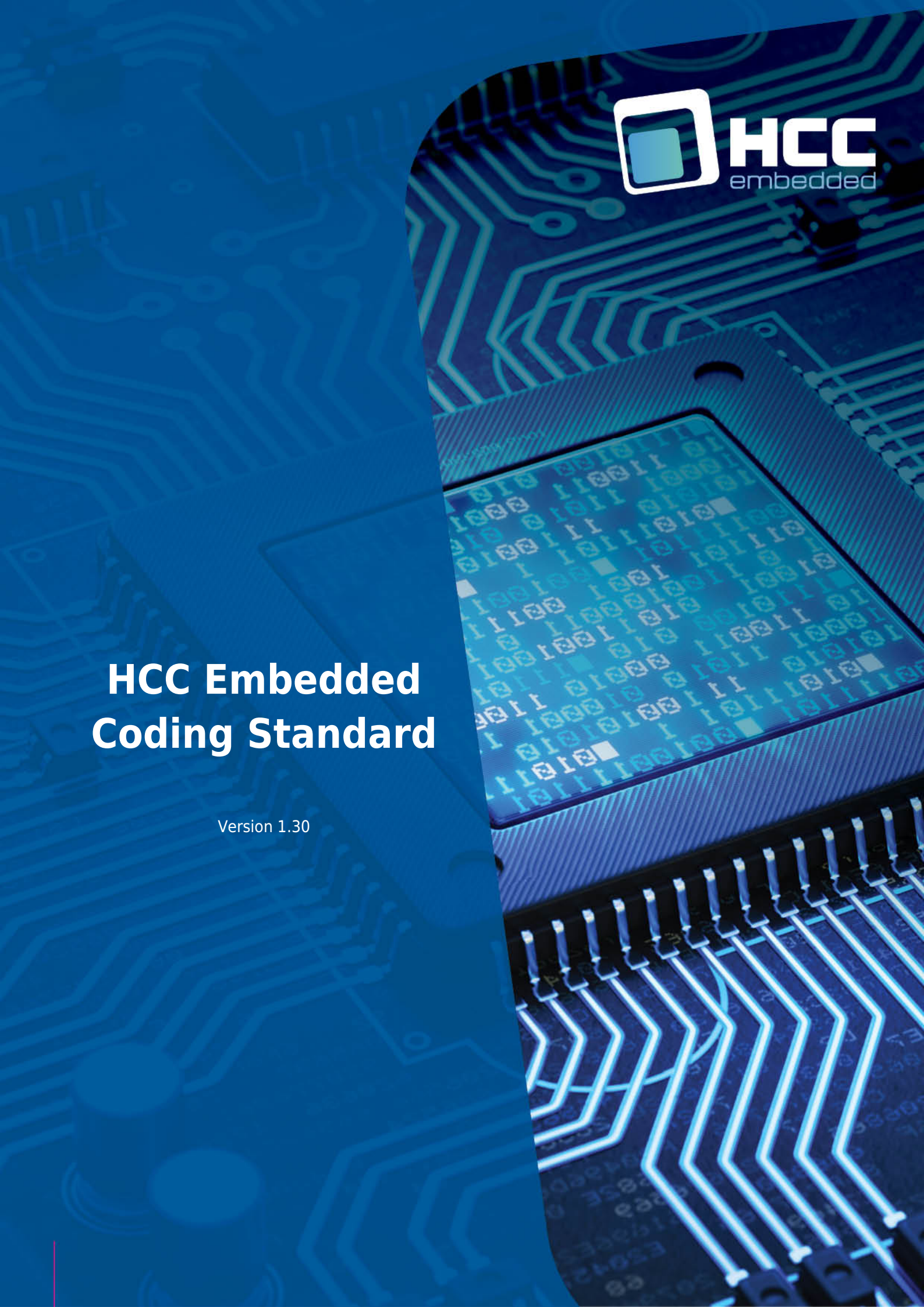

HCC Embedded Coding Standard

Version 1.30

Table of Contents

1. Glossary of Abbreviations

Abbreviation	Description
AEM	Advanced Embedded Middleware
API	Application Programming Interface
CEO	Chief Executive Officer
FSM	Functional Safety Management
HCC	HCC-Embedded Kft.
OAL	OS Abstraction Layer
PM	Project Manager
PSP	Platform Specific Package
QM	Quality Manager
QS-Plan	(Qualitatssicherung) Quality Assurance Plan
RSC	Reusable Software Component
SIL	Safety Integrity Level
SRS	Software Requirements Specification
SW	Software
TD	Technical Director
TN	TÜV NORD
VCS	Version Control System
WR	Work Result

2. Coding Style

This section gives coding guidelines.

2.1. Naming Conventions

General Guidelines

S1 Use meaningful names

Choose names carefully. A reader should be able to tell what a variable or function name represents by looking at its name. Here are some examples:

Variable Description	Good Names	Bad names
Initialize the flash array	flash_init	ini
Set layout of the flash array	flash_layout_t	layout

S2 Keep the name length reasonable

When choosing names that completely describe an object, do not create excessively long names.

Empirical studies have determined that variable names are ideally from eight to 20 characters in length. A short name works if its meaning is clear, but if a name has to be longer, make it longer. Procedure names are generally longer than variable names because of the actions they represent.

To maintain portability among different compilers, no name can be longer than 32 characters.

S3 Put qualifiers at the end of the name

Qualifiers are names like *sum, total, maximum,* and *minimum*. When you create a name that uses a qualifier, put it at the end of the name. This keeps the most important part of the variable, its meaning, in front. This also establishes a convention that helps to avoid confusion. Here are some standard qualifiers:

Suffix	Abbreviation	Description
Count	Cnt	Running count
Total	Tot	Total or running total
Size	Sz	Size of an object
Minimum	Min	Minimum
Maximum	Max	Maximum
Index	Idx	Index into an array or list

S4 Avoid use of confusing acronyms and abbreviations

Only use acronyms and abbreviations that are widely accepted. Examples of widely accepted acronyms are the protocols IP, USB, HTTP, and so on.

