C Programming Standards

IPMI Development

(aka: IPMI Coding Standard)

Author: Vlad Rakocevic
Email: vlad.rakocevic@radisys.com
Owner: Vlad Rakocevic

002-nnnnn-0000
Revision 0.90
September 7, 2018

Approvers:

Reviewers:

Note: To verify signoff, see the on-line repository.

Name	Title/Department
Nilan Naidoo	SW Architect
Vlad Rakocevic	
Jared Lewis	SW Development Engineer
Arun Kalluri	SW Development Engineer
Rahul Ghosh	SW Development Engineer
Scott McClelland	SW Development Engineer
Paul Burkey	SW Development Engineer
Mike Meyer	SW Development Engineer
Erik Johansen	SW Development Engineer

Document control

This is a copy of a RadiSys controlled document. This document is valid only if it is a copy of the latest version available in the on-line repository or from the document owner. To validate the latest version of this document, refer to the on-line repository or contact the document owner.

On-line repository:
Source filename:

Agile
C Coding Standard for IPMI v.90

Preface

Revision History

All revisions that start with zero (i.e., 0.5, 0.65, etc.) are preliminary drafts for internal reviews.

Table 1. Revision history

No.	Date	Author	Description
0.7	04/12/06	Nilan Naidoo	Initial draft. Created document from ARTIC C Programming Std and Guideline 1.0
0.71	04/17/06	Vlad Rakocevic	Minor updates. Modified the file headers for .c and .h files.
0.72	04/25/06	Vlad Rakocevic	Updates based on some electronic comments from Jared L. Use Rsys EFX013-C format.
0.73	05/02/06	Vlad Rakocevic	Updates after 2 review sessions.
0.90	05/11/06	Vlad Rakocevic	Doc sent for negative ballot
0.91	05/15/06	Vlad Rakocevic	Updates from electronic review from Nilan

Notational Conventions

Contents

Figures

ERROR! NO TABLE OF FIGURES ENTRIES FOUND.

Tables

1. Overview

This document is intended to define C programming standards and guidelines for IPMI software development. It based on a coding standard described in [1].

2. IPMI Software Standards

This chapter provides the standards that relate to IPMI software development. The standards stated here provide a precise, concise reference for a software developer to use.

A reader will note a numbering convention for the standards. The format is xd, where x is a letter indicating the category of standard and d is a decimal number (can be multiple digits), within the standard category, to uniquely identify the standard. Each standard also contains a short name, shown in bold, to make referencing them easier.

2.1 General Standards

G1 **Save Without Tabs** - All C source files and header files **shall** be saved without tabs. Leaving the tab characters in the file creates unusual looking files when they are archived on the host system.

G2 **Code Width** – Try to keep source lines to about 80 characters, if it is possible and practical. Although modern day editors supports much larger line lengths and wrap-around functionality, extremely long lines usually mean that the code is getting out of control (nesting too deep, not using pointers into complex structure components).

G3 **In-line Majority** - In-line comments **should** be included with a majority of the instruction lines in the file.

G4 **In-line Comment Alignment** - In-line comments **should** (though not must) generally be aligned to the same starting column within the file. Recognizing that it is usually not possible to align all in-line comments in the entire file, they **may** at least generally be aligned to the same starting column within a code section, paragraph, or header file section.

```
void foo (int fruId,                    //in: passed in argument containing the FRU ID
          char pSensorId                //in: pointer to Sensor number
){

     return;                            //exit from the function
}
```

G5 **White Space** - White space **shall** be used appropriately to improve readability of the program.

G6 **#include Position** – All #include statements **shall** be placed at the top of the file.

G7 **Avoid using native C data types –** The native C data types **shall** not be used. Data types defined in globdef.h files from Clear Case repository **shall** be used instead.

2.2 Naming Standards

N1 **Avoid Customized Abbreviations** – Customized abbreviations **shall** be avoided unless the abbreviation is obvious and consistently used (See Appendix A, Standard Abbreviations).

N2 **Uppercase Constant Names** - The characters in all constant names **shall** be uppercase. Use underscores to separate words within the name.

N3 **Uppercase Macro Names** - The characters in all macro names **shall** be uppercase. Use underscores to separate words within the name.

N4 **No Beginning Underscore** - Identifiers beginning with an underscore character **should** generally not be used, in order to avoid conflict with compiler generated identifiers. A common

exception is to begin structure tags with an underscore so a variable using the same name (minus the underscore) can be declared.

N5 **Variable and Function Naming Convention within Global Scope** - In general all names **shall** use the following conventions:

<type><ModulePrefix><FileContext><Name>

Where:

type is a type specifier for the following examples below only:

g : for global variable.

p : if the variable is a pointer

pp : if the variable is a double pointer

For all other variable types no <type> specifer is required.

