Texas Instruments

Recommended C Style and Coding Guidelines

Coding Guidelines

Version v0.3

Topic of Contents

1. INTRODUCTION

This document describes a general standard for programming in C language and is based on MISRA-C 2004 coding guidelines. The rules and guidelines in this document standardize common code development practices to increase the code readability, productivity and maintainability.

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

Donald Knuth

1.1 Purpose

This coding standard is intended to help the developer to:

- Avoid the usage of undefined, unspecified or implementation-defined constructs of the C language
- Guard against common programming errors
- Limit program complexity
- Establish a consistent style
- Establish an objective basis of code review
- Establish a basis for static testing
- Enhancing software quality by avoidance of risky language constructs
- Easier portability to other compilers or microcontroller platforms

1.2 References

- MISRA-C 2004 guidelines
- Specification of C Implementation Rules V1.0.5 R4.0 Rev 1 (DocumentID 190: AUTOSAR_TR_CImplementationRules)

1.3 Notation

The use of work "shall" in the document requires that any project using this document must comply with the stated standard.

1.4 Implementation Guidelines

This document specifies serially numbered set of rules and guidelines for C coding.

The letter 'R' prefixed to the number identifies rules and the letter 'G' prefixed to the number identifies guidelines. There are some guidelines which refer to Code Layout; these illustrate how a .c/.h file needs to be structured.

2. MISRA-C GUIDELINES

Quick facts about MISRA-C

MISRA-C version	**MISRA-C 2004**
Rule Count	**121 (Mandatory) + 20 (Advisory) = 141**
	21 categories of rules. Rules are numbered a.b
MISRA-C website	http://www.misra-c.com/

MISRA requires that all deviations are reviewed and documented (the following steps are mandatory to be followed)

1. There may be instances where it is technically not possible to follow a rule. In this case the deviations shall be documented. The reason for allowing deviations for a particular rule shall be documented and approved.
2. Code reviews shall include a review of the MISRA violations.
3. Deviations shall be approved at least CP3B.

2.1 List of MISRA-C 2004 Rules to be adhered

Following rules of MISRA-C 2004 are mandatory and shall be followed for all implementations.

Rule#	Type	Category
2.1	Required	Language Extensions
2.2	Required	Language Extensions
2.3	Required	Language Extensions
2.4	Advisory	Language Extensions
4.1	Required	Character sets
4.2	Required	Character sets
5.2	Required	Identifiers
5.3	Required	Identifiers
5.4	Required	Identifiers
6.1	Required	Types
6.3	Advisory	Types
6.4	Required	Types
6.5	Required	Types
7.1	Required	Constants
8.1	Required	Declarations and definitions
8.2	Required	Declarations and definitions
8.3	Required	Declarations and definitions

Rule#	Type	Category
8.4	Required	Declarations and definitions
8.5	Required	Declarations and definitions
8.6	Required	Declarations and definitions
8.7	Required	Declarations and definitions
8.8	Required	Declarations and definitions
8.9	Required	Declarations and definitions
8.11	Required	Declarations and definitions
8.12	Required	Declarations and definitions
9.1	Required	Initialization
9.2	Required	Initialization
9.3	Required	Initialization
10.1	Required	Type conversion
10.3	Required	Type Conversion
10.4	Required	Type Conversion
10.6	Required	Type Conversion
11.1	Required	Pointer type Conversion
11.2	Required	Pointer type Conversion
12.1	Advisory	Expressions
12.2	Required	Expressions
12.3	Required	Expressions
12.4	Required	Expressions
12.5	Required	Expressions
12.7	Required	Expressions
12.8	Required	Expressions
12.9	Required	Expressions
12.10	Required	Expressions
12.13	Advisory	Expressions
13.1	Required	Control Statement Expression
13.2	Advisory	Control Statement Expression
13.3	Required	Control Statement Expression

Rule#	Type	Category
13.4	Required	Control Statement Expression
13.5	Required	Control Statement Expression
13.6	Required	Control Statement Expression
14.1	Required	Control Flow
14.2	Required	Control Flow
14.5	Required	Control Flow
14.8	Required	Control Flow
14.9	Required	Control Flow
14.10	Required	Control Flow
15.1	Required	Switch statement
15.2	Required	Switch statement
15.3	Required	Switch statement
15.4	Required	Switch statement
15.5	Required	Switch statement
16.1	Required	Functions
16.2	Required	Functions
16.3	Required	Functions
16.4	Required	Functions
16.5	Required	Functions
16.8	Required	Functions
16.9	Required	Functions
16.10	Required	Functions
17.2	Required	Pointers and Arrays
17.3	Required	Pointers and Arrays
17.5	Advisory	Pointers and Arrays
17.6	Advisory	Pointers and Arrays
18.1	Required	Structures and Unions
18.2	Required	Structures and Unions
18.4	Required	Structures and Unions
19.1	Advisory	Preprocessor directives
19.2	Advisory	Preprocessor directives
19.3	Required	Preprocessor directives
19.5	Required	Preprocessor directives