Abbreviations must be clear. For example, *pkt* generally means *packet*. Less clear, however, is *ttl;* does this mean *total, title,* or *time to live*?

We recommend writing a glossary as part of the program documentation, but do keep it up to date.

Modules and Files

S5 Keep file lengths reasonable

A file consists of sections that should be separated by blank lines.

Although there is no maximum length limit for source files, files with more than about 1000 lines are difficult to deal with.

It is good practice to organize files based on their functionality. Implement each item of functionality or feature in a different file; this keeps file lengths reasonable.

S6 Follow the file naming rules

File names comprise a base name, a period, and a suffix. The first character of the name should be a letter. All characters (except the period) should be lowercase letters, numbers, and underscores. No spaces are allowed within the file name. All file base names must be unique in the first eight characters; duplicate names are allowed only for source and header (**.c** and **.h**) files. Do not use the names of the standard library files (for example, **math** or **errno**).

Any module (file) containing a **main()** function must have the word “main” in its filename.

If the development tools allow you to select a filename suffix, the following suffixes are required:

- C source file names must end in **.c**.
- Assembler source file names must end in **.s** or **.asm**, if the compiler allows it. (There are compilers which have other restrictions.)
- Include header file names must end in **.h** (**.inc** is allowed in the Assembler case).

Sometimes the development environment does not allow usage of the filename suffixes listed above. In this case, the Development Specification document must contain the description of the naming convention and an explanation of the deviation.

Data Types

S7 Follow the data type naming rules

When you declare new data types, their names must consist of lower case characters and internal underscores. The name of these data types, including structures, unions and enums, shall begin with *t_*.

S8 Name all new structures, unions, and enumerations by using typedefs

Use *typedefs* to name all new structures, unions, and enumerations.

Functions and Procedures

Functions always return a value. *sin(x)* and *cos(x)* are typical examples of functions. Procedures, on the other hand, either do not return a value, or return only a failure or status indication. Choose names carefully, depending on whether you are dealing with a procedure or a function.

S9 Naming Rules

No function (or procedure) shall have a name that:

- Is a keyword of C or C++.
- Overlaps with a function name in the standard library (for example, **memset()**).
- Begins with an underscore (indicate static functions in another way).

A procedure name should only be as long as required to describe what it does. Note that names must be shorter than 32 characters (MISRA rule).

Function names must not contain any upper case letters. Use underscore characters (_) to separate words in names. Macro names must not contain any lower case characters.

S10 Function names must describe the returned value

Functions that return a meaningful value should describe what is returned. Examples of good function names are **weekday()**, **sector_addr()**, and **error_count()**.

Functions returning Boolean values should always have names that imply a true or false value. For example, use **ep_rx_ready()** instead of **tes_ep_rx_flag()**.

Some common prefixes can be used for function names. The following are typical examples and should be used for consistency:

Prefix	Description
get	Gets a value. This is generally used with handle types. For example **get_file_id(handle)**.
is	Use this prefix on Boolean variables to make the name clearer if necessary. For example, **is_available()** may be clearer than **available()**.

S11 Names of *void* functions should describe an action

Void functions in C are similar to procedures in other languages. Usually they return a success/failure indication rather than a meaningful value. Procedure names should describe the action that they perform. This will generally be a verb or command. Examples of clear procedure names are **combine_segments()**, **close()**, **send_alarm()**, and **rewind()**.

Do not use ambiguous and unclear names. A name must describe clearly what the procedure does. An example of a poor name is **process_input()**.

Use the following command words within procedure names to describe the type of action taken:

Command	Description
Set	Sets a value. An example is **SetEpNbr().**
Open	Opens a file or handle.
Close	Closes a file or handle.
Cancel	Cancels an operation. An example is **CancelTimer().**
Send	Sends an object somewhere.

Variables

S12 Naming rules

No variable shall have a name that:

- Is a C or C++ keyword.
- Overlaps with a variable name in the standard library (for example, *errno*).
- Begins with an underscore.
- Contains any uppercase letters.

In addition:

- No variable name shall be longer than 31 characters (MISRA rule).
- No variable name, including a loop counter, shall be shorter than three characters because of the possibility of global search (for example, *grep*). The name of a loop counter in a function can be shorter (for example "*i*") if the function does not use more than two loop counters, and the length of the function allows it.
- Use underscore character(s) to separate words in names.
- Names of global variables shall begin with the letter "*g*", for example *g_input_ready*.
- Prefix pointer variable names with a *p_*. Pointers to objects should begin with a '*p*' to indicate that they are pointers to the specified object. This convention is so widely used in the 'C' programming community that it is a de facto standard. A notable exception to this recommendation is any string variable declared as a '*char **'. This does not need to follow this rule, since the logical use of the variable is not as a 'pointer to character' but rather as a string. When you really want a pointer to a character, use the prefix *p_*.
- Global pointers shall begin with *gp_*.

S13 Variable names should describe an object

Variable names should describe the data they represent. They are generally nouns with possible qualifiers or adjectives. Examples of good variable names are *jitter_buffer* and *retry_count.* The only exception to this rule is in the naming of Boolean variables.

Avoid using unclear names. For example: *my_buffer* refers to a buffer, but it does not really describe how the buffer is used.

S14 Name temporary variables precisely

Do not use temporary variable names. Use descriptive names. Avoid general names such as *tmpbuf*; use a name that describes the object that the buffer holds, for example *command_buffer*.