ModulePrefix is an acronym or the full name of the module. All exported functions, global variables, global typedefs, enums and Macros **shall** use this prefix.

ModulePrefix when used in function names **should** start with a small letter.

FileContext is used to indicate the submodule if the module implementation spans multiple files. This is optional if the module spans only one file.

Name specifies the rest of the name. The first letter of each word **shall** be capitalized.

See the C Sample code for examples of naming convention usage.

2.3 #ifdef Standards

The use of the term #ifdef in the context of the following standards is meant to imply other applicable conditional compiler directives as well.

I1 **One Function Per #ifdef** - No more than one function **should** be contained within a #ifdef. Repeat the #ifdef for each function.

I2 **Function #ifdef Position** - If the #ifdef is for the entire function, the #ifdef **should** be placed immediately before the function header. Closing #endif statement **should** be immediately after the function's closing bracket.

I3 **#ifdef Code Alignment** - The code inside the #ifdef **shall** be indented to appropriately align with the code immediately above it.

I4 **#ifdef Containment** - All code within the #ifdef **shall** be completely contained within the C code block where the #ifdef started.

I5 **#endif Comments** - All #endif statements **should** contain a comment indicating the symbol name used in the opening #ifdef.

I6 **Nested #ifdef indented** - All #if and #ifdef statements **should** indented to indicate their level of nesting.

I7 **Minimize the use of #ifdefs** - If you find a lot of code using #ifdef to do different things for different platforms, consider consolidating the logic into a few functions that hide all of the complexity.

2.4 Function Standards

F1 **Function Headers** - Functions **shall** be preceded by a Function Header, as described below. This function header is based on syntax required for **Doxygen**. The <> and included text **shall** be replaced by appropriate text as described within the brackets:

```
/*!      \brief <A brief description of the function purpose>
 *
 *         <List the purpose of the function in more detail. This should include
 *         information to help the user better understand the function, and
 *         should include any pre-conditions, post-conditions, algorithms.
 *       If there is any info that is not obvious by looking at the code then
 *       it should be included here.>
 *
 *         \param <param1 name>  <param1 details>
 *         \param <param2 name>  <param2 details>
 *         …
 *         \retval <return value1>  <condition when value1 is returned>
 *         \retval <return value2>  <condition when value2 is returned>
*          …
 */
<retype> <FuncName>( <param1_type> <param>1, <param2_type>< param2>, …)
{
}
```

> NOTE: Function prototype descriptions **shall** be done in .C files as oppose to header files. Even though this approach may request that **Doxygen** tool to be ran on all .H and .C files to get the complete documentation of the data structures and functions, it was chosen for maintenance reasons (it is easer to update description for example of the function algorithm in the source code rather to switch between header file and implementation).

F2 **Function Starting Column** - Function definitions, along with the associated Function Header, opening brace, and closing brace, **shall** start in column 1.

F3 **Meaningful Function Names** - Function names **shall** accurately convey what the function does (no cryptic names). The function name **shall** follow the conventions of N5, and the name portion **shall** specify the action that the function is performing. Function name **shall** start with small letter. For example, devRead(), sensorDigitalUpdate(), msgRmcpInit().

F4 **Non-uppercase Function Names** - Function names **shall** not conflict with the macro naming convention. This means the function name cannot be all upper case.

F5 **ANSI Function Declaration** - Functions **should** be declared using the ANSI C form instead of the “old style” Kernnigan & Richie (K&R) form. The example below demonstrates this. Please note, the Function Header has been left off for brevity, but **should** be included in actual software.

```
long provideExample(int     arg1,
                    long  arg2,
                   char *arg3)
{
  :
}
```

Rather than:

```
long provideExample (arg1, arg2, arg3)  //This is K&R form
int   arg1,
long  arg2,
char *arg3)
{
  :
}
```

F6 **Functions Returning Void** - Functions that return no value **shall** be declared as type void.

F7 **Function Parameter Order** - Functions **should** define parameters in the following order (each group included only if it applies):

- Input (to the function) parameters
- Combined input/output parameters
- Output (from the function) parameters

F8 **Cover Functions** - When using C Library functions that are not included in the standard ANSI C Libraries, as listed in the appendices, or system functions, create a cover function or macro from which to call the non-standard function. This localizes the dependency on the non-standard functions and will help keep code portable, or at least, easier to change.

F9 **Function Prototype Format** - Function prototypes **shall** specify an argument list and return type.

F10 **Variable Instruction Separation** - At least one blank line **should** separate the declaration of variables from the first instruction.

