C Language Programming Conventions

Document No.320SM0009

Approval	Review	Preparation
Name: Engineering Div. 2 Yonemura	Name: Engineering Div. 2 Takei	Name: Engineering Div. 2 Saiki
Date 2016/12/22	Date 2016/12/22	Date 2016/12/22

Revision history

No.	Date	Version	Revised content	Remarks
Ex.	20XX/XX/XX	RevX. X	Create New	
1	2005/4/27	Rev1. 0	Create New	
2	2016/12/22	Rev1. 1	Correction with reference to Misra-C material	
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				
21				
22				
23				
24				
25				
26				
27				
28				
29				

Table of contents

Defining everything within a program would take away from software its most significant characteristic, freedom. Therefore this document scope will be strictly limited to the structural conventions in programming. Structural conventions refer to program coding conventions, variable and constant naming conventions, and file structure conventions.

1. Coding structure conventions

1-1. Coding structure

The basic C language program file structure consists of the parts shown below:

```
1) File specification header
/*********************************************/
/*          Definition of specification      */
/*********************************************/
2) #Include definition
#include "include.h"  /* include General */
3) Internal function specification header
/*********************************************/
/*          Definition of specification      */
/*********************************************/
main()
{
4) Local variable declaration
U_INT lui_flag;    /* comment */
5) Program statement
Execution statement
}
3) Internal function specification header
/*********************************************/
/*          Definition of specification      */
/*********************************************/
6) Argument variable declaration
submenu(U_CHR  puc_flag,S_INT psi_number)
{
4) Local variable declaration part
U_INT lui_flag;    /* comment */
5) Execution executive
Execution statement
}
```

1-2. Structure overview

1) File specification header

Details the program processing specification

Refer to *Header format conventions* for more information

2) #Include definition

Defines the program's #include files

Usually defines the #include.h files incorporating all desired header definitions

3) Internal function specification header

Specifies the internal function type and parameters

Refer to *Header format conventions* for more information

4) Local variable declaration

Defines the desired function variables

Refer to *Local variable naming conventions* for more information

5) Program statements

Contains the execution statements

Use functional modules as far as possible

6) Argument variable declaration

Defines the argument type if present in the function

Refer to *Argument variable naming conventions* for details

2. Header format conventions

There are two formatting conventions for the header, examples of which are shown below.

2-1. File specification header

Unify the file header format as follows

```
/*********************************************************************/
/*                                                                   */
/*    1) Product name              :  GX-94                          */
/*    2) System                    :  Calculation module             */
/*    3) Development language      :  C language                     */
/*    4) Creator · Date            :  s.kizaki            1995-01-02 */
/*    5) Updater · Date            :  s.kizaki   :        1995-01-30 */
/*                                      (Function name changed)      */
/*                                                                   */
/*    6) Overview                    :Perform density calculation and */
/*                                       temperature calculation.    */
/*                                                                   */
/*********************************************************************/
```

2-2. Internal function header

```
/*-------------------------------------------------------------*/
/*      Overview        : Self-diagnosis                       */
/*      Function name   : trbl_chk                             */
/*      Creator · Date  : s-kizaki 1995-01-02                  */
/*      Updater · Date  : s-kizaki 1995-01-30                  */
/*      Argument        : None                                 */
/*      Return value    : None                                 */
/*      Restrictions    :                                      */
/*      Remarks         :                                      */
/*      Unit number     : req1-1-1-1                           */
/*-------------------------------------------------------------*/
```

3. Commenting conventions

a . Comment statements are added from position 40 to position70.

When the program statement extends beyond position 40, add the comment as shown below:

```
if((abc == def) && (ghi == ikl) && (lmn))
                                    /* conditions......... */
{
.........
.........
}
```

b . Comments for debugging are added from position 71 (To make it stand out)

```
if((abc == def) && (ghi == ikl) && (lmn))
                                    /* conditions......... */ /*dump*/
```

4. Data type definition conventions

4-1. Data types

The data type definitions shown below are to be used when developing in C. Only the definitions in the right-most column may be used for defining global and local variables in the program source code.

Type	Attribute	Definition
Char	signed unsigned	S_CHR U_CHR
Int	signed unsigned	S_INT U_INT
Long	signed unsigned	S_LNG U_LNG

4-2. Data type definitions

Always add these definitions to the #include.h file.

```
#define S_CHR signed      char
#define U_CHR unsigned    char
#define S_INT signed       int
#define U_INT unsigned     int
#define S_LNG signed       long
#define U_LNG unsigned     long
```

5. Global variable naming conventions

Define global variables collectively (in the *ram.c* file).

