‘C’ Style Guide and Programming Guidelines

```
Filename : Peter_CStyleGuide.doc/pdf
Modified : October 23, 2003
Author   : P.K. van der Vlugt
e-mail   : peter@ckernel.nl
```

TABLE OF CONTENTS

1. INTRODUCTION 1

2. GENERAL MODULE TEMPLATE 2

2.1 HEADER FILE TEMPLATE 2
2.1.1 File Header 3
2.1.2 Version History 3
2.1.3 Compiler directives 4
2.1.4 MODULES USED 4
2.1.5 DEFINITIONS AND MACROS 4
2.1.6 TYPEDEFS AND STRUCTURES 5
2.1.7 EXPORTED VARIABLES 6
2.1.8 EXPORTED FUNCTIONS 6
2.1.9 EOF 7
2.2 SOURCE FILE TEMPLATE 8
2.2.1 General remarks 9
2.2.2 File Header 9
2.2.3 Version History 9
2.2.4 Compiler directives 9
2.2.5 MODULES USED 9
2.2.6 DEFINITIONS AND MACROS 9
2.2.7 TYPEDEFS AND STRUCTURES 9
2.2.8 PROTOTYPES OF LOCAL FUNCTIONS 9
2.2.9 EXPORTED VARIABLES 10
2.2.10 GLOBAL VARIABLES 10
2.2.11 EXPORTED FUNCTIONS 10
2.2.12 LOCAL FUNCTIONS 11
2.2.13 EOF 11

3. PROGRAMMING GUIDELINES 12

3.1 GENERAL RULES 12
3.2 VARIABLES AND FUNCTION PARAMETERS 13
3.2.1 Global Variables 14
3.2.2 Local Variables 14
3.2.3 Function Parameters 15
3.3 FUNCTIONS 15
3.3.1 Prototypes of local functions 15
3.3.2 Other functions 16
3.4 TYPE DEFINITIONS 16
3.5 DEFINITIONS AND MACRO'S 17
3.6 OPERATORS 17
3.7 STATEMENTS GENERAL 17
3.8 EXPRESSIONS 20
3.9 FUNCTION CALLS 21
3.10 COMMENTS 21

1. Introduction

This C style guide can be used as a guideline for writing readable and structured (embedded) C code in a clear and reproducible style.

By using this style the portability and reusability of the code is improved, maintenance reduced and therefore the reliability can be increased. Also, some of the style suggestions will help avoid the possible problems with the C language. But above all it will improve readability, which will positively affect the following areas in software development:

- Code generation. A standard reduces the probability of coding errors.
- Code reviews. A standard increases the efficiency of reviews and inspections.
- Quality. A standard increases the overall quality of the project/product.
- Maintenance. A standard increases the maintainability of the project/product.

Although you may disagree with decisions made in these guidelines, try to conform to this style when you start using it. This actually applies to any style you want to adopt. When uniformity is a goal in your personal our your team's style, you have to comply to a simple set of rules at all time. It is very important that everyone, including you and your team members can read and understand the produced code.

The document specifies the complete layout of a software module and all of it's subsections like where to put global variables, structure definitions, function prototype headers etc. A number of rules will be defined for setting up a common structure and style of the code and where needed with various examples. Many rules and suggestions can be found in these guidelines. Too many it seems, however this is intentional. It is better to work under too many constraints than too few. The final code will look better.

These guidelines are meant to be used on standard ISO C code (ISO 9899). Please try to stay within standard C as much as possible. If you need to write system-dependent code, try to modularize and isolate it as much as possible.

2. General Module Template

In this chapter the source (.c) and header (.h) files for the general module template will be explained. All new modules that are being created have to comply to the layout of these standard template files. If you create your own template files you can choose to change the so called File Header to suit your own requirements, however do not change the rest of the sections.

The order of the comment blocks (start of a *section*) in the template files is fixed and it is not allowed to change this order. The various blocks determine where certain parts of your code have to reside. Each part of the template files will be described in more detail in this chapter.

2.1 Header File Template

```
/*
                             *******************
******************************* C HEADER FILE *******************************
**                           *******************                           **
**                                                                         **
** project   : General Template                                            **
** filename  : TPL.H                                                       **
** version   : 1                                                           **
** date      : August 02, 2003                                             **
**                                                                         **
*****************************************************************************
**                                                                         **
** Copyright (c) 2003, P.K. van der Vlugt                                  **
** All rights reserved.                                                    **
**                                                                         **
*****************************************************************************

VERSION HISTORY:
----------------

Version     : 1
Date        : August 02, 2003
Revised by  : P.K. van der Vlugt
Description : Original version.

*/

#ifndef _TPL_INCLUDED
#define _TPL_INCLUDED

/****************************************************************************/
/**                                                                        **/
/**                     MODULES USED                                       **/
/**                                                                        **/
/****************************************************************************/

/****************************************************************************/
/**                                                                        **/
/**                     DEFINITIONS AND MACROS                             **/
/**                                                                        **/
/****************************************************************************/

/****************************************************************************/
/**                                                                        **/
/**                     TYPEDEFS AND STRUCTURES                            **/
/**                                                                        **/
/****************************************************************************/

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

#ifndef _TPL_C_SRC
#endif

/****************************************************************************/
/**                                                                        **/
/**                     EXPORTED FUNCTIONS                                 **/
/**                                                                        **/
/****************************************************************************/

#endif

/****************************************************************************/
/**                                                                        **/
/**                               EOF                                      **/
/**                                                                        **/
/****************************************************************************/
```

2.1.1 File Header

General information describing the file can be found in the File Header. The File Header should look exactly like the example above or like the one you have chosen. The following items give a description of the fields that must be present in the File Header.