F11 **Limit Global Data** - Functions **shall** operate on parameter data and avoid the use of global data as much as possible.

2.5 C Source Standards

C1 **C File Extension** - All C source files **shall** use a .c file extension.

C2 **C File Header** - All C source files **shall** start with a standard File Header, which includes a RadiSys standard copyright, and a description of the file/module functionality. Please refer to EWX011 document for the latest copyright wording. The format of this file header is based on syntax required for **Doxygen** tool.

```
/*****************************************************************************/
/*
 *  RadiSys Confidential
 *  RadiSys Platform Management Software
 *  (C) Copyright RadiSys Corporation 2006.
 *  All Rights Reserved.
 *  Reproduce Under License
 *
 *  The source code for this program is not published or otherwise
 *  divested of its trade secrets, irrespective of what has been
 *  deposited with the U.S. Copyright Office.
 */
/*****************************************************************************/

/*! \file    sample.c
```

```
 * \brief   Sample C code to demonstrate concepts in Programming Standard
 *
 *          This file contains sample C code to demonstrate the concepts
 *          iscussed in the C Programming Standards and Guidelines.
 *          This sample application will accept input data from a user
 *          and pass it on to other programs.
 *
 *          The first character of the input buffer indicates the buffer
 *          type (data buffer or control buffer). For control buffers,
 *          the second character of the buffer indicates the control
 *          command to perform (open, close, read).
 *
 *          In all cases, a Transfer Control Block (TCB) is built for
 *          passing the buffer to the target routine. These target
 *          routines are not shown in this sample code.
 *
 *          Note: This sample code does not represent a complete working
 *          program.
 *
 *
 *          Note: In the change activity below, the first column is for
 *          the initials of the programmer that made the change. The
 *          second column is for the date the change was made.
 */
```

C3 **Function Modularized** – Run-on code **should** be eliminated with the liberal use of functions.

C4 **Use Constants** – Constants **shall** be used to make your code more readable and easier to change. For example:

```
int Buf[20];                        //Will hold input characters
int i;                              //Loop variable and index
:
// Initialize the buffer
for (i=0; i<20; i++)
        Buf(i) = 0;
```

Is better coded as:

```
#define INPUT_BUFFER_SIZE   20      //Size of input buffer

//This code is in the C source file
int Buf[INPUT_BUFFER_SIZE];         //Will hold input characters
int i;                              //Loop variable and index
:
// Initialize the buffer
for (i=0; i< INPUT_BUFFER_SIZE; i++)
          Buf(i) = 0;
```

C5 **C File #defines** - #defines **should** generally be placed in header files if they are used outside of the C source files, but when needed only in a C source file, they **should** be placed at the top of the C file. **Doxygen !\def** command **shall** proceed #define, regardless whether we use it for a constant definition or a macro, whenever is a need to document it. Example below is used to illustrate it:

```
/*! \def EXAMPLE_DEFINE
```

```
*\brief Define to demonstrate usage of Doxygen documentation of the constants.
*/
#ifndef EXAMPLE_DEFINE
#define EXAMPLE_DEFINE
```

C6 **One Variable Per Line** - Each variable **should** be declared on a separate line and **should** be described by a comment regarding its use. Several variables may be declared together on one line only if they meet all of the following conditions:

- They are of the same type.
- They are used in a similar manner.
- They can be described by the same comment.

C7 **Local Variable Declarations** – Local variables **should** be declared at the beginning of the function in which they are used, before any executable instructions.

C8 **Local Variable Initialization** – Local variables **should** only be initialized on the declaration line if the initialization value is a constant. Separating initializations to actual lines of code provides:

- More readable code.
- Allows space for the recommended in-line comment.
- Allows placing debug code before the variable is initialized.

C9 **Single Purpose Of Variables** – Variables within a program **should** be used for a single purpose, the one that the name implies.

C10 **Register Variables** – Register variables **should** not be used unless the code is part of a performance critical code path.

C11 **One Statement Per Line** - No more than one program statement **should** be on a program line. This will help with readability and debugging.

C12 **Keep Statements Simple** – Over compact C code **shall** be avoided.. C is a powerful language that allows you to do many things on one line. However, this makes code hard to read, difficult to comment, and impossible to get debugging statements between the clauses of the statement. Performance is generally not an issue since newer compilers can optimize variable references over multiple statements. In the following example, consider that:

```
// If x and y are both true, we want the first word
// off the stack. Otherwise we want the second word.
x += (IsItTrue(x,y)) ? pop(stack), stack.y :
                pop(stack), pop(stack), stack.y
        would be better written as:
// If x and y are both true, we want the first word
// off the stack. Otherwise we want the second word.
if (IsItTrue(x,y) != 0) pop(stack);     //Want 2nd word. Do extra pop

pop(stack);                             // Gets 1st or 2nd stack word
x += stack.y;                           //Add stack word to x
```

C13 **Source Code Indentation** - Source statement blocks **shall** be indented 4 columns as they are nested under their opening statements (if, while, do, switch, etc.). Indentation must be consistent throughout the project. Indentation creates a powerful visual feedback. Inconsistent indentation can be very confusing to the reader as a program quickly scrolls by in a code editor. See the Sample C Source File in the appendices for an example of indentation.