Global variable names should be meaningful (Do not use your name, etc).

The variable names should have the prefix specified below to identify the global variable type.

An underscore ("_") should be inserted between the prefix and variable name.

Always add comments.

Do not define multiple variables of the same type on the same line by separating them with commas.

2-character variable prefixes:

Type	Prefix	Example	Remarks
S_CHR	sc	sc_data1	-
U_CHR	uc	uc_data2	-
S_INT	si	si_data3	-
U_INT	ui	ui_data4	-
S_LNG	sl	sl_data5	-
U_LNG	ul	ul_data6	-
Structure	st	st_file. uc_flag	Add a type to a member

3-character variable prefixes:

For a pointer type variable (an address reference), append a "p" to the prefix for a 3-character prefix for pointers.

Examples:

Define a global variable to store data.

(signed char) Define as a type variable: sc_data

(unsigned char) Define as a type variable: uc_data

(unsigned char) Defined as pointer variable to type: ucp_data

6. Local variable naming convention

Local variable names (for other than simple loop counters) should be meaningful (Do not use your name, etc).

The variable name should have the prefix specified below to identify the local variable type.

An underscore ("_") should be inserted between the prefix and variable name.

3-character variable prefixes:

Type	Additional character	Example	Remarks
S_CHR	lsc	lsc_data1	-
U_CHR	luc	luc_data2	-
S_INT	lsi	lsi_data3	-
U_INT	lui	lui_data4	-
S_LNG	lsl	lsl_data5	-
U_LNG	lul	lul_data6	-
Structure	st	st_file. luc_flag	Add a type to a member

4-character variable prefixes:

For a pointer type variable (an address reference), append a "p" to the prefix for a 4-character prefix for pointers.

Examples:

Define a local variable that stores data.

(signed char) Define as a type variable: lsc_data

(unsigned char) Define as a type variable: luc_data

(unsigned char) Defined as pointer variable to type: lucp_data

7. Argument naming conventions

Argument variable names should be meaningful (Do not use your name, etc).

The variable name should have the prefix specified below to identify the argument variable type

An underscore (“_”) should be inserted between the prefix and variable name.

3-character variable prefixes:

Type	Additional character	Example	Remarks
S_CHR	psc	psc_data1	-
U_CHR	puc	puc_data2	-
S_INT	psi	psi_data3	-
U_INT	pui	pui_data4	-
S_LNG	psl	psl_data5	-
U_LNG	pul	pul_data6	-
Structure	st	st_file. puc_flag	Add a type to a member

4-character variable prefixes:

For a pointer type variable (an address reference), append a “p” to the prefix for a 4-character prefix for pointers.

Examples:

Define a local variable that stores data.

(signed char) Define as a type variable: psc_data

(unsigned char) Define as a type variable: puc_data

(unsigned char) Defined as pointer variable to type: pucp_data

8. File types and purpose

Below are typical examples of the conventions.

1) #Include file

Purpose: Collectively includes the header files containing the following definitions and declarations: MPU register and I/O related definitions, global variable definitions, function prototype declarations, constant definitions

Extension: h (header)

Name: include.h (fixed)

2) MPU register and I/O related definition file

Purpose: A definition file, mainly defines the registers and I/O address definitions for each MPU.

Extension: h (header)

Name: [MPU name].h (variable)

Example: (For Hitachi H8/532 MPU) 532. h

3) Source file of each function module

Purpose: Each function module has a source file. Typically there are multiple files for each main module source file function.

Extension: c (Source)

Name: main. c (fixed)

[Function module name].c (variable)

Example: main. c
denst. c
alarm. c
disp. c

4) Header file of each function module

Purpose: Declares the prototype of the function declared in the source file for each function module.
Extension: h (header)
Name: main. h (fixed)
[Function module name]. h (variable)
Example: main. h
denst. h
alarm. h
disp. h

5) Global variable definition source file

Purpose: Defines global variables commonly referred to by each function module.
Extension: c (Source)
Name: ram. c (fixed)

6) Global variable definition header file

Purpose: Header file declaring global variables for referencing from each functional module.
Extension: h (header)
Name: ram. h (fixed)

7) Constant definition source file

Purpose: Source file of constant variable definitions for preventing change to the definition variable value. Mainly used for referencing calibration curve table and comment table addresses.
Extension: c (Source)
Name: constant. c (fixed)