S15 Boolean variable names

Although C does not have a specific Boolean type, most groups create a *typedef* for Boolean. It is unsafe to *define* a constant such as TRUE is equal to 1. It is safe (and conforms to MISRA rule 13.2) to compare the Boolean variable to 0. Use the following (and similar) structures for testing purposes: *if(!b_is_buffer_full).*

The name of a variable of Boolean type should begin with *b_*. The name must imply a true or false value; it should indicate the question it answers, for example *b_is_buffer_full* or *b_is_buffer_empty*.

Take into consideration the fact that C99 contains a Boolean type and a standard header file <**stdbool.h**>.

Defines, Macros and Constants

S16 Macros - Place parentheses around macro parameters

Place parentheses around macro parameters. This avoids unexpected precedence problems when the macro is expanded into code. For example, use this:

```
#define PRODUCT(a , b) =  ((a) * (b))
```

But do not use this:

```
#define PRODUCT(a , b) =  (a * b)
```

S17 Make #defined constants and macros all capitals with underscores between words

Using all capital letters alerts the reader that the name is a macro and special care must be taken in its use. Examples are *BUFFER_SIZE* and *RETRY_COUNT.*

S18 Name constants for what they represent, not their value

Do not name constants for their value, it is pointless. This is shown in this example:

```
#define SIXTY_SECS 60
/* get the number of minutes */

minutes = secs / SIXTY_SECS;
```

The following is better:

```
#define SECS_PER_MINUTE 60
/* get the number of minutes */

minutes = secs / SECS_PER_MINUTE;
```

2.2. Layout and Style

White Space and Blank Lines

S19 Separate blocks of code with blank lines or comments

A blank line between logical breaks in a long sequence of lines will help the reader understand the code.

S20 Surround operators with blanks

Blanks around operators make them easier to see. For example, use the following:

```
ep_dsc = cd_dsc[cd].ep_dsc + 1;
```

Instead of:

```
ep_dsc=cd_dsc[cd].ep_dsc+1;
```

S21 Insert a blank after commas

The following example is good:

```
func(a, b, c)
```

The following examples are bad:

```
func(a,b,c)
func(a, b ,c)
```

S22 Insert a space before and after pointer variants

Put a space before and after pointer variants (star, ampersand) in declarations, as shown below:

```
t_cd_dsc * cd_dsc;
```

Precede pointer variants in expressions with a space, but do not use a following space:

```
 *sc=1;
```

Multiple Statements per Line

S23 Avoid multiple statements per line

It’s easier to find syntax errors when the compiler gives you a line number, and it’s easier to step through the code using a debugger.

S24 Use comma separators correctly

Do not use the comma (**,**) separator within variable declarations. Place declarations one per line.

For example, the following is very misleading as C binds the asterisk with *buffer*, so *command* is declared as type *char*, not *char **:

```
char * buffer, command;
```

It is much clearer to write this:

```
char * buffer;
char * command;
```

It is also easier to comment each declaration if it is on a separate line.

Indentation and Braces

S25 Indent blocks

Programs should use a consistent number of spaces per tab stop. Usually three or four is used, but we use two. This is a good compromise between readability and excessive indentation.

S26 Make case labels in switches easy to identify

Case labels should stand out from the code. Here is a good example:

```
switch(flash_info.state)
{
    case FLASH_ST_PRG:
        do_something();
        break;

    case FLASH_ST_ERASE:
        do_something_else();
        break;

    default:
        error_message();
        break;
}
```

This example also shows the use of blank lines after the *break* statement. This helps each case statement to stand on its own.

S27 Use “Allman style” for braces

Use the so-called “Allman style”. This means that braces should surround blocks on their own line at the previous indent level. Braces should be on their own line to highlight the fact that they enclose a new block. Placing them at the previous indent level is common practice and is strongly recommended. Using a consistent technique is a good thing so we have adopted this format.

This example shows this:

```
if (a == b)
    {
    /* intended body of if statement */
    }
```

S28 Use braces with single statement blocks after a while, do-while, or if

The main reason for using braces in this context is for maintenance and clarity.

The following is an example of correct use:

```
while (!timer.done)
{
   // Even an empty statement should be surrounded by braces.
}
```

Lines

S29 Limit line widths

Limit the length of all lines in a program to a maximum of 80 characters, including the source code and possible inline comments. Keeping this rule eases onscreen side-by-side code differencing, so at least the source code part shall be limited to 80 characters.

S30 Break and indent long lines after a separator, comma, or binary operator

Keep line lengths reasonable. Almost everyone has the ability to display and print lines greater than 80 characters long. Even so, it is a good idea to try to keep lines below that length.

Break lines before a binary operator or after a comma or semicolon. Pay special attention to the pointer operator ->. Consider the pointer as part of the variable name.

Break lines as shown below:

```
segment = segment
        + offset;

while (function_1(argument1, argument2, argument3)
    != function_2())

for (buffer_index = buffer_start; buffer_index < buffer_end;
     buffer_index++)
```

Parentheses

S31 Parentheses in logical expressions

Unless it is a single identifier or constant, surround each operand of the logical && and || operators with parentheses. For example:

```
if ((a<b) && (b<c))
```

S32 Use parentheses to clarify expressions

Use parentheses to make clear the order in which operations are performed, even if the precedence rules of C don’t require them.

For example, the following is a valid expression, but probably does not do what you intended:

```
if (a & b == 0 && c == d)
```

Use parentheses within the expression as shown below:

```
if (((a & b) == 0) &&  (c == d))
```

Tabs

There is just one issue relating to tabs.