C14 **Paragraph Size** - Code paragraphs **should** generally include ten or less instructions.

C15 **Paragraph Comment** - Code paragraphs **should** be preceded by a blank line and a paragraph comment.

C16 **Paragraph Purpose** – The instructions within a code paragraph **should** have a common, stated purpose.

C17 **Code Sectioning** - Long, contiguous groups of code paragraphs **should** be broken into sections by the use of eye catchers.

C18 **Section Purpose** – The paragraphs within a code section **should** have a common, stated purpose.

C19 **Arithmetic Type Casting** – Casts **shall** be explicitly used in arithmetic expressions of different types. It prevents the occasional error caused by automatic type conversions.

C20 **Pointer Type Casting** - All assignments of pointers of differing types **should** include a type cast to insure proper operation of the assignment.

C21 **Parentheses For Precedence** - Operator precedence is not always intuitive in C. Parentheses **shall** be used to keep the order of evaluation of complicated expressions clear.

C22 **One Pre/Postfix Per Instruction** - No more than one prefix or postfix operator **should** be used per statement. The over use of these operators is confusing and makes programs difficult to read.

C23 **Brace Positioning** – The opening and closing braces of statement blocks **shall** be placed on lines by themselves, column aligned to the statement (if, while, do, switch, etc.) that opened the block. The only exception to this is an in-line comment on the line. However, plenty of white space **should** be left between the brace and the comment.

C24 **Brace Line Of Sight** - Looking down the column containing the opening brace, nothing **should** come between the opening and closing braces (with the exceptions of #ifdef lines and goto labels). This allows subordinate code, as well as the end of a particular statement, to be readily identifiable by scanning down from the opening brace to find the closing brace. See Chapter 2, General Issues, for more information on brace positioning.

C25 **Avoid Conditional Assignments** - Conditional statements **should** not contain assignment statements (see the example below). While these nested assignments are certainly C legal, they:

- Are easily missed when reading programs.
- Make desk checking difficult for the common error of typing = when you meant ==.
- Make debugging more difficult since intervening debug statements cannot be put in.

In the following example:

```
if (a = x) ...
```

The programmer probably meant:

```
if (a == x) ...
```

An allowable exception to this standard is the following common sequence used for assigning and checking the return value of a function:

```
if (rc = MyFunction(x)) ...
```

C26 **Constants in conditional statements** - In conditional statements with a constant, a constant **should** be used on the left hand side of the double assignment operator, e.g.

```
if(MAX_NUMBER ==i)
```

```
{

}
```

Rather than

```
if(i== MAX_NUMBER)

{


}
```

C27 **Avoid if (!Variable)** – Do not use the negative logic form shown below without very meaningfully naming the variable:

```
if (!Variable) ...                  //Instead: if (0 == Variable)
```

C28 **Else Alignment** - In an if-else construct, the else keyword **should** line up below the matching if keyword. It **should** not be placed on the line with a previous closing brace. The construct **should** look like this:

```
// Do an if to demonstrate else keyword placement
if (a == b)
{
  :
}
else
{
  :
}
```

C29 **else-if** Alignment - In an if-else if-else if-else construct, the else keywords **should** line up below the first if keyword and the successive if keywords **should** appear in line with the previous else keywords. The number of else-if clauses **should** be limited since determining how the code got to this point can become difficult. Note that in some cases, a switch statement can be used and would be the preferred method. The if-else if-else if-else construct **should** look like this:

```
// Do an if to demonstrate else if construct
if (a == 1)
{
  :
{
else if (a == 2)
{
 :
}
else if (a == 3)
  {
 :
}
else
  {
 :
}
```

C30 **Curly Brackets required even on a single line** – Curly brackets **shall** be placed around even a single C code statement, after the conditional statement. For example:

```
if (MAXNUMBER == i){
i = 0;
}
```

C31 **Switch Alignment** - In a switch statement, the case keywords **should** be indented from the opening switch keyword. The instructions within the case **should** be indented from the case keyword. The construct **should** look like this:

```
// Do a switch to demonstrate the construct
switch (a)
{
  case 1:
    :
    break;

  case 2:
    :
    // Fall through into next case

  case 3:
    :
    break;

  default:
    ;
}
```

C31 **Case Fall Through** - When a case in a switch statement is not terminated with a break statement, a comment **shall** be included to indicate the case is "falling into" the next case. See the preceding example.