8) Constant definition header file

Purpose: Header file for conveying to other function modules that the variable defined in the constant definition source file is constant (unchangeable). Mainly used for referencing calibration curve table and comment table addresses.
Extension: h (header)
Name: constant. h (fixed)

9. Source code creation

Overview of conventions for standard application coding

Rule	New code	Existing code (Proven in use)	Existing code (Not Proven in use)
1 to 2	Required[*1]	Request[*1]	Required[*1]
3 to 8	Required[*1]	Not required	Request[*1]
9	Required[*1]	Not required	Request[*1]
10	Required[*1]	Not required	Not required

[*1]If compliance to the coding conventions cannot be met, verify the source code's suitability with a Design Review (DR).

C language rules
[*1] Page refers to the page number of the MISRA-C 2004 C Reliability Guide for C language for embedded developers.
[*2] If you start with P, R or M with MISRA rules, please refer to the Coding Method Guide for Embedded Software Development [C Language Version].

Law of subset of C language		Page[*1]	MISRA rule
1	About function/variable declaration		
(a)	Minimisation of run-time failures shall be ensured by the use of at least one of (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.	p. 316	21.1 (Req)
(b)	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	p. 96	8.1 (Req)
	Whenever an object or function is declared or defined, its type shall be explicitly stated.	p. 101	8.2 (Req)
(i)	Functions shall not be defined with variable numbers of arguments.	p. 231	16.1 (Req)
(ii)	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.	p. 71	5.1 (Req)
(iii)	No identifier name should be reused.	p. 83	5.7 (Adv)
(iv)	A const declared variable must have a single variable name	-	-
2	All arithmetic expressions and assignment statements		
(a)	Limited dependence should be placed on C's operator precedence rules in expressions	p. 163	12.1 (Adv)
(b)	When performing arithmetic operations or comparisons of expressions mixed with signed and unsigned, an explicit cast to the expected type shall be performed.	p. 37[*2]	R2.42
3	All logical expressions		
(a)	Limited dependence should be placed on C's operator precedence rules in expressions	p. 163	12.1 (Adv)
	The value of an expression shall be the same under any order of evaluation that the standard permits.	p. 166	12.2 (Req)
	The right-hand operand of a logical && or || operator shall not contain side effects.	p. 171	12.4 (Req)
(b)	Unsigned integer constant expressions shall be described within the range that can be represented with the result type.	p. 34[*2]	R2.3.1
(c)	Assignment operators shall not be used in expressions to examine true or false, except for conventionally used notations.	p. 94[*2]	M3.3.3

(d)	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	p. 96	8.1 (Req)
(e)	Floating point expressions shall not be tested for equality or inequality.	p. 194	13.3 (Req)
4	All pointer expressions		
(a)	Destination pointed by a pointer shall be referenced to after checking that the pointer is not the null pointer.	p. 52*2	R3.2.2
(b)	Conversions shall not be performed between a pointer to a function and any type other than an integral type.	p. 155	11.1 (Req)
(c)	Before accessing the pointer, its data type and value range shall be verified. before accessing	-	-
(d)	Pointer arithmetic shall only be applied to pointers that address an array or array element.	p. 248	17.1 (Req)
(i)	Pointer subtraction shall only be applied to pointers that address elements of the same array.	p. 250	17.2 (Req)
(ii)	A cast should not be performed between a pointer type and an integral type.	p. 159	11.3 (Adv)
	A cast should not be performed between a pointer to object type and a different pointer to object type.	p. 160	11.4 (Adv)
(iii)	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	p. 259	17.6 (Req)
5	All goto statements		
(a)	The goto statement shall not be used.	p. 209	14.4 (Req)
6	All switch statements		
(a)	The type contained in a *switch* statement shall be consistent throughout the project.	-	-
(b)	A *switch* expression shall not represent a value that is effectively Boolean.	p. 228	15.4 (Req)
(c)	The final clause of a *switch* statement shall be the *default* clause	p. 225	15.3 (Req)
(d)	Every *switch* statement shall have at least one *case* clause.	p. 230	15.5 (Req)
(e)	An unconditional *break* statement shall terminate every non-empty *switch* clause.	p. 223	15.2 (Req)
7	Bit field, bit representation and bit operator		
(a)	Bitwise operators shall not be applied to operands whose underlying type is signed.	p. 180	12.7 (Req)
(b)	The implementation defined behaviour and packing of bitfields shall be documented if being relied upon.	p. 64	3.5 (Req)
8	Definition of a set of software and tools that utilize the whole function		
(a)	Assembly language shall be encapsulated and isolated	p. 52	2.1 (Req)
(b)	No reliance shall be placed on undefined or unspecified behaviour.	p. 47	1.2 (Req)
(c)	To use characters other than those defined in the language standard for writing a program, the compiler specifications shall be confirmed, and their usage shall be defined.	p. 124*2	P1.2.1
(d)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.	p. 49	1.3 (Req)
9	Guidelines for error prevention		
(a)	Trigraphs shall not be used.	p. 70	4.2 (Req)
(b)	Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures.	p. 129	9.2 (Req)
(c)	In an enumerator list, the “=” construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialised.	p. 132	9.3 (Req)

