parentheses. Never let the compiler resolve the operator precedence.

1.3.13 Add Comments

1. Add comments whenever possible and wherever possible. Particularly, while patching the code, explain the implementation.

1.3 CODE OPTIMIZATION

The optimizations should be done on those parts of the program that are run the most, especially those methods which are called repeatedly by various inner loops that the program can have.

2 EXCEPTIONS

If some external packages are used in the project then coding standards used in that package should be followed while modifying the code. These packages can be device driver code provide by the device manufactures, open-source code, auto generated code using UML tool, etc.