C32 **Default Case** - The last case in a switch statement **should** be a default case (using the default keyword). When there is nothing to do in the default, simply put a semicolon (;) to indicate this. See the preceding example.

C33 **Avoid Goto** - Code **should** be structured and goto statements **should** not be used unless absolutely necessary. The common exception to this is to use a goto within a function to jump ahead to a common point rather than coding excessive if/else statements. In no case **should** a goto jump outside a function or module.

C34 **Goto Label Column** - In the rare cases where a goto is used, the target label **should** start in column 1.

C35 **Do While Positioning** - In a do-while construct, the while keyword follows a space after the closing brace.

```
// This example shows the while keyword placement
do
{
  :
} while (c == d);
```

C36 **Right Shift Caution** – When using the right shift operator (>>) do not assume the new value of the high order (shifted in) bits.

2.6 Header File Standards

H1 **Header File Extension** - All header files **shall** use .h file extension.

H2 **Header File Header** - All header files **shall** start with a standard File Header, which includes a RadiSys standard copyright. Please refer to EWX011 document for the latest copyright wording. The header file header is followed with **Doxygen** style description section, which **shall** indicate the purpose of the header file.

```
/*****************************************************************************/
/*
 *  RadiSys Confidential
 *  RadiSys Platform Management Software
 *  (C) Copyright RadiSys Corporation 2006.
 *  All Rights Reserved.
 *  Reproduce Under License
 *
 *  The source code for this program is not published or otherwise
 *  divested of its trade secrets, irrespective of what has been
 *  deposited with the U.S. Copyright Office.
 */
/*****************************************************************************/

/*! \file    sample.h
 *  \brief   Sample header file to demonstrate concepts in Programming Standard
 *
 *                  This file contains sample .h file to demonstrate the concepts
 *          iscussed in the C Programming Standards and Guidelines.
 *
 */
#ifndef __SAMPLE_H__
#define __SAMPLE_H__
```

H3 **Framing #ifdef in C header files** – C hader files are often invoked from various source files and may also be included in C++ code. In order to avoid duplicate definitions and conflicts with C++ code, the contents of the header file **shall** be framed by a conditional ifdef and R_DECLS macro, as follows:

```
#ifndef __FILENAME_H__
#define __FILENAME_H__


R_BEGIN_DECLS


<Header file contents go here. See H4 for Element Order>


R_END_DECLS

#endif                          // __FILENAME_H__
```

Please note that FILENAME is the name of the file in all uppercase. Also note, that FILENAME_H is preceded by double underscores and followed by double underscores. The BEGIN_DECLS and R_END_DECLS macro are defined in the rtypes.h file as follows:

```
#ifdef  __cplusplus
# define R_BEGIN_DECLS  extern "C" {
# define R_END_DECLS    }
#else
# define R_BEGIN_DECLS
# define R_END_DECLS
#endif
```

H4 **Header File Element Order** – The elements of header files **should** be coded in the following order:

- Individual constants (grouped for commonality)
- Individual typedefs
- Structure definitions
- Function prototypes
- Macros

H5 **Typedefs** - Use typedef names for all user defined structures/unions. For documentation purposes, all typedefs **shall** be accompanied by **Doxygen** documentation commands.

```
/*! \typedef EXMP_STRUCT
 */
typedef struct {
.
.
.
}EXMP_STRUCT;
```

H6 **Field Value Constant Positioning** - Field value constants (#define directives) **may** always be placed near the top of the header file, before any data definitions. In addition, they **may** be placed in the header file based upon their use. If the constants are used for:

- **One individual field** - They **may** be placed just after the individual field.
- **Field(s) in one structure** - They **may** be placed just after the structure (and after any access shortcuts defined for that structure).

H7 **Enumerated Constants** – Constants that enumerate field values **should** do so explicitly, as opposed to using the enum feature of the C language.

H8 **Clean Structures** - Structure definitions **shall** not contain intervening statements that are not part of the structure. Specifically, no #define directives **should** be used within a structure definition. Structures that would need to be part of documentation **shall** be preceded by **Doxygen** commands. See the example below:

```
/*!
 *  \struct R_SLIST
 *
```

```
 *  The struct is used for each element in the singly-linked
 *  list. The data field holds the element's data, which can be a
 *  pointer to any kind of data, or any integer value using the
 *  Type Conversion Macros. The next field contains the link to
 *  the next element in the list.
 */
typedef struct _R_SLIST                 R_SLIST;
struct _R_SLIST
  {
    R_POINTER data;             /*!< Pointer to opaque data */
    R_SLIST next;               /!< Pointer to the next Slist node*/
  };
```

H9 **Structure Bit Fields** – Bit fields in structure definitions **should** not be used unless the bit positions assigned by the compiler are not important and where the data is internal to the program, not exchanged with other programs or systems.