- ***project***. A short description of the name of the project or product where the file/module has been created for. Examples: General Modules, Linux Driver, DSP Project, ISO Protocol.
- ***filename***. The filename in CAPITALS. It is not allowed to add drives and directories to the filename. The filename and the Project Management software (IDE, Version Control Management) determine where a file is stored.
- ***version***. The current (last) version number of the file. The version number is simply a sequential number that increases on every change session. This can be one simple change or a block of changes done during some time of maintenance. If you use a Version Control Database this version number should keep track with the version number in the VCS.
- ***date***. The date of change when the last version number was created.
- ***copyright***. A copyright notice for the file. Each year a version of the file has been created in should be put in the copyright notice. With a dash you can specify a range of years, with a comma you can specify individual years. For example 1997-1999, 2001 means that a version of the file has been created in 1997, 1998, 1999 and 2001. You must put your name or company's name following the copyright statement.

2.1.2 Version History

The Version History overview contains a block of comments describing each change in chronological order. The first block always shows the 'Original Version' of the module and should as such be described in the Description field. The last block of Version History always describes the last and current version of the file. Thus for each new version of the file a block has to be added at the end of the Version History list.
Important ! Not every small change results in a new version. It is possible to have a list of changes in the Description field belonging to one new version. The number of changes depend on the requirements for that particular version when the module is maintained. All changes to a certain version can be copied to the Version Control Database when the file is checked in. The following items give a description of the comments that must be present in the File Header.

- ***Version***. The version number for the described change(s).
- ***Date***. The date of change to this version.
- ***Revised by***. Responsible person for this version.
- ***Description***. When it is the first revision of a file, "*Original Version*" has to be put here, but additional general comments are allowed as well. In the following revisions the description must exactly reflect the changes done to the file. The description of a single change should be marked by an asterisk, as shown in the example. If the description is very large or special an exception can be made. However the most important rule is that the change has to be documented as clearly as possible.

An example of a revision:

```
Version     : 2
Date        : August 03, 2002
Revised by  : P.K. van der Vlugt
Description : * Changed input sampling time to 50 msec.
              * Added support for different CAN physical layers.
              * Added activation of the reference output.
```

2.1.3 Compiler directives

The template files contain specific compiler directives which are necessary to allow multiple inclusion of header files by the application modules and to allow inclusion of a header file by it's own C-source file. The latter is needed for the function prototyping of the exported functions of a module.

Each header file has the following compiler directives in the beginning:

```
#ifndef _..._INCLUDED
#define _..._INCLUDED
```

The module's name has to be filled in for the three dots in the example. The closing *#endif* can be found at the bottom of the header file just before the ***EOF*** block. By using this mechanism, a header file can be included multiple times without generating compiler errors, only the first time the compiler reaches an include of the header file, the header file will actually be included by the compiler.

Just below the block ***EXPORTED VARIABLES*** another compiler directive shows as follows:

```
#ifndef _..._C_SRC
#endif
```

The module's name has to be filled in for the three dots in the example. Between these two lines the exported variables of a module, if there are any, have to be declared using the keyword *extern*. In the C source file of the module this directive will be defined first before the include section and thus before the compiler includes it's own header file. When the header file is then included during compilation of the C source file, the compiler will skip the part under the ***EXPORTED VARIABLES*** section and thus compiler errors are prevented.

2.1.4 MODULES USED

Under the section ***MODULES USED*** all modules that are used or needed have to be included using the #include-statement. When standard C-header files (example *#include <stdio.h>*) are used then these will have to be included first. After this, separated by an empty line, the include section follows for the local include files (example *#include "can.h"*). It is recommended to use an alphabetical ordering on the modules included, so that it not only looks better, but also allows for quickly finding out if a specific module is used or not.

Example:

```
/****************************************************************************/
/**                                                                        **/
/**                      MODULES USED                                      **/
/**                                                                        **/
/****************************************************************************/

#include <stdio.h>

#include "can.h"
#include "delay.h"
#include "i2c.h"
#include "rtc.h"
```

2.1.5 DEFINITIONS AND MACROS

All definitions (#define) and macro's have to be placed under the ***DEFINITIONS AND MACROS*** section. This can be done in both the header and source file. In which file the definition or macro appears will depend on it's scope. Definitions or macro's which have to be known outside the modules (thus exported) will have to be put in the header file.
A definition is everything that is defined using a *#define*. A macro is a definition that uses arguments. To allow for a good recognition of defines and macro's it is recommended to use capitals completely. To increase readability in this case it is allowed to use underscores.
Logical groups of defines and macro' s have to be ordered together. All defines use the same indentation columns starting at the left margin. The different groups can be separated by using proper comment blocks and empty lines.