S33 Tab size

If tabs are used in source, they must be equivalent to two chars. Some editors allow you to set a different tab size; this is a bad idea. We use different IDEs, with different editors. Two character tab size is widely accepted so use this with every editor and every IDE to maintain consistency.

Every time you check in a source file, the tabs should be expanded to two spaces. There should be no tab in a versioned source file.

Use a C code beautifier such as 'indent' to format code

There is just one issue relating to beautifiers.

S34 “C” Beautifier

We use UniversalIndentGUI, a GNU tool. This uses either **astyle.exe** or **gc.exe** or **indent.exe**, maintaining a common GUI for the three. The three indenter programs are GNU publicly provided programs for formatting ‘C’ source and header files. UniversalIndentGUI uses one config file, **gc.cfg**, for all the formatter programs.

Example: indent.exe

This example shows recommendations for the config file of the **indent.exe** program:

```
-bli0 Indent braces 0 spaces.
-nbbo Do not prefer to break long lines before boolean operators
-cbi4 Indent braces after a case label 4 spaces
-cli4 Case label indent of 4 spaces
-di4 Put variables in column 4
-i4 Set indentation level to 4 spaces
-npcs Do not put space after the function in function calls
-npsl Put the type of a procedure on the same line as its name
-sc Put the '*' character at the left of comments
-ts4 Set tab size to 4 spaces
-bls Put braces on the line after "struct" declaration lines
-prs Put a space after every '(' and before every ')'.
```

The config file should be refined later. This file is also stored in the version control repository.

Conditional Formatting if using a beautifier

Sometimes formatting of certain areas of a ‘C’ source or header file is not desired. This is sometimes the case for data arrays. Often, formatting should not be applied to data arrays that require tabulation for easier reading. Special INDENT-OFF and INDENT-ON labels can be used to stop and resume formatting within a ‘C’ file.

To stop formatting on a code block, use:

```
/*  *INDENT-OFF*   */
```

To resume formatting, use:

```
/*  *INDENT-ON*   */
```

By using comments above, the configured style is not applied. Use of INDENT-OFF and INDENT-ON must include the ‘*’ on both ends as shown. This works with **indent.exe**.

3. Comments

3.1. General Usage

S35 Use English for all comments

S36 Never nest comments

S37 Never use comments to disable a block of code, even temporarily

There should be no commented code at all. That is, nothing like this:

```
x = x + 1;
/* printf("The x value is %d", x); */
```

Use *#ifdef* if it is absolutely necessary to leave, for example, debug code in place after the code has been debugged.

The *#ifdef* argument is a symbol (that is, do not use *#if* 0) that is documented in a comment at the beginning of the file, for **all** the files where it is used.

The following is allowed:

```
        x = x + 1;
#ifdef INCREMENT_TEST
       printf("The x value is %d", x);
#endif
```

if and only if the following (or a similar comment) appears at the head of the file:

```
/*
       Compilation Flags

       INCREMENT_TEST  enable the code to test the frobar incrementer.
                       Left here because the frobar implementation will
                       soon change and we need to keep the test.
*/
```

S38 Update comments

One of the perils of good comments is that over time they no longer reflect the truth of the code. Comment drift is intolerable. You must change the comments as you change the code. The two things happen in parallel. Never defer fixing comments until later, as it just will not happen. Better still: edit the descriptions first, and then fix the code.

S39 Single line comments

Single line comments in the C++ style (preceded by //) are not allowed. Though C99 allows us to use these, we want to keep the backward-compatibility with all the compilers that have been used. Most of these are C90-compliant and C90 does not allow C++ style comments.

S40 Comment your intent, not a solution

When you write a comment about a procedure, describe your intent, not how you are solving it. For example, here's a comment describing the purpose of the function:

```
 /*
When a fatal error occurs, this function is called to end with a standard
error message and a never end loop.
*/
void _dead_(void)
{
    printf("\nError: cannot continue!\n");
    while (1);
}
```

S41 Do not repeat code in comments

Repeating code in comments is redundant. It also makes the comment harder to maintain because you have double the work.

S42 Avoid abbreviations

Write comments clearly and precisely. Avoid abbreviations, except for those in common usage such as *etc.*, if you want your comments to be understood by everyone.

S43 Comment order dependencies

If it is unclear that one block of code must come after another one, add a comment indicating this:

```
 /* The output queue must exist before any services using it are started. */
init_output_queue();
init_eventlogger();
init_scheduler();
log_message('System Started');
```

In the above example, it would be unclear that **init_output_queue()** must be executed before **init_event_logger()** without a comment indicating this.

3.2. Inline Comments

There is just one issue relating to inline comments.

S44 Blocks of code and individual lines when it adds clarity

It is difficult to comment individual lines. Usually you just end up repeating the code. The exceptions to this are documenting data and variable declarations.

Explain the meaning and function of every variable declaration. Long variable names are merely an aid to understanding; accompany the descriptive name with a deep, meaningful, prose description. Comment members of a *struct* or *enum* similarly.

Annotate declarations with comments to the right, like this:

```
BUFFER trace_buffer;       /* buffer to hold the trace data */
uint16_t line_count;       /* Number of lines in trace buffer */
typedef struct
{
  uint16_t drivenum;            /* drive number 0-A 1-B 2-C    */
  char path [FN_MAXPATH];       /* pathname  /directory1/dir2/ */
  char filename[F_MAXNAME];     /* filename                    */
  char fileext [F_MAXEXT];      /* extension                   */
} F_NAME;
```

If a single line of code deserves its own comment, position this before the line of code at the same indent level as the line or block of code. For example:

```
/* Comment here */
lines of code...
```

3.3. Module (file) and function comments

Include a heading comment at the beginning of each file, explaining what the file is for.

Begin every module (source file) and function with a header in a standard format. The format should be consistent.