H10 **Major Structure Order** – The elements of major structures **should** be coded as follows:

· Structure eye catcher (comment)

· Supporting structures (small structures referenced in the major structure, if only used by this major structure)

· Structure definition

· Access Shortcuts

· Field Value constants

H11 **Access Shortcuts** - #define directives **should** be used create access shortcuts to fields whose references would be lengthy due to structure/union nesting. Do not place them within the structure, but place them immediately after the structure to which they apply. Use the same naming conventions as used for the field the access shortcut references. See the sample files in the appendices for examples of defining and using access shortcuts (e,g, IPMC_IS_REDUNDANT(p)).

H12 **Access Shortcut Positioning** - Access shortcuts, if used, **should** be placed immediately after the definition of the structure containing the referenced field (but before any field value constants for the fields of the structure).

H13 **Access Shortcuts Comments** - Access shortcuts for the fields of a structure **should** start with a comment identifying them as access shortcuts and for what structure. For example:

```
/*!
*\brief Define access shortcuts for the Transfer Control Block structure
*/
#define ...

#define ...

:
```

H14 **Function Prototypes In Headers** - Place function prototypes in a header file if they are public (used from more than one C file). Function prototypes for functions that are local to one C file **may** be placed in the C file itself. They need not be so formal.

The following is an example of a prototype placed in a header file:

```
long provideExample(int   arg1,        // Description of arg1
                    long  arg2,        //Description of arg2
                    char arg3);        // Description of arg3
```

H15 **Macros documentation** – Macros that need to be documented **shall** follow the example below:

```
/*! \def MY_MACRO
*\brief This Macro does …
* Detail.
*/
*#define MY_MACRO    1 + 1
```

H16 **Macro Parentheses** - Use parentheses within macro definitions to make sure they map as intended when they are used. For example:

```
#define MY_MACRO    1 + 1


a = b * MY_MACRO;
        maps into:
q = b * 1 + 1;
```

which is b+1, not b*2 as was probably intended. This macro **should** be coded as follows:

```
#define MY_MACRO    (1 + 1)
```

H17 **EXTERN variables** - extern variables **should** only be declared in the header files if they are used outside of the .c file

H18 **#include** - If a header file has a dependency on additional header file – i.e. the user must include both files explicitly in the right order – make sure this is documented somewhere.

2.7 Data Standards

D1 **Appropriate Type Casting** -. Check that your data is properly defined to avoid excessive type casting.

D2 **Generic Pointer Type** - Use void * as the proper generic pointer. Do not use char * for this purpose. The following convenience types R_POINTER and R_CONST_POINTER are defined for casting to generic pointers

D3 **Avoid Type int** - Variables of type int **should** be avoided. Use a predefined typedef instead.

D4 **Floating Point Double** – For precision and consistency when using floating point variables, type double (64 bits) **should** be used instead of type float (32 bits).

D5 **Floating Point Format** – Do not make assumptions about the floating point format (e.g. IEEE vs. IBM370 vs. others)

D6 **Pointer Size** – Do not assume pointers are the same size as longs.

D7 **Uninitialized Data** - No assumption **should** be made as to the initial contents of data fields not explicitly initialized. Do not expect fields to be zero or null.

D8 **Endianess** – When **Endianess** matters, it **shall** be abstracted using macros.

D9 **Variable width** – When the size of the variable is important, then the predefined types U8, U16. U32, U64, S8, S16 S32 or S64 shall be used.

A Standard Abbreviations

This appendix provides abbreviations for common terms we encounter in our environment. Some are obvious. Other are often abbreviated different ways by different people.

Abbrev	Meaning
buf	buffer
char	character(s)
cntl	control
cnt	count
func	function
max	maximum
min	minimum
ptr	pointer

B Precedence of Operators

The table below provides the C operator evaluation precedence.