(d)	If a function returns error information, then that error information shall be tested.	p. 246	16.10 (Req)
(e)	*#include* statements in a file should only be preceded by other preprocessor directives or comments.	p. 272	19.1 (Adv)
(f)	All uses of the *#pragma* directive shall be documented and explained.	p. 63	3.4 (Req)
(g)	Precautions shall be taken in order to prevent the contents of a header file being included twice.	p. 294	19.15 (Req)
(h)	Unions shall not be used.	p. 270	18.4 (Req)
(i)	Automatic variables of the same type used for the similar purposes may be declared in one declaration statement, but variables with initialization and variables without initialization shall not be mixed.	p. 58*2	M1.2.1
(j)	Whenever an object or function is declared or defined, its type shall be explicitly stated.	p. 101	8.2 (Req)
(k)	The comma operator shall not be used.	p. 187	12.10 (Req)
(l)	There shall be no unreachable code.	p. 202	14.1 (Req)
(m)	0 shall be used for the null pointer. NULL shall not be used in any case.	p. 110*2	M4.6.1
(n)	An area of memory shall not be reused for unrelated purposes.	p. 268	18.3 (Req)
(o)	Dynamic heap memory allocation shall not be used.	p. 308	20.4 (Req)
(p)	Interrupts shall be limited only to simple processes.	-	-
(q)	The signal handling facilities of <*signal.h*> shall not be used.	p. 312	20.8 (Req)
(r)	All usage of implementation-defined behaviour shall be documented.	p. 60	3.1 (Req)
(s)	Functions shall not call themselves, either directly or indirectly.	p. 232	16.2 (Req)
1O	Approval process specified in the justification functional safety plan		
(a)	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.	p. 68	3.6 (Req)
B. 3. 3 Rules for modular approach			
(a)	A software module must have only a single task.	-	-
(b)	Connections between software modules shall be strictly defined. Software must be robustly consistent.	-	-
(c)	Source code shall be limited so that a module does not exceed 100 lines excluding comments.	-	-
(d)	The cyclomatic complexity of a module should be kept to 10 or less.	-	-
(e)	A function shall have a single point of exit at the end of the function.	p. 214	14.7 (Req)
(f)	A software module must be used in conjunction with other software modules via an interface. Global variables should be made available only within a structure, with access restrictions, and only for the instance for which it is used.	-	-
(g)	All usage of implementation-defined behaviour shall be documented.	p. 60	3.1 (Req)
(h)	Unnecessary variables shall not be used.	-	-
B. 3. 4 Rules for structured programming			
(a)	Follow the rules defined in section 3. 3 (Rules for Modular Approach).	-	-
(b)	Follow the rules defined in section 3. 2 (Rules for Language Subset of C++).	-	-
(c)	Simplify as much as possible the relationship between input and output so as to minimize the number of paths when creating a software module.	-	-
(d)	For any iteration statement there shall be at most one *break* statement used for loop termination.	p. 212	14.6 (Req)

(e)	Numeric variables being used within a *for* loop for iteration counting shall not be modified in the body of the loop.	p. 199	13.6 (Req)
(f)	The three expressions of a *for* statement shall be concerned only with loop control.	p. 197	13.5 (Req)
-	Software that has been recognized as proven in use is not subject to the rules above. However, if critical design changes are made to the hitherto proven-in-use source code, the rules above shall apply to the redesign of the source code.	-	-

References

1. Hiroshi Shima, *Guide to Embedded Developers MISRA-C 2004 Guide for High Reliability in C Language Use*, Japan Standards Association

2. Kouji Hayami, *Coding method guide for embedded software development [C language version]*, Shoeisha Co., Ltd.

-END-