S45 General module templates

To encourage a uniform module look and feel, we created module templates named "**module_template.c**" and "**module_template.h**", stored in the source directory, which becomes part of the code base maintained by the VCS. Use one of these files as the base for all new modules. The module template includes the following:

- A standardized form for the header (the comment block preceding all code).
- A standard location for file includes and module-wide declarations (defines section), function prototypes and macros.
- The standard format for the comment block of functions.

The suggested order of sections for a program file is as follows:

1. First, in the source file, is a file prologue that tells what is in that file. A description of the purpose of the objects in the file (whether they be functions, external data declarations or definitions, or something else) is more useful than a list of the object names. The prologue may optionally contain author(s), revision control information, references, etc. An example of the file prologue adopted can be seen in the section on coding examples.
2. Any header file includes should be next. If the include is for a non-obvious reason, the reason should be commented. In most cases, system include files like **stdio.h** should be included before user include files. Use relative paths to define the include files position.
3. Any *defines* and *typedefs* that apply to the file as a whole are next. One normal order is to have "constant" macros first, then "function" macros, then *typedefs* and *enums*.
4. Next comes the function prototype section. The function prototypes are defined here, this section shall precede the external declaration section – a declaration can refer to a prototyped function.
5. Next come the global (external) data declarations, usually in the order: externs, non-static globals, static globals. If a set of defines applies to a particular piece of global data (such as a flags word), the defines should be immediately after the data declaration or embedded in structure declarations, indented to put the defines one level deeper than the first keyword of the declaration to which they apply. Generally, it is better to put these globals (externals) in the appropriate header file associated with the owner.
6. Next section contains the local variable declarations. Their scope is the file where they are declared. Use the “static” storage class identifier.
7. The functions come last, and should be in some sort of meaningful order. Similar functions should appear together. A "breadth-first" approach (functions on a similar level of abstraction together) is preferred over depth-first (functions defined as soon as possible before or after their calls). Considerable judgement is called for here. If defining large numbers of essentially-independent utility functions, consider alphabetical order.

Header example

```
/*****************************************************************
 * Module name: xxxxxxxxx
 *
 *            Copyright (c) 2003-2011 by HCC Embedded
 *
 * This software is copyrighted by and is the sole property of
 * HCC.  All rights, title, ownership, or other interests
 * in the software remain the property of HCC.  This
 * software may only be used in accordance with the corresponding
 * license agreement.  Any unauthorized use, duplication, transmission,
 * distribution, or disclosure of this software is expressly forbidden.
 *
 * This Copyright notice may not be removed or modified without prior
 * written consent of HCC.
 *
 * HCC reserves the right to modify this software without notice.
 *
 * HCC Embedded
 * Budapest 1133
 * Vaci ut 76
 * Hungary
 *
 * Tel:  +36 (1) 450 1302
 * Fax:  +36 (1) 450 1303
 * http: www.hcc-embedded.com
 * email: info@hcc-embedded.com
 *
 *
 *
 *****************************************************************/


/*****************************************************************
* Module name: xxxxxxxxx
*
*****************************************************************/

/*****************************************************************
* Module Description:
* (fill in a detailed description of the module's function here).
*
*****************************************************************/

/*****************************************************************
*
* Include section
* Add all #includes here
*
*****************************************************************/

/*****************************************************************
*
```

```
* TYPEDEFS
*
*****************************************************************************/

/*****************************************************************************
*  Defines section
* Add all #defines here, in the following order:
*
* DEFINITIONS AND MACROS
*
*****************************************************************************/

/*****************************************************************************
*
* STRUCTURES
*
*****************************************************************************/

/*****************************************************************************
*
* Function Prototype Section
* Add prototypes for all functions called by this
* module, with the exception of runtime routines.
*
*
*****************************************************************************/

/*****************************************************************************
*
* EXPORTED VARIABLES
*
*****************************************************************************/

/*****************************************************************************
*
* LOCAL VARIABLES
*
*****************************************************************************/

/*****************************************************************************
*
* Function Declaration Section
* Add declarations of the module's functions here
* Start every function with a header - header template is below
*
*****************************************************************************/
```

S46 Function header

Start every function with a header that describes what the routine does and how, and describes the return value and the parameters (arguments). The parameter description shall show if the parameter is an input or output parameter (order: in parameters, out parameters)

Example function header:

```
/*****************************************************************************
* Function name      : TYPE foo(TYPE arg1, TYPE arg2...)
*
*  returns           : return value description
*
*  inputs:
*      arg1          : description
*
*  outputs:
*      arg2          : description
*
*
* Description        : detailed description
* Notes              : restrictions, odd modes
*****************************************************************************/
```

4. Coding Rules

4.1. General Coding Rules

The C language itself contains unspecified, undefined, implementation-defined and locale-specific construct, as it is described in Annex G of the ISO standard [ref 2]. To maintain the high quality of our product we can not use all of the possible constructs of the C language. To achieve this goal we must use a well defined, restricted subset of the standard language.

These restrictions are done by the use of coding rules. A proper set of coding rules can be found in the MISRA guidelines [ref 1]. Some of the MISRA rules are not applicable to our environment so we removed them from the list. The resulting MISRA rules are listed in MISRA Rules.

There are other rules and constrains which are not addressed in the original MISRA documents so we put them in this Coding standard as Additional HCC Rules.

HCC developers shall follow all the rules in MISRA Rules that have the “Required” category and all the other rules described in Additional HCC Rules. If, for any reason, using a rule is impractical, the developer must follow the standard Deviation Procedure. MISRA rules with category “Advisory” should also be followed as far as is reasonably practical. In case of a deviation from these rules, a formal deviation procedure is not mandatory but highly recommended. In any case, the deviation must be well commented.