Example:

```
/***************************************************************************/
/**                                                                       **/
/**                       DEFINITIONS AND MACROS                          **/
/**                                                                       **/
/***************************************************************************/

/**** Key definitions ****/
#define FUNC_KEY_INP  !P5_0
#define PLUS_KEY_INP  !P5_1
#define MINUS_KEY_INP !P5_2
#define EQUAL_KEY_INP !P5_6

/**** Timing definitions ****/
#define HANDLER_TIME          (TICK_RATE / 1000)
#define MACHINE_HANDLER_TIME  MSEC_TO_TICKS (HANDLER_TIME)
#define ONE_SECOND_TIME       (MSEC_TO_TICKS (1000) / MACHINE_HANDLER_TIME)
```

2.1.6 TYPEDEFS AND STRUCTURES

A type definition can be placed in the header as well in the source file, depending on the scope of the definition. They have to be placed under the ***TYPEDEFS AND STRUCTURES*** section. A type definition defined in this section has to meet a number of rules. The name of the typedef starts with a capital and should end with '*Type*'. Furthermore the name has to be capitalized (each word starts with a capital, e.g. ConfigType) to improve readability. The use of underscores is not recommended.

There is however one exception to these rules. The basic types of *char, int, short, long, float* and *double* should not be used, but instead newly defined equivalents should be used that define specific storage lengths (According to the MISRA rule 13; see also www.misra.org.uk). This in order to avoid the problem that storage lengths of the basic types vary from compiler to compiler. It is simply safer to work with predefined storage lengths. It makes your code more independent from a compiler and/or processor platform. These new typedefs should preferably be put into a standard header which could be named e.g. stddefs.h, cextdefs.h or common.h. Choose one header file name for your style and use it in all projects and/or throughout your company.

Examples of such replacement typedefs and some additional useful extensions:

```
/***************************************************************************/
/**                                                                       **/
/**                       TYPEDEFS AND STRUCTURES                         **/
/**                                                                       **/
/***************************************************************************/

/**** C type extensions ****/
typedef unsigned char   bool;
typedef unsigned char   int8u;
typedef          char   int8s;
typedef unsigned int    int16u;
typedef          int    int16s;
typedef unsigned long   int32u;
typedef          long   int32s;
typedef          float  float32;
typedef          double float64;

/**** C pointer extensions ****/
typedef void*           pointer;
typedef int8s*          string;
typedef void            (*handlerptr) (void);
```

A type definition always starts at the left margin. Fields in a type definition structure start with lowercase letters, just as with local variables.

Values in an enum typedef have to be defined in capitals so that they look just the same as a *#define* definition.

Examples of some typedefs:

```
/****************************************************************************/
/**                                                                        **/
/**                      TYPEDEFS AND STRUCTURES                           **/
/**                                                                        **/
/****************************************************************************/

typedef struct
{
  int16u checkSum;
  int16u pulses;
  int8u  distance;
  bool   absInstalled;
  int16u airbagThreshold;
}
ConfigType;


typedef enum
{
  DUTCH,
  ENGLISH,
  GERMAN,
  FRENCH,
  NR_OF_LANGUAGES
}
Languages;
```

2.1.7 EXPORTED VARIABLES

In this section all exported variables from the module have to be declared between the compiler directives using the keyword *extern*. See paragraph *Compiler Directives* for more details on the meaning of the necessary directives here. The declared variables in the exported section have to be declared in the C source file as global variables under the same section and without the *static* keyword in order to export it's scope to the outside world.

Example:

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

#ifndef _KEYCODE_C_SRC

extern const KeyCodeType KeyTable [];
extern int8u             NrOfKeyCodes;

#endif
```

2.1.8 EXPORTED FUNCTIONS

In this section the exported functions of the modules are placed with their function prototypes. For each exported function a comment block has to be added below the function prototype that gives a good description of the meaning and use of the functions and it's arguments.

Example:

```
/****************************************************************************/
/**                                                                        **/
/**                      EXPORTED FUNCTIONS                                **/
/**                                                                        **/
/****************************************************************************/

/****************************************************************************/
int16s CK_Init (void);
/****************************************************************************/
/*
 * Initializes the C kernel. This call must be called before any other call
 * is used to the C Kernel. All resources are initialized, kernel timing is
 * initialized and the Idle task is created with priority 0 (lowest).
 * CK_Init turns off all interrupts. The application should not enable
 * interrupts before CK_Run is called.
 *
 * Parameters:  None
 *
 * Returns:     CK_OK                 Success
 *              CK_NO_QUEUE           Failed to allocate Queue resources
 *              CK_NO_TCB             Failed to allocate Idle Task resource
 */
```

2.1.9 EOF

This is the End of File marker section. Nothing should appear below this block. It is always the last block in both the header and the C source file.

2.2 Source File Template

```
/*
                             *******************
******************************* C SOURCE FILE *******************************
**                           *******************                           **
**                                                                         **
** project  : General Template                                             **
** filename : TPL.C                                                        **
** version  : 1                                                            **
** date     : August 02, 2003                                              **
**                                                                         **
*****************************************************************************
**                                                                         **
** Copyright (c) 2003, P.K. van der Vlugt                                  **
** All rights reserved.                                                    **
**                                                                         **
*****************************************************************************

VERSION HISTORY:
----------------

Version     : 1
Date        : August 02, 2003
Revised by  : P.K. van der Vlugt
Description : Original version.