Rule#	Type	Category
19.6	Required	Preprocessor directives
19.8	Advisory	Preprocessor directives
19.9	Required	Preprocessor directives
19.10	Required	Preprocessor directives
19.12	Required	Preprocessor directives
19.13	Advisory	Preprocessor directives
19.14	Advisory	Preprocessor directives
19.15	Required	Standard Libraries
19.16	Required	Preprocessor directives
19.17	Required	Preprocessor directives
20.1	Required	Standard Libraries
20.4	Required	Standard Libraries
20.5	Required	Standard Libraries
20.6	Required	Standard Libraries
20.7	Required	Standard Libraries
20.8	Required	Standard Libraries
20.9	Required	Standard Libraries
20.10	Required	Standard Libraries
20.12	Required	Standard Libraries

Following rules of MISRA-C 2004 shall be "Blanket deviations" and are disabled in the tool. Remaining rules will be enabled in the tool.

Rule#	Type	Category
1.3	Required	Environment
1.5	Advisory	Environment
3.1	Required	Documentation
3.2	Required	Documentation
3.6	Required	Documentation
5.1	Required	Identifiers
5.6	Advisory	Identifiers
5.7	Advisory	Identifiers
12.11	Advisory	Expressions

Rule#	Type	Category
18.3	Required	Structures and Unions
19.4	Required	Preprocessor directives
19.11	Required	Preprocessor directives
20.3	Required	Standard Libraries

Following rules of MISRA-C 2004 are partially checked by the tool and hence the code cannot be completely checked for compliance to these rules.

Rule#	Type	Category
1.1	Required	Environment
1.2	Required	Environment
1.4	Required	Environment
3.3	Advisory	Documentation
3.4	Required	Documentation
3.5	Required	Documentation
12.12	Required	Expressions
13.7	Required	Control Statement Expression
16.6	Required	Functions
20.2	Required	Standard Libraries
21.1	Required	Run time failures

Following rules of MISRA-C 2004 shall be "Acceptable deviations" when each instance of the deviation is reviewed and signed off. Software developer will review the violation reported for the rules under this category and will decide to fix it or not on case by case basis. For the situations, if one doesn't fix he/she shall put a comment on top of the source code line

Rule#	Type	Category
5.5	Advisory	Identifiers
6.2	Required	Types
8.10	Required	Declarations and definitions
10.2	Required	Type conversion
10.5	Required	Type conversion
11.3	Advisory	Pointer type Conversion

Rule#	Type	Category
11.4	Advisory	Pointer type Conversion
11.5	Required	Pointer type Conversion
12.6	Advisory	Expressions
14.3	Required	Control Flow
14.4	Required	Control Flow
14.6	Required	Control Flow
14.7	Required	Control Flow
16.7	Advisory	Pointers and Arrays
17.1	Required	Pointer and Arrays
17.4	Required	Pointer and Arrays
19.7	Advisory	Standard Libraries
20.11	Required	Standard Libraries

3. SOURCE ORGANIZATION

G1 Software development should follow the directory structure as specified by the organization. Refer to each organization's directory structure before starting any new development activity. Careful thought must be given to the directory structure so that it is easy to navigate through the folders and identify purpose of each folder.

4. STYLE GUIDELINES

4.1 General Guidelines

R1. A line of code shall not exceed 80 characters in length. Indent, spaces and line numbers are counted in the 80 character total.

R2. Any statement that exceeds 80 characters in length shall be broken into multiple lines. Break lines after a comma delimiter or after an operator.

When an expression or a statement does not fit into a single line, consider breaking into multiple lines using the following thumb rules:

- Break after a comma delimiter
- Break after an operator

```
{
    if (a != 10 &&
        b != 11 &&
        c != 12)
}
```

R3. Tab characters shall not be used for indentation.

R3.1. The indentation level shall begin at column one.