4.2. Deviation Procedure

The rules described in this document should only be broken in exceptional circumstances and with the agreement of the relevant code supervisor. All deviations must be commented.

4.3. MISRA Rules

No.	MISRA Rule	Category
	GROUP 1: ENVIRONMENT	
1.1	All code shall conform to ISO 9899:1990 Programming languages – C, amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.	Required
1.2	No reliance shall be placed on undefined or unspecified behavior.	Required
1.4	The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	Required
	GROUP 2: LANGUAGE EXTENSIONS	
2.1	Assembler language shall be encapsulated and isolated.	Required
2.2	Source code shall only use /* ... */ style comments.	Required
2.3	The character sequence /* shall not be used within a comment.	Required
2.4	Sections of code should not be commented out.	Required
	GROUP 3: DOCUMENTATION	
3.1	All usage of implementation-defined behavior shall be documented.	Required
3.2	The character set and the corresponding encoding shall be documented.	Required
3.3	The implementation of integer division in the chosen compiler should be determined, documented, and taken into account.	Advisory
3.4	All uses of the #pragma directive shall be documented and explained.	Required
3.6	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.	Required
	GROUP 4: CHARACTER SETS	
4.1	Only those escape sequences that are defined in the ISO C standard shall be used.	Required
4.2	Trigraphs shall not be used.	Required
	GROUP 5: IDENTIFIERS	
5.1	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.	Required
5.2	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	Required
5.3	A typedef name shall be a unique identifier.	Required
5.4	A tag name shall be a unique identifier.	Required
5.5	No object or function identifier with static storage duration should be reused.	Advisory

5.6	No identifier in one namespace should have the same spelling as an identifier in another namespace, with the exception of structure member and union member names.	Advisory
5.7	No identifier name should be reused.	Advisory
	GROUP 6: TYPES	
6.1	The plain *char* type shall be used only for the storage and use of character values.	Required
6.2	*signed* and *unsigned char* type shall be used only for the storage and use of numeric values.	Required
6.3	typedefs that indicate size and signedness should be used in place of the basic types.	Advisory
	GROUP 7: CONSTANTS	
7.1	Octal constants (other than zero) and octal escape sequences shall not be used.	Required
	GROUP 8: DECLARATIONS AND DEFINITIONS	
8.1	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	Required
8.2	Whenever an object or function is declared or defined, its type shall be explicitly stated.	Required
8.3	For each function parameter, the type given in the declaration and definition shall be identical and the return types shall also be identical.	Required
8.4	If objects or functions are declared more than once, their types shall be compatible.	Required
8.5	There shall be no definitions of objects or functions in a header file.	Required
8.6	Functions shall be declared at file scope.	Required
8.7	Objects shall be defined at block scope if they are only accessed from within a single function.	Required
8.8	An external object or function shall be declared in one and only one file.	Required
8.9	An identifier with external linkage shall have exactly one external definition.	Required
8.10	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.	Required
8.11	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.	Required
8.12	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.	Required
	GROUP 9: INITIALISATION	
9.1	All automatic variables shall have been assigned a value before being used.	Required
9.2	Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	Required

9.3	In an enumerator list, the “=” construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	Required
	GROUP 10:ARITHMETIC TYPE CONVERSIONS	
10.1	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a. it is not a conversion to a wider integer type of the same signedness, or b. the expression is complex, or c. the expression is not constant and is a function argument, or d. the expression is not constant and is a return expression.	Required
10.3	The value of a complex expression of integer type shall only be cast to a type that is not wider and of the same signedness as the underlying type of the expression.	Required
10.5	If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	Required
10.6	A U suffix shall be applied to all constants of unsigned type.	Required
	GROUP 11:POINTER TYPE CONVERSIONS	
11.1	Conversions shall not be performed between a pointer to a function and any type other than an integral type.	Required
11.2	Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type, or a pointer to void.	Required
11.3	A cast should not be performed between a pointer type and an integral type.	Advisory
11.4	A cast should not be performed between a pointer to object type and a different pointer to object type.	Advisory
11.5	11.5 A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.	Required
	GROUP 12: EXPRESSIONS	
12.1	Limited dependence should be placed on the C operator precedence rules in expressions.	Advisory
12.2	The value of an expression shall be the same under any order of evaluation that the standard permits.	Required
12.3	The sizeof operator shall not be used on expressions that contain side effects.	Required
12.4	The right-hand operand of a logical && or || operator shall not contain side effects.	Required
12.5	The operands of a logical && or || shall be primary expressions.	Required
12.6	The operands of logical operators (&&, ||, and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, ||, !, =, ==, !=, and ?:).	Advisory
12.7	Bitwise operators shall not be applied to operands whose underlying type is signed.	Required

12.8	The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.	Required
12.9	Trigraphs shall not be used. The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	Required
12.10	The comma operator shall not be used.	Required
12.11	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	Advisory
12.13	The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.	Advisory
	GROUP 13 : CONTROL STATEMENT EXPRESSIONS	
13.1	Assignment operators shall not be used in expressions that yield a Boolean value.	Required
13.2	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.	Advisory
13.5	The three expressions of a for statement shall be concerned only with loop control.	Required
13.6	Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.	Required
13.7	Boolean operations whose results are invariant shall not be permitted.	Required
	GROUP 14 : CONTROL FLOW	
14.1	There shall be no unreachable code.	Required
14.2	All non-null statements shall either have at least one side effect however executed, or cause control flow to change.	Required
14.3	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a whitespace character.	Required
14.4	The goto statement shall not be used.	Required
14.5	The continue statement shall not be used.	Required
14.6	For any iteration statement, there shall be at most one break statement used for loop termination.	Required
14.7	A function shall have a single point of exit at the end of the function.	Required
14.8	The statement forming the body of a switch, while, do ... while, or for statement shall be a compound statement.	Required
14.9	An if expression construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement or another if statement.	Required
14.10	All if ... else if constructs shall be terminated with an else clause.	Required
	GROUP 15 : SWITCH STATEMENTS	
15.0	The MISRA C switch syntax shall be used.	Required