Operator Description	Associativity	Operators
Simple tokens		*identifier constant literal* (*expression*)
Primary Expression	Left to right	*f*( ) • [ ] • . • ->
Unary Postfix	Left to right	++ • --
Unary Prefix	right to left	++ • -- • - • + • ! • ~ • & • * • (*typename*) • sizeof
Multiplicative	Left to right	* • / • %
Additive	Left to right	+ • -
Bitwise Shift	Left to right	<< • >>
Relational	Left to right	< • > • <= • >=
Equality	Left to right	== • !=
Bitwise Logical AND	Left to right	&
Bitwise Exclusive OR	Left to right	^
Bitwise Inclusive OR	Left to right	|
Logical AND	Left to right	&&
Logical OR	Left to right	||
Conditional	right to left	? :
Assignment	right to left	= • += • -= • *= • /= • <<= • >>= • %= • &= • ^.= • |=
Comma	Left to right	,

C Sample code

```
/*****************************************************************************/
/*
 *  RadiSys Confidential
 *  RadiSys Platform Management Software
 *  (C) Copyright RadiSys Corporation 2006.
 *  All Rights Reserved.
 *  Reproduce Under License
 *
 *  The source code for this program is not published or otherwise
 *  divested of its trade secrets, irrespective of what has been
 *  deposited with the U.S. Copyright Office.
 */
/*****************************************************************************/

/*! \file    ipmcCfg.h
 *  \brief   sub-section IPM Controller Configuration
 *
 *   In general each element in a configuration  may be either a constant (i.e.
 *   its value never changes through the life of the IPMC) or variable (i.e. its
 *   value changes through the life of the IPMC). The constant elements should be
 *   stored in the FLASH area so that we don't use RAM unnecessarily since it is
 *   limited on the H8 processor.Therefore to optimize the RAM usage each
 *   structure will have pointers to two structures:
 *      - pC - which points to the constant or Read-Only parts, and
 *      - pV - which points to the variable or Read/Write parts.
 *   From the the programmers point of view the location of the elements (either
 *   Constant or Variable part) should be hidden so that it allows the elements to
 *   be moved around without requiring modification of the code. This can be
 *   achieved by providing macros to get each element of the structure.
 */

#ifndef __IPMCCFG_H__
#define __IPMCCFG_H__


R_BEGIN_DECLS


/*!
 *  \struct  IPMC_RO_INFO
 *  The following structure is the root configuration of the IPMI Controller. All
 *  config should be accessible through this structure, and there should be only
 *  one such structure created.
 */

typedef struct _IPMC_RO_INFO
{
  BOOL    fIsRedundant;           /*!< True if the IPMC is in a redundancy group */
  BOOL    fIsShmc;                /*!< True if the IPMC performs the ShMC role */
  BOOL    fIsAmcCarrier;          /*!< True if the IPMC performs AMC carrier role */
  BOOL    fHasRtm;                /*!< True if the IPMC controls an RTM */
  U8      NumMsgChannels;         /*!< Number of IPMI messaging Channels supported */
  PMSG_CHANNEL_T pMsgChannel;     /*!< Pointer to array of message channel descriptors */
  PFRU_INFO_T pFruInfo;           /*!< Pointer to the FRU Info object.*/
  PSDR_GTABLE_T pSdrGtable;       /*!< Pointer to the global SDR Table */
  PSENSOR_GTABLE_T pSensorGtable; /*!< Pointer to the global Sensor Table */
  PCONTROL_GTABLE_T pControlGtable; /*!< Pointer to the global control table */
  PDEVICE_TABLE_T pDeviceTable;   /*!< Pointer to the device table */

  /* --- TODO ADD more parameters here */
} IPMC_RO_INFO_T, PIPMC_RO_INFO_T;


/*!
 *  \struct  IPMC_VAR_INFO_T
 *  The following structure is the root configuration of the IPMI Controller.
 *  All config should be accessible through this structure, and there should be
 *  only one such structure created.
```

```
 */
typedef struct IPMC_VAR_INFO_T
{
  U8      hwAddress;             /*!< Hardware address of the IPMC. */
} IPMC_RW_INFO_T,PIPMC_RW_INFO_T;

/*!
 *  \struct   _IPMC_INFO_T
 */
typedef struct _IPMC_INFO_T
{
  PIPMC_RO_INFO_T pC;          /*!< Constant info of IPMC */