*/

#define _TPL_C_SRC

/****************************************************************************/
/**                                                                        **/
/**                     MODULES USED                                       **/
/**                                                                        **/
/****************************************************************************/

/****************************************************************************/
/**                                                                        **/
/**                     DEFINITIONS AND MACROS                             **/
/**                                                                        **/
/****************************************************************************/

/****************************************************************************/
/**                                                                        **/
/**                     TYPEDEFS AND STRUCTURES                            **/
/**                                                                        **/
/****************************************************************************/

/****************************************************************************/
/**                                                                        **/
/**                     PROTOTYPES OF LOCAL FUNCTIONS                      **/
/**                                                                        **/
/****************************************************************************/

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

/****************************************************************************/
/**                                                                        **/
/**                     GLOBAL VARIABLES                                   **/
/**                                                                        **/
/****************************************************************************/

/****************************************************************************/
/**                                                                        **/
/**                     EXPORTED FUNCTIONS                                 **/
/**                                                                        **/
/****************************************************************************/

/****************************************************************************/
/**                                                                        **/
/**                     LOCAL FUNCTIONS                                    **/
/**                                                                        **/
/****************************************************************************/
```

```
/****************************************************************************/
/**                                                                        **/
/**                                 EOF                                    **/
/**                                                                        **/
/****************************************************************************/
```

2.2.1 General remarks

Most of the sections in the source file are the same as already described in the *Header File Template* paragraph. Extra sections are:

- ***PROTOTYPES OF LOCAL FUNCTIONS***,
- ***GLOBAL VARIABLES***
- ***LOCAL FUNCTIONS***.

2.2.2 File Header

General information describing the source file can be found in the File Header. The File Header should look exactly like the example above. Furthermore refer to the *Header File Template* paragraph for a detailed description of this section.

2.2.3 Version History

Refer to the *Header File Template* paragraph for a detailed description on this section.

2.2.4 Compiler directives

Just below the File Header and Version History a compiler definition shows as follows:

```
#define _..._C_SRC
```

The module's name has to be filled in for the three dots in the example. This definition will be defined first before the include section and thus before the compiler includes it's own header file. When the header file is then included during compilation of the C source file, the compiler will skip the part under the ***EXPORTED VARIABLES*** section in the header file and thus compiler errors are prevented.

2.2.5 MODULES USED

Refer to the *Header File Template* paragraph for a detailed description on this section.

2.2.6 DEFINITIONS AND MACROS

Refer to the *Header File Template* paragraph for a detailed description on this section.

2.2.7 TYPEDEFS AND STRUCTURES

Refer to the *Header File Template* paragraph for a detailed description on this section.

2.2.8 PROTOTYPES OF LOCAL FUNCTIONS

In this section the prototypes of the local functions of the source file are placed. All local functions and their prototypes have the keyword *static* in order to keep the scope 'inside' (not exported) to the source file. Note however that the functions are globally accessible from within the source file.

Example:

```
/****************************************************************************/
/**                                                                        **/
/**                        PROTOTYPES OF LOCAL FUNCTIONS                   **/
/**                                                                        **/
/****************************************************************************/

static void   kernelTickHandler (void);
static void   reschedule        (void);
static void   ready             (int8u tid, int8u action);
```

2.2.9 EXPORTED VARIABLES

In this section all exported variables from the module have to be declared ***without*** using the keywords *static* or *extern*. By omitting the *static* keyword the variable is exported or public.

Example:

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

int8u ActiveTask;
int8u OldTask;
```

2.2.10 GLOBAL VARIABLES

Static declared global variables must be placed under the section ***GLOBAL VARIABLES*** in the source file. The variable names should be placed one below the other, preferably left aligned in the same column and by preference on the same indent position. The declaration should start at the left margin. The scope of these variables is global to the source file and are not known to the outside world due to the *static* keyword.

Example:

```
/****************************************************************************/
/**                                                                        **/
/**                        GLOBAL VARIABLES                                **/
/**                                                                        **/
/****************************************************************************/

/**** IdleTask stack ****/
static stacktype IdleStack[IDLE_STACK_SIZE];

/**** Kernel management variables ****/
static int16u RRcount;                  /* Round-robin value */
static int16u PreemptionCount;
static int8u  ReadyQ;
static int8u  SleepQ;
static bool   ISRswitch;                /* Interrupt switch indication */

/**** Clock variables ****/
static int32u SystemTime;
static int16u SecondCounter;
```

2.2.11 EXPORTED FUNCTIONS

In this section the implementations of the exported functions of the module are placed.

Example:

```
/****************************************************************************/
/**                                                                        **/
/**                        EXPORTED FUNCTIONS                              **/
/**                                                                        **/
/****************************************************************************/
```

```
/****************************************************************************/
bool TimerInit (void)
/****************************************************************************/
{
int8u i;

  for (i = 0; i < NR_OF_TIMERS; i++)
  {
   ...
  }
}
```

2.2.12 LOCAL FUNCTIONS

The function declaration and the implementation of a local function must be placed under this section in the source file. All functions appearing in this section must have a corresponding function prototype in the section ***PROTOTYPES OF LOCAL FUNCTIONS***.