R3.2. Each indent level shall be four (4) spaces.

R3.3. Entering a block (all statements between '{'and '}' brackets) increases the indentation level by one.

R3.4. An opening brace shall be on the same level as the preceding line.

R4. Logical units within a block shall be separated by one blank line. No random blank lines must exist in the code. No blank lines shall appear at the end of the file.

4.2 Spacing

Spacing around operators and delimiters shall be consistent to improve readability.

R5. The code shall have consistent spacing for operators.

R5.1. There shall be one white space on either side of the binary operators.

Binary Operators

* / %	Multiplicative operators
+ -	Additive operators
<< >>	Shift operators
< > <= >= == !=	Relational operators
& | ^	Bitwise operators
&& ||	Logical operators
= += -= *= /= %= <<= >>= &= ^= |=	Assignment operators

R5.2. There shall be no white space on either side of primary operators "->", ".", ",","[]","()" and their operands.

R5.3. There shall be no white space between a unary operator and operand.

Unary Operators

~ !	:	Negation and complement operators
* &	:	Indirection and address-of operators
sizeof	:	Size operator
+	:	Unary plus operator
++ - -	:	Unary increment and decrement operators

R5.4. There shall be one white space after a comma and semicolon unless it is the last character in the line. No white space shall exist before a comma. No white space shall follow a semicolon.

```
variable = variable2 + 5;   /* correct */
variable =variable2+5;      /* wrong */

++variableName;             /* correct */
++ variableName;            /* wrong */

myFunction(oper1, oper2);  /* correct –space after comma */
myFunction(oper1 , oper2); /* wrong – space before comma */

a->foo();                   /* correct */
b.bar();                    /* correct */
b . bar();                   /* wrong */
```

4.3 File Preamble

An essential piece of documentation is the interface. The interface must be documented in a header file. Any developer who needs to call a function shall not look anywhere else but in the function's header file.

R6. Every source file shall have a file header. The file header shall describe purpose of the file.

G2 Every source file should have revision history that includes author, date and purpose of the change at the end of the file. Change history shall be kept up to date.

```
*-------------------------------------------------------------
* Revision History
*-------------------------------------------------------------
* Version   Date     Author   Change ID        Description
*-------------------------------------------------------------
* 00.01.00  6Apr2012  Sunil    0000000000000    Initial Version
* 00.01.01  11May2012 Sunil    SDOCM00092133    Removing All
Wrong #Includes in this file(Memmap)
*************************************************************/
```

R7. All files shall include the standard comment header with Texas Instruments copyright information.

```
/*************************************************************
 *  (C) Copyright 2013, Texas Instruments, Inc.
 *  ---------------------------------------------------------
 *  \verbatim
 *
 * TEXAS INSTRUMENTS INCORPORATED PROPRIETARY INFORMATION
 *
 * Property of Texas Instruments, Unauthorized
 * reproduction and/or distribution is strictly
 * prohibited.  This product is under  copyright  law and
 *  trade  secret law as an  unpublished work
 * (C) Copyright Texas Instruments.  All rights reserved.
 *  \endverbatim
 *  ---------------------------------------------------------
 *************************************************************/
```

4.4 Precedence

R8. Brackets shall always be used in complex expressions even if the C precedence rules do not necessarily demand this for operators. This also applies to expressions evaluated by the preprocessor.

R9. Use of '++' and '- -': The use of '++' and '- -'shall be limited to simple cases. They shall not be used in statements where other operators occur. The prefix use is always forbidden.
E.g.: x -= y++; /* violation */

4.5 Pre-processor

R10. Preprocessor directives shall begin in column 1.

```
#ifdef DEBUG
     int TestFlag;
#ifdef TEST
     #define TESTFLAG 1
#else
     #define TESTFLAG 0
#endif
#endif
```

4.6 Comments

Document the interface (what is the expected input, what is the expected output, and what it is actually, among others) as it cannot necessarily be derived from the code. The implementation may be more general. If the implementation uses specific tricks, makes unusual assumptions, or uses a pretty complex algorithm a corresponding note is, however, indicated. Avoid restating implementation in comments, as they tend to get out of sync.

Well-written source code with appropriately chosen variable, type and function names requires relatively little commenting. The standardization of source code makes it possible to automatically generate man-pages from source code. This may be used to keep source code and documentation together.

G3 Avoid unnecessary comments. If something can be said in code as well as comments, say it in code. If something needs to be said and cannot be said in code, say it in comments. A precise declaration is the best way to say something in code. The compiler checks usage of each variable against the attributes established in its declaration.

