TCNOpen	**Open Source Developement**

TRDP

Train Real Time Data Protocol

TRDP Coding Rules

Document reference no: TCN-TRDP1-A-BOM-008-07

Author :	Armin-Hagen Weiss
Organisation :	Bombardier
Document date:	07 May 2013
Revision:	7
Status:	Issued

Dissemination Level		
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	X
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

DOCUMENT SUMMARY SHEET

This are the coding rules to be used in the project.

Participants		
Name and Surname	Organisation	Role
Armin-Hagen Weiss	BOM	Lead
Bernd Löhr	BOM	Review
Mikel Korta	CAF	Review
Simone Pachera	FAR	Review

History			
V1	28 Feb 12	Armin-H. Weiss	Initial version
V3	05 Mar 12	Armin-H. Weiss	Updated version
V4	07 Mar 12	Armin-H. Weiss	Released version
V5	13 Mar 13	Armin-H. Weiss	Chapters 2.6, 5.1, 5.2 updated
V6	15 Mar 13	Armin-H. Weiss	Following rules were deleted: 20106, 21002, 21602, 21702, 30403, 30405, 30406, 30508, 30511, 40006 Following rules were adapted: 21101, 21601, 30403, 30504, 30506, 50101
V6	18 Mar 13	Armin-H. Weiss	Chapters 5.1, 5.2 updated Rules 20804, 21601, 40005 updated Rule 30403 deleted (content in 20804 and 21601)
V7	7 May 2013	Armin-H. Weiss	Issued

Table of Contents

TABLE OF CONTENTS 3

TABLE OF TABLES 4

1. INTRODUCTION 5

1.1. PURPOSE 5
1.2. INTENDED AUDIENCE 5
1.3. REFERENCES/RELATED DOCUMENTS 5
1.4. ABBREVIATIONS AND DEFINITIONS 5
1.5. CONVENTIONS 6
1.5.1. Rule numbering 6
1.5.2. Use of colours 6
1.5.3. Automatic documentation of code 6

2. COMPLEMENTARY RULES TO MISRA-C AND MISRA-C++ 7

2.1. ENVIRONMENT 7
2.2. LANGUAGE EXTENSIONS 8
2.3. DOCUMENTATION 8
2.4. CHARACTER SETS 8
2.5. IDENTIFIERS 8
2.6. TYPES 8
2.7. CONSTANTS 9
2.8. DECLARATIONS AND DEFINITIONS 9
2.9. INITIALIZATION 10
2.10. ARITHMETIC TYPE CONVERSIONS 10
2.11. POINTER TYPE CONVERSIONS 11
2.12. EXPRESSIONS 11
2.13. CONTROL STATEMENT EXPRESSIONS 11
2.14. CONTROL FLOW 11
2.15. SWITCH STATEMENTS 12
2.16. FUNCTIONS 12
2.17. POINTERS AND ARRAYS 13
2.18. STRUCTURES AND UNIONS 14
2.19. PREPROCESSING DIRECTIVES 14
2.20. STANDARD LIBRARIES 14
2.21. RUN-TIME FAILURES 14

3. STYLE RELATED RULES 15

3.1. FILE HEADERS AND COMMON LAYOUT OF FILES 15
3.2. CODING IN GENERAL 16
3.3. INDENTATION AND SPACING 16
3.4. COMMENTS 18
3.5. NAMING CONVENTIONS 19

4. PORTABILITY RELATED RULES **22**

5. METRICS RELATED RULES **23**

5.1. FILES 23
5.2. FUNCTIONS 23
5.3. LINES 23

6. APPENDICES **25**

6.1. INCLUDE FILE (.H) 25
6.2. IMPLEMENTATION FILE (.C) 29

Table of Tables

Table 1: References 5
Table 2: Abbreviations and Definitions 5
Table 3: Basic Data Types 8

1. Introduction

1.1. Purpose

This document describes the rules when writing C code within the software development department. The rules in this document are complementary rules to MISRA-C:2004 (chapter 2) and MISRA-C++:2008 and are structured in the same way as in the MISRA-C:2004-document. The MISRA-C:2004 contains no subjective style rules but this document does (chapter 3). It also contains rules related to portability (chapter 4) and finally rules for metrics (chapter 5).

1.2. Intended Audience

This document is intended for all persons involved in the development and maintenance of software under the responsibility of the software development department.

1.3. References/Related Documents

Reference	Number	Title
[MISRAC]	MISRA-C:2004	Guidelines for the use of the C language in critical systems 2004
[MISRAC++]	MISRA-C++:2008	Guidelines for the use of the C++ language in critical systems 2008
[LINT]		Reference Manual for PC-lint/FlexeLint 9.00

Table 1: References

1.4. Abbreviations and Definitions

Abbreviation	Definition

Table 2: Abbreviations and Definitions

1.5. Conventions

1.5.1. Rule numbering

The purpose of the rule numbering in this document is to allow for adding of rules at the end of each subchapter without causing renumbering of all rules. Removing rules or adding rules in between will cause renumbering within the current subchapter but not in the other subchapters.
The rule numbering scheme is as follows:
XXXYY where:
XXX is three digits forming the subchapter number. YY is a sequential number (starting with 01) within the subchapter.