Example:

```
/****************************************************************************/
/**                                                                        **/
/**                     LOCAL FUNCTIONS                                    **/
/**                                                                        **/
/****************************************************************************/

/****************************************************************************/
static void kernelTickHandler (void)
/****************************************************************************/
{
  ...
  ...
}

/****************************************************************************/
static void reschedule (void)
/****************************************************************************/
{
  ...
  ...
}

/****************************************************************************/
static void ready (int8u tid, int8u action)
/****************************************************************************/
{
  ...
  ...
}
```

2.2.13 EOF

This is the End of File marker section. Nothing should appear below this block. It is always the last block in both the header and the C source file.

3. Programming Guidelines

An important rule is that the C code has to be written using English terms. This will improve readability since a lot of terms and definitions in the C language are already in English, as well as most specifications for projects or product developments. It is however allowed to use (language-) specific terms or definitions when they are commonly used in a project or product specification, just to keep the reference to it clear.

To define where specific parts of code has to be placed in a module a reference will be made to the predefined comment blocks or sections that are specified in the templates (see Chapter 2). A reference to these sections will be shown in capitals and italics throughout this document. Source code must always be inserted below one of those comment blocks. An exception has to be made for the ***EOF*** block which will always be the last block in a module.

3.1 General rules

The following rules are general rules that apply to all written C code:

- Each source file (.c) and header file (.h) must contain a File Header, a Version History and Comment Blocks as shown in the template files in the previous chapter of this document. The Comment Blocks are used to separate specific sections of code. An exception to this rule can be made for standard files needed from a compiler development environment

- The readability of names of functions, variables and functions parameters must be supported by the use of capitalizing. This means that 'words' in a name always start with a capital and continue with lower case. The first letter of a name is used to indicate the scope (local, global) of the particular function or variable. Example: thisIsAnExample.

- It is preferred that the line width does not exceed the width of the Comment Blocks in order to keep the code readable on screen and on the printer. In some cases however it is difficult to avoid.

- The additional C type definitions in the standard header file as described under paragraph 2.1.6, ***TYPEDEFS AND STRUCTURES,*** have the preference over the standard C types, i.e. type ***int8u*** should be used preferably over the C type ***unsigned char***. The reason for this is that type ***int8u*** will always be an unsigned 8-bit type independent of hardware or compiler.

- Turn off compiler optimizations. In general they produce more errors due to compiler bugs than benefits. Until proven otherwise.

- Don' t use floating-point variables where discrete values are needed. Using a float for a loop counter is a great way to run onto problems. Always test floating-point numbers as <= or >=, never use an exact comparison (== or !=).

- When testing incrementing or decrementing counters try to use the >= or <= instead of == in if-statements. This is one of the important defensive programming rules. If for some reason the value of the counter gets higher or lower than expected in the test for equal you would run into problems.

- Never rely on your 'memorized' precedence table for C. By using full parenthesizing every expression, the question of when to and when not to do it is eliminated. It also helps in documenting the original intent on expressions.

- Use correct type casting in calculations. Make sure the calculation is expressed with all variables in the same type. Do not rely on the compiler's interpretation and automatic casting otherwise you will b e surprised by errors compilers sometimes generate.

3.2 Variables and Function Parameters

There are a number of rules for the use of variables and function parameters:

- Global variables should to be declared at the left margin of the document.
- Local variables in functions should be declared under the preceding curly brace (‘{‘) and thus also at the left margin.

Example:

```
{
static int8u index;

int16s  i;

  ...
}
```

- When a variable or function parameter is declared as a pointer or multiple pointer, then the variable name has to reflect the number of pointer indirections. For each pointer indirection “Ptr” has to be added at the end of the name of the variable.

Examples:

```
static int8u   *ABytePtr;
static int16u **AWordPtrPtr;
```

- When the type of a variable or function parameter ends with “Ptr” then also the name of that variable or parameter has to end with “Ptr”. The same rule applies when the type of the variables implies a pointer indirection (e.g. pointer).

Examples:

```
static handlerptr TimerHandlerPtr;
static handlerptr *TimerHandlerPtrPtr;
static pointer    memPtr;
```

- Variables and parameter of the type ***string*** must end with “Str”.

Example:

```
static string SignOnStr = "Hello World V1.12";
```

- The asterisk (‘*’) in a pointer declaration sticks to the variable name or parameter name.

Example:

```
static float32 *RealValuePtr;
```

- An array declaration looks like the following example:

```
int8s Screen[NR_OF_ROWS][NR_OF_COLS];
```

It is preferred not to use spaces between the variable name and the first ‘[’ in order to allow Code Completion functionality in editors like CodeWright to function properly. The index value or definitions must stick to the ‘[’ and ‘]’ characters without using spaces.

An array declaration that has an initialization looks preferably as follows:

```
static int16u BaudrateTable[NUMBER_OF_BAUDRATES] =
{300, 600, 1200, 4800, 9600, 19200};

or:
```

```
static int16u BaudrateTable[NUMBER_OF_BAUDRATES] =
{
  300,
  600,
  1200,
  4800,
  9600,
  19200
};
```

3.2.1 Global Variables

Basically there are two types of global variables. The ones that are *static* variables have a global scope in it's own module, the variables that are not *static* are global to the application. A global variable must always start with a capital.

Static declared variables are placed under the section ***GLOBAL VARIABLES*** in the source file. The variables are placed left aligned preferably in the same column. The declaration must always start at the left margin.