R11. Sections of code shall not be commented out. If a piece of code is not needed it shall be removed. Pieces of code shall not be kept in files for the purpose of history.

R12. A strategic comment describes what a section of code is intended to do and is placed before the code. The text of the comment shall be indented by 2 spaces.

```
/*  This is line 1 of the strategic comment
 *  This is continuation of the comments
 */
Statements
```

G4 A tactical comment describes what a single line of code is intended to do and is placed, if possible at the end of the line. The text for tactical comment shall be indented by one space.

```
int something;
something = complex + calculation; /* This is tactical comment
*/

/* a tactical comment.              */
return index;
```

R13. Comment adequately for all the macros, typedefs, structures using doxygen style.

G5 Comment density should not be less than 20%

4.7 File inclusions

R14. Protection against multiple inclusions. Each header file shall protect itself against multiple inclusions.

E.g.: #ifndef FILENAME_H
#define FILENAME_H
.....
#endif /* FILENAME_H */

R15. Inclusion of own header file. Each module shall include its own header file.

R16. Body inclusion: A ".c" file shall not be included in another file; it shall be compiled and provided as an object module.

4.8 Identifiers

R17. Names with leading and/or trailing underscores shall not be used. The exception to this rule is standard header files in operating systems that have leading underscores to distinguish between public and private headers.

R18. Type definitions must begin with their module name in Camel Case starting with a capital letter, followed by an underscore (i.e. '_').

e.g. ModuleName_FruitTypes

R19. All functions, data types, macros, variables and constants shall be uniquely and fully declared once (and only once) and subsequently implemented, initialized and used fully consistent with their declared meaning.

R19.1. Functions shall have a complete prototype that explicitly specifies a return type and a definite number of parameters that are fully and uniquely defined including name and data type and return type.

R19.2. Functions that are exposed as part of the API shall be declared in an external in the associated header file. Functions that are not part of the API shall be declared as static.

R19.3. All typedefs and macros shall be defined only once.

R20. Identifier names that differ only in case shall not be used.

R21. Identifier names shall not contain the underscore character '_' twice in succession.

R22. Use NULL/NULL_PTR instead of 0 for pointer initializations.

G6 The name of an identifier should clearly describe its purpose.

G7 Abstract names that are likely to be reused in other parts of the system should be avoided.

G8 Avoid abbreviations in identifier names except where the abbreviation is an industry or project standard.

G9 Include units in identifier names.

If a variable represents time, weight or some other unit, then include the unit in the name. For example:

```
uint32 timeoutMsec;
uint32 myWeightKgs;
```

4.9 File Naming

A system may be composed of files from different sources. Special care shall be given to the naming of the header files because of conflicts with other modules. A standard file naming convention is required to avoid name collusions in a system.

R23. Header and source files that are distributed, as part of the product shall be uniquely named.

R24. The file name must be Camel Case, where the first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized.

R25. The interface file shall have the extension ".h" and the body file shall have the extension ".c".

R26. Standard file type shall use consistent file name extensions.

The following extensions shall be used for standard files:

C source Code → filename.c

C header file → filename.h

The project is free to define extensions for other non-standard file types e.g., data file may have the extension .dat .

R27. Filenames shall contain only alphanumeric characters. White spaces, plus, minus and other meta-characters shall be avoided.

R28. File names that differ only in case shall not be used.

4.10 Functions

R29. Every function shall have a function header. The function header shall capture its purpose and assumptions, full description of each parameter, the return type, all possible return values, required pre-conditions, and post-conditions and reference to related functions.

R30. No function definition within header file. Functions (other than macros) shall not be defined within a header file; however definitions of inline functions in the header file are permitted.

R31. Declaration of function parameters

R31.1. Declarations of functions shall always be stated with detailed parameter list, i.e. the type and a practical designation of the relevant parameters.

R32. Declaration and definition of local functions shall have the storage-class specifier "static". Local function means function with internal linkage (only visible inside the module).

E.g.: static void MyLocalFunction (void);

R33. Declaration of global functions: A declaration with storage –class specifier "extern" shall exist for each global function in the header file of the module.

E.g.:

```
extern Std_ReturnType Eep_Erase
(
Eep_AddressType EepromAddress,
Eep_LengthType Length
);
```

G10 Use doxygen style of coding for the function headers.

G11 Function name should identify the action performed or the information provided by the function.

