Technical Document

GENERIC PROTOCOL STACK FRAMEWORK

GPF

CODING STANDARD

MEMO

Document Number:	8415.100.00.103
Version:	0.6
Status:	Draft
Approval Authority:	
Creation Date:	2000-Apr-07
Last changed:	2015-Mar-08 by XINTEGRA
File Name:	Gpf_memo_crules.doc

Important Notice

Change History

Date	Changed by	Approved by	Version	Status	Notes
2000-Apr-07	MP et al.		0.1		1
2000-Jun-07	HSC et al.		0.2		2
2000-Jun-21	HCS et al.		0.3		3
2000-Jul-18	HSC et al.		0.4		4
2001-Feb-15	HSC et al.		0.5		5
2003-May-20	XINTEGRA		0.6	Draft	

Notes:

1. Initial version
2. Review in TUCT
3. After plenary assembly of TUCT
4. Feature Flags and other minor updates
5. Minor updates

Table of Contents

List of Figures and Tables

List of References

[ISO 9000:2000] International Organization for Standardization. Quality management systems - Fundamentals and vocabulary. December 2000

1 Introduction

The C Coding Standard documents rules for the programming in C. This concerns the general layout of the code and the header files, code generation, the spelling of file names and identifiers, indentation and last not least programming discipline.

The rules are in force from July 1st, 2000 for all software written in C at the Technology Unit Communication Technology (TUCT). Older programs may remain unchanged. Automatically generated software may diverge from these rules.

2 How to apply?

1. Follow the rules.
2. If this is - in rare cases - not possible or not practicable, you should be able to give the reasons for breaking the rule. For instance, consider the rule 'warnings, that occur during compilation should be minimized'. If you see nevertheless a warning, there should be a comment in the code at the location producing the warning that explains the harmlessness of the code.

3 General

3.1 ANSI C

Modern C compilers support some or all of the ANSI proposed standard C. You should always write code to run under standard C, and use features such as function prototypes, constant storage, and volatile storage. Standard C improves program performance by giving better information to optimizers. Standard C improves portability by insuring that all compilers accept the same input language and by providing mechanisms that try to hide machine dependencies or emit warnings about code that may be machine-dependent.

3.2 Headers

Every file containing C source code must begin with a standard header that describes the purpose of the file. In addition, each function declaration must be launched with a header announcing name, purpose, parameters, and return value. Templates for file as well as function headers are provided under GPF\template\pei. They also prescribe the order of defines, includes, declarations etc. in the file.

3.3 Comments

The comments should be meaningful. They should describe *what* is happening and *how* it is being done. Avoid, however, comments that are clear from the code, as such information rapidly gets out of date. Comments that disagree with the code are of negative value. C is not assembler; putting a comment at the top of a 3-10 line section telling what it does overall is often more useful than a comment on each line.

- Comments to code lines appear either in extra lines before or if space permits on the same line after the commented code.
- A comment to a variable declaration should always be after the declaration in the same line.
- To follow ANSI rules comments must be included in /* ... */. It is not allowed to use // as some compilers do not interpret this comment. As an extension to the ANSI rules where nested comments are forbidden, it is generally not allowed to put any comment sign into a comment (e.g. like /* … /* … */).
- In each #if(n)def/#endif block, the closing #endif and (if existing) the #else must be commented with the #if(n)def condition. The same holds for #if (!)defined.
  Example: #ifdef NEW_FRAME

```
…
#else /*NEW_FRAME */
…
#endif /* NEW_FRAME */
```

3.4 Include Files

- Include files are files that are included in other files prior to compilation by the C preprocessor. They are also used to contain data declarations and defines that are needed by more than one program. They should be functionally organized, i.e., declarations for separate subsystems should be in separate files.
- Include files should not be nested. In very rare cases, where a large number of files are to be included in several different source files, it may be acceptable to put all common #includes in one include file. But keep in mind that such a solution probably produces more dependencies between the files than necessary.
- It is common to put the following into each *.h* file to prevent accidental double-inclusion.
  ```
  #ifndef EXAMPLE_H
  #define EXAMPLE_H
  … /* body of example.h file */
  #endif /* EXAMPLE_H */
  ```

3.5 Prototypes

Each global function must have a function prototype that is part of the corresponding include file.