Examples:

```
static int8u        Speed = 0;
static int8u        Acceleration = 0;
static MachineType *MachinePtr;
static int16s       SpeedArray[NR_OF_SPEEDS];
```

Variables that are declared without the static keyword have to be placed under the section ***EXPORTED VARIABLES*** in the source file. These variables also have to be declared *extern* in the header file under the same section between the two special compiler directives.

Example:

```
#ifndef _MODULE_C SRC

extern int8u   VehicleSpeed;
extern int16u  PulseCount;
extern float32 TrailerWeight[NR_OF_TRAILERS];

#endif
```

3.2.2 Local Variables

A local variable always starts with a lowercase letter and must be placed directly under the opening curly brace ('{') of a function at the left margin and thus not indented. The local variables are declared preferably line by line below each other and indented in the same column. Local variables that are *static* are preferably declared first. Static local variables have a local scope to the function, but the content of the variable is not lost outside the function.

Due to the fact that local variables start with a lowercase letter and global variables with a capital, there is a good distinction possible within the module. The name of a variable gives an indication of the scope of that variable, whether global in the module or local in a function.

Examples of local variables:

```
{
bool   quit = FALSE;
int8s  ch;
int16u pinCode1, pinCode2;
```

or:

```
{
```

```
static int8u canId;

int16u nrOfRetries = 0;
int16s sts;
```

3.2.3 Function Parameters

The variable name of a function parameter starts with a lowercase letter just as local variables in the function. In C, a function parameter has the same scope as a local variable. The declaration of a function looks like follows:

```
void SendCanCmd (int8u cmd, int8u *dataPtr);
```

The type of the first parameter 'sticks' to the opening bracket of the function parameter list, so no spaces in between. The same rule applies of course to the closing bracket, following the last parameter directly without spaces. The comma used to separate the parameters in the parameter list, have to be followed by a space. When the function parameter list exceed the maximum line length it is also allowed to use the next line(s) preferably in an ordered way like:

```
void SendCanCmdRsp (int8u  cmd,
                    int8u *cmdDataPtr,
                    int8u *cmdRspPtr);
```

When a function does not have any parameters the keyword ' void' must always be placed between the brackets.

3.3 Functions

There are two types of function declarations; *prototypes* of functions and the actual function implementations. Both can be further subdivided in *exported functions* and *local functions*. So there are four groups of function declarations, the groups can likewise be found in the module's file templates:

- Prototypes of exported functions (in the header (.h) file of a module). ***EXPORTED FUNCTIONS.***
- Prototypes of local functions (in the implementation (.c) file of a module). ***PROTOTYPES OF LOCAL FUNCTIONS.***
- Implementations of exported functions (in the implementation (.c) file of a module). ***EXPORTED FUNCTIONS.***
- Implementations of local functions (in the implementation (.c) file of a module). ***LOCAL FUNCTIONS.***

The naming convention of functions uses the same way of capitalizing as with variables. Names of functions that are local to the module have to start with a lowercase letter and names of functions that are exported have to start with a capital. In the next paragraphs the style of the various possibilities are discussed.

3.3.1 Prototypes of local functions

A prototype declaration of a function is needed in order to instruct the C-compiler how a specific functions looks like (thus is prototyped), so that the calling convention/usage of that function has been clearly defined. Therefore it is needed that the prototype definitions reside in the upper part of the file, before the sections where it can be used in a piece of code. That is the reason why ***PROTOTYPES OF LOCAL FUNCTIONS*** comes as one of the first sections. A block of prototype definitions looks for example like:

```
static void    calcLitersLeft  (void);
static float32 calcVolume      (int16u diameter, int16u length);
static bool    editLength      (int8s ch, int16u *valPtr);
```

A function prototype always starts at the left margin. The opening brace of the function parameter list is always separated at least one space from the function name, or like in the example, nicely ordered at the same indent position as the other function parameter lists.

3.3.2 Other functions

The other three functions declarations are being supported by lines of ‘comment asterisks’, so that it is clear where the function declaration starts. The declaration always starts at the left margin. The *return type* is preceded and followed by only one space. When the return type is a pointer, the pointer asterisk sticks to the function name just as with pointer variables. The following example shows a piece of code of a local function:

```
/******************************************************************************/
static float32 calcVolume (int16u diameter, int16u length)
/******************************************************************************/
{
int8u   temp;
float32 volume;

  if (diameter < MAX_DIAMETER)
  {
    ...
  }
  ...

  return (volume);
}
```

If there is an optional comment block added for the function implementation then this has to be placed preferably following the function declaration and before the opening curly brace ({):

```
/******************************************************************************/
static float32 calcVolume (int16u diameter, int16u length)
/******************************************************************************/
/*
 * This function calculates the volume in liters for one tank hose.
 */
{
}
```

There should always be one empty line before a new function declaration starts in order to separate the functions properly. In the header file (.h) it is mandatory to add explaining comments to get a good description of the functionality and behavior of the functions and the module itself.