G12 All functions must begin with their module name in Camel Case starting with a capital letter, followed by an underscore (i.e. '_'). Following the underscore, functions names must be Camel Case starting with a lowercase letter and start with a verb describing the action of the function

e.g. Uart_putChar(); Pwm_setPeriod(); Timer_getValue();

4.11 Variables

G13 Variables should be named with nouns or noun phrases. Variables should have close correlation with the names used in the corresponding standard.

R34. All variables are Camel Case starting with a lowercase letter. Global variables must begin with their module name in Camel Case starting with a capital letter, followed by an underscore (i.e. '_').

e.g. uint16_t loopCounter; uint16_t ModuleName_globalVariable

R34.1. Local variable names must be Camel Case, where the first letter is lowercase and the first letter of each subsequent concatenated word is capitalized.

e.g. : fruitType, vegetableType

R34.2. Global variables must be preceded by the module name, in Camel Case starting with a capital letter, followed by an underscore (e.g. ModuleName_).

e.g. : FruitBasket_fruitType, VegBasket_vegetableType

R35. Global constants and Macros shall use upper case. Digits and underscores are allowed but not at the start.
e.g. #define NR_OF_ELEMENTS 10

R36. External declarations of global variables shall be done in header files and never in .c files

R37. Variables shall not be defined within a header file. They shall be defined within the module's C file.

4.12 Other Names

R38. Every data structure definition shall be preceded with a data definition header that describes the purpose of the structure, where it is used and why it is used.

G14 Every data structure definition should have the suggested usage in the data definition header.

R39. Types (structure types, etc.) shall be given meaningful names and begin with an uppercase letter in order to distinguish them from variables.

R40. Macros, enumeration constants and global constant and global typedef names shall be in all uppercase with individual words separated by underscores.

```
#define DATA_VALID   0x20
```

R41. Each self-defined type has to have an explicit type declaration even if there is only one variable of this type.

E.g.:

```
typedef struct
{
uint16 Position;
uint8 Direction;
} MotorType;
static MotorType MotorData;
```

R42. Each enumeration shall be preceded by an enumeration header that describe the purpose of the enumeration and clearly define its members.

4.13 Braces

R43. Use a single bracing style that follows these rules:

R43.1. The opening brace for a function definition shall be placed on next line aligned with the first character of the function name. This brace shall be alone in its line.

R43.2. Opening brace for a block shall be placed at next line of the construct (e.g., if, while, for etc).

R43.3. The closing brace shall appear on a line by itself at the same level on indent as the initiating keyword.

R43.4. In constructs such as if() { } else or do { } while(), place the else or while on the line following the closing brace of the preceding element.

R43.5. For looping and conditional constructs, the opening brace shall be on the next line as the construct.

R43.6. The starting and ending braces for a function body shall be on a line by itself and aligned with the starting character of the function.

R44. Braces shall follow a loop or a conditional construct even if there is only one statement.

The following examples show how certain types of statements should be formatted.

- For statement

```
for (expr; expr; expr)
{

}
```

- If statement

```
if (expr)
{
    'statement(s)'
}
```

- If-else-if statement.

```
if (expr)
{
    'statement(s)'
}
else if (expr)
{
    'statement(s)'
}
else
{
    'statement(s)'
}
```

- Switch statement

```
switch (expr)
{
case CONST:            /* note the indentation of
case */
          break;
case CONST:
          break;
default:              /* note the indentation of
case */
          break;
}
```

- While and do statements

```
while (expr)
{
     'statement(s)'
}

do
{
     'statement(s)'
}
while (expr);
```

- Functions

```
int
myFunction(int param)
{                                        /* correct */
}

int
myFunction(int param) {                  /* wrong */
}
```

5. FILE STRUCTURE

R45. All software in a file must reside in one of the following sections. The sections must appear in the following order within a file - header, includes, defines, typedefs, globals, function prototypes.

6. EMBEDDED DOCUMENTATION

R46. All variables, defines, type definitions and function prototypes must have Doxygen comments. PDF and HTML document generation is required. Documentation will be generated with an up to date version of Doxygen and Latex.

7. GENERAL GUIDELINES

G15 Do not create big "C" files. Split files into smaller logical files.

G16 Limit the size of each function. Functions should fit into single page whenever possible. Number of executable lines in a function should not exceed 50.

G17 Avoid use of multiple returns from function. Multiple returns will reduce the readability of the code. Functions should have only one exit point.