3.6 Warnings

- Warnings that occur during compilation should be minimized.
- The warning level to be selected is compiler dependent. There can't be a general rule about warning and warning levels. If the warning level is too high, many unimportant compiler information messages may hide the warnings about possible errors. If it is too low the warnings about possible errors may be suppressed. Thus, select a suitable error level, consider each compiler warning, and eliminate it by changing your code. Unavoidable warnings should be marked with a corresponding comment giving the reasons why they are unavoidable and substantiating their harmlessness. At least the warnings given in chapter 9 should not be suppressed and not appear during compilation.
- Don't use '#pragma' to change the warning level.

3.7 Code Generation

- Don't use precompiled headers. At the moment this mechanism doesn't work generally reliable and its benefit is slight.
- Don't rely on project files. Use makefiles instead. It must be possible to compile each part of the code and to build the whole system via usual (not automatically generated) makefiles.

4 Names

All names should be meaningful and give an idea of their application.

4.1 Files

In general the names of the C-modules should consist of two parts separated by an underscore. The first part represents the component the file belongs to; the second part gives more detailed information. Components usually consist of several related files and are defined by their dedicated functionality. A component may be e.g. an entity like cc or a layer like vsi.

Example:

cc_pei.c	PEI interface of the entity CC

vsi_tim.c Timer functionality of the VSI in the frame

Exceptions from this rule are allowed if necessary or useful.

Example:

tools.c general functions for converting data

In addition to the underscore and the dot character only lower case characters and digits are allowed in file names. The base name of a file is not limited to 8 characters.

4.2 Types

The names of new data types defined with the 'typedef' command have to start with a 'T_' and only have to contain upper-case letters, digits and underscores.

Example:

```
typedef struct
{
.....
} T_CC_DATA;
```

Exceptions are allowed for standard data types U8, S8, U16, S16, U32, S32. These standard types should be used if a certain number of bytes is required to represent a variable, e.g. at interface definitions. In all other cases the C standard data types should be used.

4.3 Functions

Function names must consist of the component name, an underscore, and any other sequence of lower case letters, digits, and underscores. For the component name also lower case letters must be used.

Example:

```
void vsi_d_callback (T_DRV_SIGNAL *signal)
{
}
```

4.4 Variables

Variable names must consist of any sequence of lower case letters, digits, and underscores. Global variables (i.e. variables exported from the file) – if the use of global variables is unavoidable - must be prefixed with the component name. For the component name also lower case letters must be used.

Shorter and less meaningful names may be used for variables local to functions.

Example:

```
for (i = 1; i < MAX_VAL; i++)
  tif_rcvbuffer[i] = data[i].
```

4.5 Constants

Constant names must consist of upper-case letter that may be separated by '_' characters and must not begin with 'T_'. As for variables the names should be meaningful but not oversized.

Example:

```
#define TIMER_SLOW_DOWN        4
```

Further possibilities for constants are 'const' and 'enum' declarations, but take chapter 8 into consideration for their use. For 'const' the spelling rules for variables must be used. An enumeration has to spelled like constants defined by '#define'.

4.6 Macros

Macro names must consist of upper-case letter that may be separated by '_' characters and must not begin with 'T_'.

4.7 Feature Flags

In order to allow that different products are possible, different configurations shall be supported. "Configuration" in the context here is related to *features* which are or are not supported by a specific product, e.g. one product could supports the feature Calling Name Presentation (CNAP) whilst another does not support CNAP. On the C code level, C preprocessor constructs are used, to realize the so-called feature flags.

Feature flag names must consist of upper-case letter that may be separated by '_' characters and must begin with 'FF_'.