15.1	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	Required
15.2	An unconditional break statement shall terminate every non-empty switch clause.	Required
15.3	The final clause of a switch statement shall be the default clause.	Required
15.4	A switch expression shall not represent a value that is effectively Boolean.	Required
15.5	15.5 Every switch statement shall have at least one case clause.	Required
	GROUP 16 : FUNCTIONS	
16.1	Functions shall not be defined with a variable number of arguments.	Required
16.2	Functions shall not call themselves, either directly or indirectly.	Required
16.3	Identifiers shall be given for all of the parameters in a function prototype declaration.	Required
16.4	The identifiers used in the declaration and definition of a function shall be identical.	Required
16.5	Functions with no parameters shall be declared and defined with the parameter list void.	Required
16.6	The number of arguments passed to a function shall match the number of parameters.	Required
16.7	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.	Advisory
16.8	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	Required
16.9	A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list which may be empty.	Required
16.10	If a function returns error information, then that error information shall be tested.	Required
	GROUP 17 : POINTERS AND ARRAYS	
17.1	Pointer arithmetic shall only be applied to pointers that address an array or array element.	Required
17.2	Pointer subtraction shall only be applied to pointers that address elements of the same array.	Required
17.3	>, >=, <, <= shall not be applied to pointer types except where they point to the same array.	Required
17.4	Array indexing shall be the only allowed form of pointer arithmetic.	Required
17.5	The declaration of objects should contain no more than two levels of pointer indirection.	Advisory
17.6	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.	Required
	GROUP 18 : STRUCTURES AND UNIONS	
18.1	All structure and union types shall be complete at the end of the translation unit.	Required

18.2	An object shall not be assigned to an overlapping object.	Required
18.3	An area of memory shall not be used for unrelated purposes.	Required
18.4	Unions shall not be used.	Required
	GROUP 19 : PREPROCESSING DIRECTIVES	
19.1	#include statements in a file should only be preceded by other preprocessor directives or comments.	Advisory
19.2	Non-standard characters should not occur in header file names in #include directives.	Advisory
19.3	The #include directive shall be followed by either a <filename> or "filename" sequence.	Required
19.4	C macros shall only expand to a braced initializer, a constant, a string literal, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.	Required
19.5	Macros shall not be #define'd or #undef 'd within a block.	Required
19.6	#undef shall not be used.	Required
19.7	A function should be used in preference to a function-like macro.	Advisory
19.8	A function-like macro shall not be invoked without all of its arguments.	Required
19.9	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	Required
19.10	In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.	Required
19.11	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	Required
19.12	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.	Required
19.13	The # and ## preprocessor operators should not be used.	Advisory
19.14	The defined preprocessor operator shall only be used in one of the two standard forms.	Required
19.15	Precautions shall be taken in order to prevent the contents of a header file being included twice.	Required
19.16	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	Required
19.17	All #else, #elif, and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	Required
	GROUP 20 : STANDARD LIBRARIES	
20.1	Reserved identifiers, macros, and functions in the standard library shall not be defined, redefined, or undefined.	Required
20.2	The names of Standard Library macros, objects, and functions shall not be reused.	Required

20.3	The validity of values passed to library functions shall be checked.	Required
20.4	Dynamic heap memory allocation shall not be used.	Required
20.5	The error indicator errno shall not be used.	Required
20.6	The macro offset of in the **stddef.h** library shall not be used.	Required
20.7	The setjmp macro and the longjmp function shall not be used.	Required
20.8	The signal handling facilities of **signal.h** shall not be used.	Required
20.9	The input/output library **stdio.h** shall not be used in production code.	Required
20.10	The functions atof, atoi, and atol from the library **stdlib.h** shall not be used.	Required
20.11	The functions abort, exit, getenv, and system from the library **stdlib.h** shall not be used.	Required
20.12	The time handling functions of **time.h** shall not be used.	Required
	GROUP 21 : RUNTIME FAILURES	
21.1	Minimization of runtime 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 runtime faults.	Required

Category changes

The category is changed in these rules:

- 2.4 – from “Advisory” to “Required”. To the best of our belief, commenting out big sections of source code is a dangerous thing. Use #if or #ifdef with appropriate comment to achieve this goal.

Rules removed from the original MISRA rules

The following rules are removed from the original MISRA rules:

Rule	Description
1.3	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the language/compilers/assemblers conform.

1.3 is not applicable. We do not use multiple languages but just C.

Rule	Description
1.5	Floating-point implementations should comply with a defined floating-point standard.
10.2	The value of an expression of floating type shall not be implicitly converted to a different underlying type if: a. it is not a conversion to a wider floating type, or b. the expression is complex, or c. the expression is a function argument, or d. the expression is a return expression.
10.4	The value of a complex expression of floating type shall only be cast to a floating type which is narrower or of the same size.
12.12	The underlying bit representations of floating-point values shall not be used.
13.3	Floating-point expressions shall not be tested for equality or inequality.
13.4	The controlling expression of a for statement shall not contain any objects of floating type.

1.5, 10.2, 10.4, 12.12, 13.3, 13.4 are not applicable – we do not use floating point arithmetic (C29).

Rule	Description
3.5	If it is being relied upon, the implementation-defined behavior and packing of bit fields shall be documented.
6.4	Bit fields shall only be defined to be of type unsigned int or signed int.
6.5	Bit fields of signed type shall be at least two bits long.