G18 While developing "C" code, isolate portable and non-portable sections of code and place them in separate files to enable quick porting to different architectures.

G19 Do not introduce too many variables. Clearly understand and create variables only if needed.

G20 All variables should be initialized before use.

G21 Multiple assignments should not be done.

E.g.: x = y = z; /* violation */

8. TEMPLATES

8.1 File Header

```
/**************************************************************************
 *  (C) Copyright 2013, Texas Instruments, Inc.
 *  ------------------------------------------------------------------
 *  \verbatim
 *
 *          TEXAS INSTRUMENTS INCORPORATED PROPRIETARY INFORMATION
 *
 * Property of Texas Instruments, Unauthorized reproduction and/or
 * distribution is strictly prohibited.  This product  is  protected
 * under  copyright  law and  trade  secret law as an  unpublished
 * work
 * (C) Copyright Texas Instruments.  All rights reserved.
 *  \endverbatim
 *  ------------------------------------------------------------------
 *
 *  ------------------------------------------------------------------
 *  FILE DESCRIPTION
 *  ------------------------------------------------------------------
 *         File:  can.c
 *    Component:  Tms570_AutoSarCanDriver
 *       Module:  DrvCan
 *    Generator:  -
 *
 *  Description:  The CAN driver is responsible for controlling the CAN
module. Provides API's for transmission and
 *                reception of the data over CAN module
 *
 *------------------------------------------------------------------
 *------------------------------------------------------------------
 * Author:  Sunil
 *------------------------------------------------------------------
 * Revision History
 *------------------------------------------------------------------
 * Version   Date      Author   Change ID        Description
 *------------------------------------------------------------------
 * 00.01.00  6Apr2012  Sunil    0000000000000    Initial Version
```

```
/*************************************************************
*  INCLUDE FILES
*************************************************************/
#include "csltimer.h"
/*************************************************************
 *  Local Functions  prototypes
*************************************************************/
static uint8  LoopCounter(uint8 controller,uint8 source);
*************************************************************
 Defines / data types / structs / unions
*************************************************************/
/*CTL REg Patterns */
#define LOOPBACK_REG_BIT        (0x04U)
```

8.2 Function Header (in C file)

```
/*****************************************************************************
 * Can_Init
*****************************************************************************/
/*! \brief     This function initializes the CAN controller
 *
 *  All the required initialization for the CAN driver and hardware is
 *  performed in this function.
 *  \param[in]  Can_ConfigPtr  Pointer to post-build configuration
 *  data.
 *  \context    App
*****************************************************************************/
```

8.3 Data Structure Header

```
/** \brief This type is used to provide ID, DLC and SDU from CAN
 *    interface to CAN driver.
 */
typedef struct
{
  Can_IdType id;               /**< CAN ID */
  PduIdType swPduHandle;       /**< PDU ID for Tx confirmation */
  uint8 length;                /**< Data length code (DLC) */
} Can_PduType;
```

8.4 Enum Header

```
/** \brief State transitions that are used by the function
*  Can_SetControllerMode.
*/
typedef enum
{
  CAN_T_START,  /**< CAN controller to request state STARTED. */
  CAN_T_STOP,   /**< CAN controller to request state STOPPED. */
  CAN_T_SLEEP,  /**< CAN controller to request state SLEEP.   */
  CAN_T_WAKEUP  /**< CAN controller to request state STOPPED. */
} Can_StateTransitionType;
```

8.5 Macro Header

```
#define KMAX_CONFIG_SET      (1U) /* Maximum number of Config sets */
```

9. SUMMARY OF RULES

R1. A line of code shall not exceed 80 characters in length. Indent, spaces and line numbers are counted in the 80 character total.

R2. Any statement that exceeds 80 characters in length shall be broken into multiple lines. Break lines after a comma delimiter or after an operator.

R3. Tab characters shall not be used for indentation.

- R3.1. The indentation level shall begin at column zero.
- R3.2. Each indent level shall be four (4) spaces.
- R3.3. Entering a block (all statements between '{'and '}' brackets) increases the indentation level by one.
- R3.4. An opening brace shall be on the same level as the preceding line.

R4. Logical units within a block shall be separated by one blank line. No random blank lines must exist in the code. No blank lines shall appear at the end of the file.

R5. The code shall have consistent spacing for operators.