3.4 Type definitions

A *type definition* can both be placed in the header file as well as the source file depending on the scope of the definition. The definitions have to be placed under the section ***TYPEDEFS AND STRUCTURES***. A type definition has to comply to a number of rules:

- The name of the type always starts with a capital and end with *Type*. An exception to this rule is the *enum* type.
- To make the type name readable it must be capitalized, just like with variables and functions.
- The definitions must be placed at the left margin.
- The fields in a type definition start with a lower case, since they are ‘local’ to the type.

Examples:

```
typedef struct
{
  int16u checkSum;
  int16u pulses;
  int8u  distance;
  bool   absInstalled;
  int16u airbagThreshold;
}
ConfigType;

typedef enum
{
  DUTCH,
  ENGLISH,
  GERMAN,
  FRENCH,
  NR_OF_LANGUAGES
}
Languages;
```

3.5 Definitions and macro’s

A *#define* can both be placed in the header file as well as the source file depending on the scope of the definition. The definitions have to be placed under the section ***DEFINITIONS AND MACROS***. Definitions has to comply to a number of rules:

- The definition must be in full capitals. For readability use underscores to separate words.
- Grouped definitions have to be ordered correctly and placed in the same positions when indenting.
- When a definition is more than just a number or a single name, it is advised to add brackets around the expression to make sure the substitution of the definition during compilation is not done in an unexpected way.

Examples:

```
/**** Key definitions ****/
#define FUNC_KEY_INP  !P5_0
#define PLUS_KEY_INP  !P5_1
#define MINUS_KEY_INP !P5_2
#define EQUAL_KEY_INP !P5_6

/**** Timing definitions ****/
#define HANDLER_TIME          (TICK_RATE / 1000)
#define MACHINE_HANDLER_TIME  MSEC_TO_TICKS (HANDLER_TIME)
#define ONE_SECOND_TIME       (MSEC_TO_TICKS (1000) / MACHINE_HANDLER_TIME)
```

3.6 Operators

Some generals rules for the usage of operators are the following:

- A comma is never preceded by a space, but is always followed by a space.
- A closing bracket is never preceded by a space.
- An opening bracket is always preceded by a space or another opening bracket.
- Array brackets stick to the variable name:

```
LanguageTextType Texts[NR_OF_LANGUAGES][NR_OF_TEXTS];
```

- Binary operators are always preceded and followed by a space:

```
        if ((OutputDevice == OUTPUT_SERIAL) && (val != ‘\n’))
```

- So called unary operators always ‘stick’ to the variable or expression. Be careful with some unary operators like the ‘~’, the default C type *int* is returned, even when applied to a char (int8s) or an unsigned char (int8u).

```
val++;
!isChar (ch);
~(int8u) val;
```

There are some exceptions to these rules:

- A structure pointer operator ‘->‘ sticks to both objects where it is applied on. Examples: `TextPtr->signOn`.
- The same rule applies to the structure member operator ‘.’. Example: `Config.checkSum`.
- A type cast operator is placed one space before the object the cast operates on. Example: `(int8u) intVar`.

3.7 Statements general

An important general rule for the C coding style is that the statement body of an if-statement, a for-statement, a while-statement or a label-statement always starts on the following line indented with two spaces. **Never use tabs** because different editors and/or different programmers may use different tab sizes causing different indentation levels in one file. The only way to avoid this problem is to use spaces when indenting. Professional editors like CodeWright fully support this auto indent with spaces feature, as well as auto styling C-code and chroma coding. This all improves readability on the screen and as such these type of editors are preferred and recommended.

For the following statements it is mandatory to indent:

• selection statement:	if, else, switch
• iteration statement:	do while, while, for
• label statement:	case, default

The beginning of a compound statement ({) is always placed on the next line in the same column as the preceding statement. The closing brace of the compound statement is again placed in that same column on a separate line. The compound statement braces should be the only coding-characters on that line.

Example:

```
if (...)
{
  ...
}
```

In general only one statement per line is used. However one can think of some exceptions to this rule. For certain statements that can be grouped it is allowed to put them on the same line, if they fit on that line.

Examples:

```
xPos++; Ypos++;
LcdSetCursorPosition (0, 3); printf ("%3d", val);
```

According to the general rule an if-statement and an else-statement have to be on separate lines. However also here it is allowed for certain simple groups of statements to put them on one line and group them below each other like:

```
if (..) ...; else (..);
if (..) ...;
if (..) ...; else (..);
if (..) ...; else (..);
```

The *switch* selection statement can be coded using the same rules and exceptions as described above. The normal preferred style:

```
switch (state)
{
  case CASE_NR_1:
    ...;
  break;

  case CASE_NR_2:
    ...;
  break;

  case CASE_NR_3:
    ...;
  break;

  default:
    ...;
  break;
}
```

The case- and break-statements are considered to have the same rules as for the compound statement. Likewise the *case* and *break* must reside in the same column and the statements in between are indented in the normal way.

The alternative style for the switch-statement can be used with short statements fitting on one line:

```
switch (state)
{
  case CASE_NR_1: ...; break;
  case CASE_NR_2: ...; break;
  case CASE_NR_3: ...; break;
  default:        ...; break;
}
```

One of the most frequently used iteration-statements, the for-statement, looks generally like:

```
for (i = 0; i < MAX_VAL; i++)
{
  ...;
}
```

When the expressions become too long to fit on one line, it is allowed to split up the for-statement into more lines:

```
for (i = MIN_SCALE_ADJUSTABLE_VALUE;
     i < MAX_SCALE_ADJUSTABLE_VALUE;
     i++)
{
  ...;
}
```

Or maybe even better, use the while-statement since that will fit better:

```
i = MIN_SCALE_ADJUSTABLE_VALUE;
while (i < MAX_SCALE_ADJUSTABLE_VALUE)
{
  ...;
  i++;
}
```

Write infinite loops as:

```
for (;;)
{
  ..
}
```

Or better , use a standard definition FOREVER and put it in the same header file as the C type extensions are in.