HCC rule C10 and C37 give a more disciplined way of using the bit fields.

4.4. Additional HCC Rules

This chapter provides HCC's own extensions to the MISRA rules. The category of the rules described in this chapter is “Required”; this means that if use of a structure against any of these rules is necessary, the developer must follow the deviation procedure.

C1 Use the highest compiler warning level

Code should be compiled at the highest warning level.

C6 Endianness

All code shall be written to be endianness-independent. This means that arrays that are received or transmitted should always be handled as *uint8_t[]* arrays. Macros for accessing these arrays are defined in **psp_endianness.h**.

C8 Avoid “magic” numbers

Constants must be defined either by using enumeration or as macros. Never put a constant directly within the code. For example:

```
/* set the retry timer */
set_retry_timer(5000)
```

Most readers would not understand why 5000 was used. These should be defined:

```
#define RETRY_TIME   5000   /* retry after 5000 msec */
set_retry_timer(RETRY_TIME);
```

Defining the constant makes maintenance easier. Since the value is defined in only one place, changing the constant results in only a single line change to the code.

Do not use literal constants as return codes. The preferred method is enumeration with meaningful names. For example:

```
typedef enum
{
  BLINKING_LED_SUCCESS = 0,
  BLINKING_LED_OAL_ERROR,
  BLINKING_LED_PARAMETER_ERROR,
  BLINKING_LED_PSP_ERROR,
  BLINKING_LED_ERROR
}t_blinking_led_status;
```

Use literal numbers when they are really needed, when they mean a number, for example array index. In this example the program checks the first character:

```
if (!filename[0])
{
}
```

C10 Avoid use of bit fields in structures

Do not use bit fields in structures. The layout of the bit fields depends on the compiler and the endianness of the particular microcontroller – the code cannot be portable.

C11 Number of parameters

Do not use more than four parameters in a function parameter list. If you need more than four parameters, use a pointer to a structure instead of it. If it is absolutely necessary to use more than four parameters, apply the standard deviation procedure.

C16 No `for' loops with empty body

There should be no *for* loops with an empty body. This is usually a sign of a loop conditional that is too complex. A *for* with an empty body hides its real purpose in the loop controlling clauses; it should probably be rewritten as *while*, to make its purpose a lot clearer.

For example, the statement:

```
for (p = list; p->next; p = p->next);
```

Should be written as:

```
      p = list;

      while (p->next)
        p = p->next;
```

Make it clear that the purpose of the loop is to advance the pointer (the body of the loop), until we do not have a successor (the loop control clause).

C18 Keyword "const"

The *const* keyword shall be used whenever possible, including:

- To declare variables that should not be changed after initialization,
- To define call-by-reference function parameters that should not be modified (for example, *char const * p_data*),
- To define fields in structs and unions that cannot be modified (such as in a struct overlay for memory-mapped I/O peripheral registers), and
- As a strongly typed alternative to *#define* for numerical constants.

The upside of using *const* as much as possible is compiler-enforced protection from unintended writes to data that should be read-only.

C 24 Use for(;;)

Use the *for*(;;) structure to program an endless loop. Using the *while*(1) structure results in a warning from most compilers.

C25 Use guards in every header file

Use the usual guard to avoid multiple includes. This example is in the **common.h** file:

```
#ifndef _COMMON_H_
#define _COMMON_H_
.
.
.
#endif                /* define _COMMON_H_ */
```

Use the following guard to allow the proper usage of a C++ compiler:

```
#ifdef __cplusplus
extern "C" {
#endif
.
.
.
#ifdef __cplusplus
}
#endif
```

This example shows the composed version:

```
#ifndef _COMMON_H_
#define _COMMON_H_
#
#ifdef __cplusplus
extern "C" {
#endif
.
.
.
#ifdef __cplusplus
}
#endif
#endif                /* define _COMMON_H_ */
```

C29 Do not use floating point arithmetic

Do not use floating point *typedefs*, constants, variables and arithmetic.

If it is absolutely necessary, apply the standard deviation procedure.

C34 Functions shall return a value

Functions shall return a value; the value is a meaningful error code. The type of the return value (function) is *uint16_t*. The layout of the return codes is described in the project document, particularly in the "Software Development Plan". The result of the function (for example a computed value) shall not be the return value but shall be an output parameter.

If the return value should be used as a result, apply the standard deviation procedure.

C37 Use bit fields only to store flags or other short-length data

Using bit fields is platform- and compiler-dependent. The alignment of the bit fields in the storage unit depends on the compiler. We can avoid the dependencies if using the bit fields only for storing flags and accessing them by their name.

It is a good engineering practice to declare the bit fields and their access methods (macros or functions) in the same header file, and read/set/reset bit fields are allowed only with the help of these methods.

If other usage is absolutely necessary, apply the standard deviation procedure.

C38 Bitwise shift – an addition to MISRA rules 12.7 and 12.8

In C the result of a bitwise shift operation is implementation-defined or undefined. This means that the behaviour of the compiler can depend on the type of the left operand. MISRA rules 12.7 and 12.8 clear this situation but you must take special care when using bitwise operations. You must check the manual of the particular compiler which describes the compiler behaviour in the event of a bitwise shift.

5. References

[ref1] MISRA-C:2004 Guidelines for the use of the C language in critical systems

[ref2] ISO/IEC 9899:1990, Programming Languages – C

[ref3] Coding Standard Tables, file: **Coding_Standard_tables.doc**

[ref4] Parameters of the Beautifier program, file: **gc_parameters.xls**

[ref5] Ensuring Code Integrity – see Code Integrity Warranty

[ref6] Coding Standard Rules Summary, file: **coding_standard_rules_summary.xls**

6. Version

Version 1.30