- R5.1. There shall be one white space on either side of the binary operators.
- R5.2. There shall be no white space on either side of primary operators "->", ".", ","[]","()" and their operands.
- R5.3. There shall be no white space between a unary operator and operand.
- R5.4. There shall be one white space after a comma and semicolon unless it is the last character in the line. No white space shall exist before a comma. No white space shall follow a semicolon

R6. Every source file shall have a file header. The file header shall describe purpose of the file.

R7. All files must include the standard comment header with Texas Instruments copyright information.

R8. Brackets shall always be used in complex expressions even if the C precedence rules do not necessarily demand this for operators. This also applies to expressions evaluated by the preprocessor.

R9. Use of '++' and '- -': The use of '++' and '- -'shall be limited to simple cases. They shall not be used in statements where other operators occur. The prefix use is always forbidden.
E.g.: x -= y++; /* violation */

R10. Preprocessor directives must always begin in column 1.

R11. Sections of code shall not be commented out. If a piece of code is not needed it shall be removed. Pieces of code shall not be kept in files for the purpose of history.

R12. A strategic comment describes what a section of code is intended to do and is placed before the code. The text of the comment shall be indented by 2 spaces.

R13. Comment adequately for all the macros, typedefs, structures using doxygen style.

R14. Protection against multiple inclusions. Each header file shall protect itself against multiple inclusions.

R15. Inclusion of own header file. Each module shall include its own header file.

R16. Body inclusion: A ".c" file shall not be included in another file; it shall be compiled and provided as an object module.

R17. Type definitions must begin with their module name in Camel Case starting with a capital letter, followed by an underscore (i.e. '_').

e.g. ModuleName_FruitTypes

R18. Names with leading and/or trailing underscores shall not be used. The exception to this rule is standard header files in operating systems that have leading underscores to distinguish between public and private headers.

R19. All functions, data types, macros, variables and constants shall be uniquely and fully declared once (and only once) and subsequently implemented, initialized and used fully consistent with their declared meaning.

R19.1. Functions shall have a complete prototype that explicitly specifies a return type and a definite number of parameters that are fully and uniquely defined including name and data type and return type.

R19.2. Functions that are exposed as part of the API shall be declared in an external header file. Functions that are not part of the API shall be declared as static.

R19.3. All typedefs and macros shall be defined only once.

R20. Identifier names that differ only in case shall not be used.

R21. Identifier names shall not contain the underscore character '_' twice in succession.

R22. Use NULL instead of 0 for pointer initializations.

R23. Header and source files that are distributed, as part of the product shall be uniquely named.

R24. The file name must be Camel Case, where the first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized.

R25. The interface file shall have the extension ".h" and the body file shall have the extension ".c".

R26. Standard file type shall use consistent file name extensions.

R27. Filenames shall contain only alphanumeric characters. Underscores, white spaces, plus, minus and other meta-characters shall be avoided.

R28. File names that differ only in case shall not be used.

R29. Every function shall have a function header. The function header shall capture its purpose and assumptions, full description of each parameter,

the return type, all possible return values, required pre-conditions, and post-conditions and reference to related functions.

R30. No function definition within header file. Functions (other than macros) shall not be defined within a header file, however definitions of inline functions in the header file are permitted.

R31. Declaration of function parameters.

R31.1. Declarations of functions shall always be stated with detailed parameter list, i.e. the type and a practical designation of the relevant parameters.

R32. Declaration and definition of local functions shall have the storage-class specifier "static". Local function means function with internal linkage (only visible inside the module).

E.g.: static void MyLocalFunction(void);

R33. Declaration of global functions : A declaration with storage –class specifier "extern" shall exist for each global function in the header file of the module.

E.g.:

```
extern Std_ReturnType Eep_Erase
(
Eep_AddressType EepromAddress,
Eep_LengthType Length
);
```

R34. All variables are Camel Case starting with a lowercase letter. Global variables must begin with their module name in Camel Case starting with a capital letter, followed by an underscore (i.e. '_').

e.g. uint16_t loopCounter; uint16_t ModuleName_globalVariable

R34.1. Local variable names must be Camel Case, where the first letter is lowercase and the first letter of each subsequent concatenated word is capitalized.

e.g. : fruitType, vegetableType

R34.2. Global variables must be preceded by the module name, in Camel Case starting with a capital letter, followed by an underscore (e.g. ModuleName_).

e.g. : FruitBasket_fruitType, VegBasket_vegetableType

R35. Global constants and Macros shall use upper case. Digits and underscores are allowed but not at the start.
e.g. #define NR_OF_ELEMENTS 10

R36. External declarations of global variables shall be done in header files and never in .c files

R37. Variables shall not be defined within a header file. They shall be defined within the module's C file.