1.5.2. Use of colours

In the code examples found in the rules, the normal colour is green. To highlight code that does not follow the rules or is bad example of code, red and italic is used. To highlight parts of the code that the rule is describing, **blue and bold is used**.

1.5.3. Automatic documentation of code

Automatic code documentation by DOXYGEN shall be supported by the code.

2. Complementary rules to MISRA-C and MISRA-C++

The rules in this chapter are complementary rules to MISRA-C:2004 and MISRA-C++:2008 and are structured in the same way as in the MISRA-document [MISRAC]. This implies that there are chapters without any rules if there are no complementary rules for that MISRA chapter.

Rule 20001 (M) Every time a rule is broken, this must be clearly documented. C C++

There is one rule which can never be broken: this rule. If any other rule must be broken by some reason, it must be documented in the project as a project deviation.

2.1. Environment

Rule 20101 (M) Always compile with the highest warning level to eliminate as many warnings as possible. A tool for static code analysis shall be used, e.g. PC-lint. C C++

Code that is accepted by a compiler is not always correct. In order to reduce the amount of code that must be rewritten, let the compiler produce warnings if non-ANSI features and extended keywords are used. Use PC-lint or an equivalent tool to track down potential error sources.

Rule 20102 (M) Optimize code only if you know that you have a performance problem. Think twice before you begin. C C++

Various tests are said to have demonstrated that programmers generally spend a lot of time optimizing code that is never executed. If your program is too slow, use a tool to determine the exact nature of the problem before beginning to optimize. Let the compiler perform optimizations for you where possible.

Rule 20103 (M) Filenames shall only contain letters a to z (lower case only), digits 0 to 9, underscores, and one dot followed by the file type extension. C C++

Rule 20104 (M) Implementation files in C always have the file name extension ".c", in C++ “.cpp” C C++

Rule 20105 (M) Include files in C always have the file name extension ".h". C C++

Rule 20106 (M) File names shall be unique within the project. C C++

Dependencies on the directory path would prevent the reorganization of the project directory structure without changing file names at a later time. Use operating system facilities (e.g. user environment variables) for defining directory path names that may change from configuration to configuration.
Exception: Automatic generated files in subdirectories may have the same name because of its handling inside that automatism.

2.2. Language extensions

No complementary rules.

2.3. Documentation

Rule 20301	**(M)**	**Automatic code documentation generation with DOXYGEN shall be supported by comments of the code.**	**C**	**C++**

2.4. Character sets

Rule 20401	**(M)**	**The Unicode character set with UTF-8 encoding shall be used.**	**C**	**C++**

2.5. Identifiers

No complementary rules.

2.6. Types

Rule 20601	**(M)**	**Use standard type names for commonly used types**	**C**	**C++**

See also [MISRAC], rules 6.3.

Typename	**Type representation**
UINT8	unsigned 8 bit integer
INT8	signed 8 bit integer
UINT16	unsigned 16 bit integer
INT16	signed 16 bit integer
UINT32	unsigned 32 bit integer
INT32	signed 32 bit integer
UINT64	unsigned 64 bit integer
INT64	signed 64 bit integer
REAL32	floating-point
REAL64	double floating-point
CHAR	char
BOOL	int
UTF16	wide character

Table 3: Basic Data Types

```
#define FALSE ((BOOL)(0U != 0U))
#define TRUE ((BOOL)(0U == 0U
```

2.7. Constants

Rule 20701 (M) Do not use numeric values in code; use symbolic values instead. C C++

Numerical values in code ("Magic Numbers") should be viewed with suspicion. They can be the cause of difficult problems if and when it becomes necessary to change a value. A large amount of code can be dependent on such a value never changing, the value can be used at a number of places in the code (it may be difficult to locate all of them), and values as such are rather anonymous (it may be that every '2' in the code should not be changed to a '3').

From the point of view of portability, absolute values may be the cause of more subtle problems. The type of anumeric value is dependent on the implementation. Normally, the type of a numeric value is defined as the smallest type which can contain the value.

Exception:

0,1,-1 and formula

Other exceptions such as bitmasks and bitshifts shall be agreed in review, e.g. 0xFFFF0000UL is more readable and maintainable than UPPER16BITOF32MASK.