  PIPMC_RW_INFO_T pV;          /*!< Variable info of IPMC */
} IPMC_INFO_T, *PIPMC_INFO_T;

/*!
* \brief  These Macros provide methods to access fields of the IPMC structure
*/
#define IPMC_IS_REDUNDANT(p)          (p)->pC.fIsRedundant
#define IPMC_IS_SHMC(p)               (p)->pC.fIsShmc
#define IPMC_IS_AMCCARRIER(p)         (p)->pC.fIsAmcCarrier
#define IPMC_HAS_RTM(p)               (p)->pC.fHasRtm
#define IPMC_NUM_MSG_CHANNELS(p)      (p)->pC.NumMsgChannels
#define IPMC_MSG_CHANNEL_PTR(p)       (p)->pC.pMsgChannel
#define IPMC_FRU_INFO_PTR(p)          (p)->pC.pFruInfo
#define IPMC_HW_ADDRESS(p)            (p)->pV.u8HwAddress

/*!
 * \struct  _MSG_CHANNEL_RO_INFO_T
 * \brief   This structure provides the configuration of the messging channels.
  -- Message.h
*/
typedef struct _MSG_CHANNEL_RO_INFO_T
{

}MSG_CHANNEL_RO_INFO_T,*PMSG_CHANNEL_RO_INFO_T;



/*!
 *\struct  _MSG_CHANNEL_RW_INFO_T
 *\brief   This structure provides the configuration of the messging channels.
  -- Message.h
*/
typedef struct _MSG_CHANNEL_RW_INFO_T
{
}MSG_CHANNEL_RW_INFO_T,*PMSG_CHANNEL_RW_INFO_T;


/*!
*\struct  _MSG_CHANNEL_INFO_T
*\brief Yet another MSG Structure
*/
typedef struct _MSG_CHANNEL_INFO_T
{
  PMSG_CHANNEL_RO_INFO_T pC;          /*!< Constant info of the Message Channel */
  PMSG_CHANNEL_RW_INFO_T pV;          /*!< Variable info of the Message Channel */
}MSG_CHANNEL_INFO_T,*PMSG_CHANNEL_INFO_T;


/*!
*\struct FRU_RO_INFO_T
*  This structure provides the configuration of all the FRUs controlled by the
IPMC. This
* structure contains the info that is global to all FRUs, and a list of AtcaFru
devices
* that contain the configuration and status of each FRU controlled by the IPMC.
The first
*  entry in the list is always present and is used to describe FRU 0.
*/
```

```
typedef struct _FRU_RO_INFO_T
{
  PFRU_DEVICE_T pFruDevice;             /*!< Pointer to the array of FRU devices
*/
}FRU_RO_INFO_T,*FRU_RO_INFO_T;


/*!
*\struct FRU_INFO_T
*  This structure provides the configuration of all the FRUs controlled by the
IPMC. This
* structure contains the info that is global to all FRUs, and a list of AtcaFru
devices
* that contain the configuration and status of each FRU controlled by the IPMC.
The first
*  entry in the list is always present and is used to describe FRU 0.
*/
typedef struct _FRU_RW_INFO_T
{

}FRU_RW_INFO_T,*FRU_RW_INFO_T;

typedef struct _FRU_INFO_T
{
  PFRU_RO_INFO_T pC;          /*!< Constant info of the FRU */
  PFRU_RW_INFO_T pV;          /*!< Variable info of the FRU */
}FRU_INFO_T,*FRU_INFO_T;

/*!
* \struct CONTROL_INFO_T
* \brief  Control info for a specific FRU
*/
typedef struct _CONTROL_INFO_T
{
  U8      u8DeviceId;                   /*!< Id of the device that implements the
control*/
  U8      Param;                        /*!< Parameter to pass to the device
functions */
  U8      PorState;                     /*!< set control to this state at POR */

}CONTROL_RO_INFO_T,*CONTROL_RO_INFO_T;

/*!
* \struct FRU_RW_INFO_T
*\brief FRU Read/Write struct
*/
typedef struct _FRU_RW_INFO_T
{
  U8      State;                        /*!< on/off/pulse in progress */
}CONTROL_RW_INFO_T,*CONTROL_RW_INFO_T;

typedef struct _FRU_INFO_T
{
  PFRU_RO_INFO_T pC;          /*!< Constant info of the control */
  PFRU_RW_INFO_T pV;          /*!< Variable info of the control */
}CONTROL_INFO_T,*CONTROL_INFO_T;


/*!
*\union EXMP_UNION
*\brief A union example
*/
typedef union
{
U8 FRU_ID;              /*!< FRU identifier */
U16 *pFRU_CHAR;         /*!< Pointer to alias assigned to FUR Device */
} EXMP_UNION;

/*!
*\enum ENUM_T
```

```
*\brief A enum example
*/
  typedef enum
{
      U8 EVal1,      /*!< enum value 1 */
      U8 EVal2       /*!< enum value 2 */
 } ENUM_T;

R_END_DECLS
#endif                                  // __IPMCCFG_H__
```

D References

D.1 Related Documents

D.1.1 RadiSys Documents

C Programming StandardsAnd GuidelinesVersion 1.0, Radisys Doc, September 07, 1999[1]

D.1.2 Other Documents

DocName. Publisher, Month Day, Year.

D.1.3 Industry-Standard References

Table 2. Industry-Standard References