R38. Every data structure definition must be preceded with a data definition header that describes the purpose of the structure, where it is used and why it is used.

R39. Types (structure types, etc.) shall be given meaningful names and begin with an uppercase letter in order to distinguish them from variables.

R40. Macros, enumeration constants and global constant and global typedef names shall be in all uppercase with individual words separated by underscores.

R41. Each self-defined type has to have an explicit type declaration even if there is only one variable of this type.

```
E.g.:
typedef struct
{
uint16 Position;
uint8 Direction;
} MotorType;
static MotorType MotorData;
```

R42. Each enumeration shall be preceded by an enumeration header that describe the purpose of the enumeration and clearly define its members.

R43. Use a single bracing style that follows these rules:

R43.1. The opening brace for a function definition shall be placed on next line aligned with the first character of the function name. This brace shall be alone in its line.

R43.2. Opening brace for a block shall be placed at next line of the construct (e.g., if, while, for etc).

R43.3. The closing brace shall appear on a line by itself at the same level on indent as the initiating keyword.

R43.4. In constructs such as if() { } else or do { } while(), place the else or while on the line following the closing brace of the preceding element.

R43.5. For looping and conditional constructs, the opening brace shall be on the next line as the construct.

R43.6. The starting and ending braces for a function body shall be on a line by itself and aligned with the starting character of the function.

R44. Braces shall follow a loop or a conditional construct even if there is only one statement.

R45. All software in a file must reside in one of the following sections. The sections must appear in the following order within a file - header, includes, defines, typedefs, globals, function prototypes.

R46. All variables, defines, type definitions and function prototypes must have Doxygen comments. PDF and HTML document generation is required. Documentation will be generated with an up to date version of Doxygen and Latex.

10. SUMMARY OF GUIDELINES

G1	Software development should follow the directory structure as specified by the organization. Refer to each organization's directory structure before starting any new development activity. Careful thought must be given to the directory structure so that it is easy to navigate through the folders and identify purpose of each folder.
G2	Every source file should have revision history that includes author, date and purpose of the change at the end of the file. Change history shall be kept up to date.
G3	Avoid unnecessary comments. If something can be said in code as well as comments, say it in code. If something needs to be said and cannot be said in code, say it in comments. A precise declaration is the best way to say something in code. The compiler checks usage of each variable against the attributes established in its declaration.
G4	A tactical comment describes what a single line of code is intended to do and is placed, if possible at the end of the line. The text for tactical comment shall be indented by one space.
G5	Comment density should not be less than 20%
G6	The name of an identifier should clearly describe its purpose.
G7	Abstract names that are likely to be reused in other parts of the system should be avoided.
G8	Avoid abbreviations in identifier names except where the abbreviation is an industry or project standard.
G9	Include units in identifier names.
G10	Use doxygen style of coding for the function headers.

G11 Function name should identify the action performed or the information provided by the function.

G12 All functions must begin with their module name in Camel Case starting with a capital letter, followed by an underscore (i.e. '_'). Following the underscore, functions names must be Camel Case starting with a lowercase letter and start with a verb describing the action of the function

e.g. Uart_putChar(); Pwm_setPeriod(); Timer_getValue();

G13 Variables should be named with nouns or noun phrases. Variables should have close correlation with the names used in the corresponding standard

G14 Every data structure definition should have the suggested usage in the data definition header.

G15 Do not create big "C" files. Split files into smaller logical files.

G16 Limit the size of each function. Functions should fit into single page whenever possible. Number of executable lines in a function should not exceed 50.

G17 Avoid use of multiple returns from function. Multiple returns will reduce the readability of the code. Functions should have only one exit point.

G18 While developing "C" code, isolate portable and non-portable sections of code and place them in separate files to enable quick porting to different architectures.

G19 Do not introduce too many variables. Clearly understand and create variables only if needed.

G20 All variables should be initialized before use.

G21 Multiple assignments should not be done.

E.g.: x = y = z; /* violation */

VERSION HISTORY

S No	Author	Version	Date	Description
1	Siddharth Deshpande	0.1	2-July-13	Initial Draft
2	Siddharth Deshpande	0.2	19-July-13	Moved version history to end of document, corrected indentation for rules and guidelines, added rules from HIS metrics.
3	Siddharth Deshpande	0.3	31-July-13	Updated after review