Example:

```
#ifdef FF_CNAP
        … /* some code only required for the feature CNAP */
#endif /* FF_CNAP */
```

4.8 Further Identifiers

All other identifiers not mentioned up to now (e.g. structures, structure members, or directories) must consist of any sequence of lower case letters, digits, and underscores.

5 Typography

- In new blocks the opening and closing parenthesis have to be put in the same column as the code before the block. The code within blocks has an indent of two white spaces. Comment lines have to be indented as if they were code lines.
- The number of characters in one line should be limited to 80.
- Tabulators are not allowed. If you want to use the tabulator key, make sure that the editor inserts two blanks instead. Make also sure that eight successive blanks are not replaced by a tabulator.
- When declaring functions, the leading parenthesis of the parameter list and the first argument are to be written on the same line as the function name. If space permits, other arguments and the closing parenthesis may also be written on the same line as the function name. Otherwise, each additional argument is to be written on a separate line (with the closing parenthesis directly after the last argument). The return type is written in same line as the function name. In functions without parameters the keyword 'void' should be used. The keyword 'const' in the parameter list should be used where it is suitable.

  Examples:

```
void func1 (void)
{
}

void func2 (U16 entry,
            const char* name,
            int very_long_parametername,
            unsigned long and_atleast_anotherone)
{
}
```

- The variable declarations (outside functions) as well as function declarations should start in column 0.
- In constructions with 'if' and 'else' as well as in loops using 'while' or 'for' the conditional statement resp. the statement of the loop should be included in parenthesis even if it is only a single statement. This reduces the probability to make a mistake during extension of the conditions/loops.

  Examples:

```
if (entry == FIRST_ENTRY)
{
  idx = 0;
}
```

```
do
{
  idx++;
}
while (com_table[idx] == NULL);
```

- In switch constructs the statements after a label again are meshed 2 characters to the right. If the 'break' is omitted though there is at least one statement after the label, a comment must indicate this as intentional. It is recommended to use /*FALLTHRU*/ or /*FALLTHROUGH*/ - without any blanks - as indicating comment to satisfy checkers like lint, if they are used.
  A default label is mandatory.

  Example:

```
switch (foo)
{
  case ONE:
  case TWO:
  case THREE:
    statement1;
    statement2;
    break;
  case FOUR:
    statement3;
    statement4;
  /*FALLTHROUGH*/
  case FIVE:
    {
      statement5;
      statement6;
    }
    break;
  default:
    statement7;
    break;
}
```

6 Coding

- All global data should be initialized in an initialization function. This guarantees consistent data not only after the first download to a target system but also at every restart of the system.
- The intention of using macros is to keep the code readable. As source code debugging of macros is difficult macros should only contain a few statements. If the content of the macro exceeds a certain complexity it makes sense to use a function call instead. In the macro definition each parameter must be put into parentheses to avoid any unintentional and probably target dependent effects during macro processing. Depending on the macro it could additionally be necessary to put the entire right side of a macro definition completely in parentheses.
- In complex data structures that contain different types of parameters like long and char, the parameters of the largest type e.g. long should be placed before the parameters of the type char. Due to the target dependent alignment e.g. to place variables of the type long only on addresses dividable by 4, this may support portability (and can help to save memory).
- Statement sequences with one or more 'else if' suites should be terminated with an 'else' clause.
- The types of function parameters and return codes must occupy at least 2 bytes. Stack operations always operate on even byte counts. Thus, it is better to control the second byte. This may imply explicit casts.

- The increment/decrement of variables that are used two or more times within one statement should not be done in the same statement because the result may be target dependent.

  Example: Do not use a[i] = b[i++];

  Use a[i] = b[i];

  i++; instead.

- Macro must not have side effects, e.g. by incrementing or decrementing variables in a macro call.
- The keyword 'extern' may only be used in included header files.