Finally an example of the general coding style of various C-statements within a function. This style is fully supported by the automatic styling feature of the CodeWright editor as well as other editors and is in most cases even customizable to this style. Therefore it is recommended to use this feature to ensure a high level of auto generated general coding.

```
/**************************************************************/
static void function (void);
/**************************************************************/
{
int8u i;

  if (..)
  {
    ...
    if (..)
    {
      for (i = 0; i < NR_OF_COUNTS; i++)
      {
        ...
      }
    }
    ...
    while (..)
    {
      ...
    }
  }
  else
  {
    ...
    do
    {
      ...
    }
    while (..);

    ...
    if (..)
      ...;
    else if (..)
      ...;
    else if (..)
    {
      ...
    }
    else
      ...;

    while (..)
    {
      ...
    }

    switch (..)
    {
      case 0:
        ...
      break;

      case 1:
        ...
      break;

      default:
        ...
      break;
    }
  }
}
```

3.8 Expressions

An expression is an assignment, a calculation or a condition. It is preferred to let expressions not to become too complex, where possible expressions will have to be split up into several sub expressions. If one expression does not fit on one line and has to be split up into more lines, make sure that the next line where the expression continues starts in the same column as where the expression started in the previous line.

Examples:

```
i = ((int32s) PID[id].k * (int32s) PID[id].ts * PID[id].accErr) /
     (int32s) PID[id].ti;

Meters = (int16u) (len * (float) 100);

while (getchar () != FUNC_KEY && quit != TRUE)
  ...;

quit = ((getchar () == QUIT_KEY) ||
        (TimerTimeOut (InactivityimerId) && TimerRunning (InactivityTimerId));
```

The C language lacks a true boolean type, therefore its logic operations (! == > < >= <=) and tests (in the conditional operator ?: and the if, while, do, and for statements) have some interesting semantics. Every boolean test is an implicit comparison against zero (0). However, zero is not a simple concept. It represents:

- the integer zero for all integral types
- the floating point 0.0 (positive or negative)

- the nul character
- the null pointer

In order to make your intentions clear, explicitly show the comparison with zero for all scalars, floating-point numbers, and characters. This gives us the tests:

- (i == 0) instead of (i)
- (x != 0.0) instead of (!x)
- (c == '\0') instead of (c)

An exception is made for pointers, since 0 is the only language-level representation for the null pointer. (The symbol NULL is *not* part of the core language - you have to include a special header file to get it defined.) In short, pretend that C has an actual boolean type which is returned by the logical operators and expected by the test constructs, and pretend that the null pointer is a synonym for false.

3.9 Function calls

The name of a function in a function call is always followed by one space and then the opening bracket of the function parameter list. When there are no function parameters to be passed, the closing bracket follows the opening bracket directly (`function ();`). When function parameters are passed to the function, the first parameter follows the opening bracket directly without spaces. The last parameter in the function call is followed directly by the closing bracket without spaces. Function parameters in the calling list are further separated by comma's followed by one space. If the parameters or expressions used in the function ca ll exceed the maximum line length, it is allowed to split up the call on more lines.

Example:

```
printf ("%2d.%1d", Cfg.advisedSpeed / 10, Cfg.advisedSpeed % 10);
```

or:

```
printf ("%2d.%1d",
        Cfg.advisedSpeed / 10,
        Cfg.advisedSpeed % 10);
```

3.10 Comments

For the usage of comments in modules a number of general rules apply:

- Use as much English as possible.
- For single line comments it is allowed to use the C++ style comments (//). However try to be consequent and choose for one type of single line comments throughout the module.
- The header files must provide as much comments as possible in order to describe the functionality of the functions and the module in enough detail. The header file contains the definitions for it's usage by the outside world.
- It is not allowed to have code follow a comment on the same line. When comments are used the closing comment should always be the last on that line.
- It is preferred that comments following statements on multiple lines all start in the same column.
- Comments have to be opened and closed on the same line, exception to this rule are the comment blocks used in function headers in the header files or comment blocks used below the start of the function implementation.
- Comments should not exceed the maximum line length.
- Commenting out blocks of code must be done with the *#if 0* preprocessor statement. Do not comment out blocks by using the normal comment symbols, C compilers can have problems with nested comments.

Some examples of comments:

```
/**** Define the screen states ****/
typedef enum
```

```
{
  DAY_COUNT,
  TOTAL_COUNT,
  RPM,
  ALARM
};


/**** Initialize CAN modules ****/
CanInit ();                                   /* CAN driver module */
CanProtocolInit ();
CanProtocolStart ();


/**** Handle Rpm every timer tick ****/
handleRpm ();
/**** Every 60 msec, manage all inputs and outputs ****/
if ((TickCount % MSEC_TO_TICKS (60)) == 0)
  ...
```