7 Recommendations

- As macros should increase the readability of the code it should be avoided to call macros within macros.
- Do not use more than one statement in one line of code.
- Avoid the use of global variables. In multi-threaded environments the access to global variables may be a critical section. In this case it has to be controlled by a semaphore or a similar synchronization mechanism.
- Avoid labels and the corresponding 'goto' statement.
- Be extremely careful with typecasts of pointers. It may cause severe alignment problems to cast a char pointer to a pointer to a data type containing unsigned long parameters as these unsigned long parameters could sometimes only be located on addresses that can be divided by 4 (target dependent), whereas the char pointer itself can point anywhere.
- Avoid the usage of signed and unsigned variables in the same context. On the one hand this may cause problems during comparison of these variables, as they are not considered equal although they are apparently the same. On the other hand – if the variables also have different types like 'char' and 'int' - these problems may be even worse when the compiler does any extension to signed int that may not be realized by the programmer.
- Use pointers to pointers only if really necessary.
- Avoid the use of pointer to void (see also chapter 8).
- Try to avoid the use of identifiers that are keywords in C++.
- Do not use macros like 'GLOBAL' and 'LOCAL' for function or variable declarations.
- Avoid the using of macros TRUE and FALSE. Be aware that in C it is possible to define a unique value for FALSE, but not for TRUE. Thus never use a construct like 'var == TRUE'. If ever use 'var != FALSE' or better 'var'.

8 Problematic C Constructs

The use of the following constructs is permitted. But be careful with their use, because the may cause trouble in some cases.

- Union: There is no problem in using unions in the classical way of a variant record e.g. if data structures and their contents are represented at different moments in different ways. Error-prone and therefore prohibited is the use of unions for data conversion i.e. if the same data content is interpreted in different ways. This is increased in network environments with different byte-sex (little endian vs. big endian).
- Enum: The number of bytes an enum-constant occupies is not generally defined. It may depend on the machine, on the compiler, and last not least on the number of constants defined in one enumeration. This may cause errors especially when used via network interfaces. Another drawback of enums comes with their printing in traces because the printed values usually cannot be found in any source file.
- Const: The keyword 'const' is generally a useful feature of the static C type system, but constants defined with 'const' are not really constants but more a kind of variable that occupies addressable memory, for which the compiler tries to assure that it's content is not changed. Of course it is always possible to change the content of any memory cell. Thus, there is no advantage to use 'const'. Nevertheless, in some cases it must be used, e.g. if data has to be stored in the ROM of the target. Independent of this it is recommended to use 'const' for function parameters if it is suitable.

- Void*: The use of a pointer to void always implies a cast. This may cause problems if the use of the data pointed to is expected to be aligned (e.g. in structs).

9 Warnings

The following or analogous warnings should not be suppressed and not appear during compilation. Some of them are treated as errors by some compilers anyhow.

- Number of macro parameters (difference between definition and use).
- Macro redefinition.
- No function prototype / Undefined function / Old style declaration.
- Different types of formal and actual parameters.
- Function must [not] return a value.
- Different levels of indirection.
- Type mismatch in expressions (pointer, signed/unsigned).
- Useless operations on unsigned variables.
- Array bounds overflow in initialization.
- Case constant does not match switch expression.
- Local variable used, possibly without assigning it a value.
- Statement has no effect / Statement not reached.
- Assignment within conditional expression.
- Divide by zero / Zero modulus.

Appendices

A. Acronyms

DS-WCDMA	Direct Sequence/Spread Wideband Code Division Multiple Access

B. Glossary

International Mobile Telecommunication 2000 (IMT-2000/ITU-2000)	Formerly referred to as FPLMTS (Future Public Land-Mobile Telephone System), this is the ITU's specification/family of standards for 3G. This initiative provides a global infrastructure through both satellite and terrestrial systems, for fixed and mobile phone users. The family of standards is a framework comprising a mix/blend of systems providing global roaming. <URL: http://www.imt-2000.org/>