Rule 20702 (M) Use pre-defined names (defined in a header file) wherever possible. C C++

In particular, note the following definitions:

- Use NULL for zero pointer tests.
- Use FUNCPTR or VOIDFUNCPTR for pointer-to-function types (VxWorks example).

```
typedef int (*FUNCPTR) (); /* ptr to function returning int */
typedef void (*VOIDFUNCPTR) (); /* ptr to function returning void*/
```

2.8. Declarations and definitions

Rule 20801 (M) Avoid using global variables. Use global variables only when necessary. C C++

Rule 20802 (M) For basic type variables, the type appears first on the line and is separated from the identifier by one or more spaces C C++

The definition should be completed by a meaningful one-line comment.

Each variable declaration shall be on a separate line.

Example: Basic definition:

```
UINT32 rootMemBytes; /* memory for TCB and root stack */
```

Rule 20803 (M) The reference operator '*' should be directly connected with the name in declarations and definitions. The character '*' should be written together with the name of variables instead of with the type of variables in order to emphasize that they are part of the name. C C++

Instead of saying that i is an int*, say *i is an int. Traditionally, C recommendations indicate that '*' should be written together with the variable name, since this reduces the probability of making a mistake.

Example: Pointer declarations:

```
FooNode *pFooNode; /* foo node pointer */
FooNode **ppFooNode; /* pointer to a foo node pointer */
```

Rule 20804 (M) Static variables, types, constants, macros and #defines shall be described at least where they are declared. The description shall be done according to the following DOXYGEN example. C C++

Example:

```
/** Process data receiving information */
typedef struct
{
   UINT32      srcIpAddress;   /**< source IP address for filtering       */
   UINT32      dstIpAddress;   /**< destination IP address for filtering  */
   UINT32      seqCount;       /**< sequence counter                      */
   UINT16      protVersion;    /**< Protociol version                     */
   TRDP_MSG_T  msgType;        /**< Protocol ('PD', 'MD', ...)            */
   UINT32      comId;           /**< ComID                                */
   UINT32      topoCount;      /**< received topocount                    */
   BOOL        subs;            /**< substitution                         */
   UINT16      offsetAddress;   /**< offset address for ladder architecture*/
   UINT32      replyComId;     /**< ComID for reply (request only)        */
   UINT32      replyIpAddress;  /**< IP address for reply (request only)   */
   TRDP_ERR_T  resultCode;     /**< error code, user stat ???             */
 } TRDP_PD_INFO_T;
```

2.9. Initialization

Rule 20901 (M) If possible, always use initialization instead of assignment. C C++

By always initializing variables, instead of assigning values to them before they are first used, the code is made more efficient since no temporary objects are created for the initialization. For objects having large amounts of data, this can result in significantly faster code.
Example:

```
INT16 i = 5;
```

2.10. Arithmetic type conversions

Rule 21001 (M) Do not write code which depends on functions that use implicit type conversions. C C++

There are two kinds of implicit type conversions: either there is a conversion function from one type to another, written by the programmer, or the compiler does it according to the language standard. Both cases can lead to problems.

Some of these conversions can result in loss of information (e.g. conversions to a narrower type).

As a general principle, avoid mixing arithmetic of different precisions in the same expression, and avoid mixing signed and unsigned integers in the same expression. Mixed arithmetic normally entails implicit promotions and balancing of types (i.e. conversions), some of which can lead to unexpected behavior.

Extra care is necessary when using small integer types (char, short, bit-field and enum) because these are always converted to type int or unsigned int before an arithmetic operation. This is called integral promotion. See also [1], rule 10.5 regarding integral promotion when using bitwise operators ¨, <<, and >>.

Rule 21002 (M) Do not make pointer conversions from a “shorter” type to a “longer” one. C C++

A processor architecture often forbids data of a given size to be allocated at an arbitrary address. For example, a word must begin on an "even" address for MC680x0. If there is a pointer to a char which is located at an "odd" address, a type conversion from this char pointer to an int pointer will cause the program to crash when the int pointer is used, since this violates the processor's rules for alignment of data.

2.11. Pointer type conversions

Rule 21101 (M) NULL shall be defined as C C++
#ifndef NULL
#define NULL ((void*)0)
#endif
Uinitialized pointers shall be tested against NULL

.

2.12. Expressions

Rule 21201 (M) Mixed precision arithmetic shall use explicit casting to generate the desired result. C C++

In an expression, sub-expressions are evaluated at the precision appropriate to the types of the operands. This may be less than the precision of the final result. It is therefore possible to be misled into creating sub-expressions which are evaluated at the wrong precision, which may result in values which are not as the programmer intended.
Example:

```
UINT16 i = 65535U;
UINT16 j = 10U;
UINT32 eO = i + j; /* incorrect = 9 */
UINT32 ei = (UINT32)(i + j); /* incorrect = 9 */
UINT32 e2 = (UINT32) i + j; /* correct = 65545 */
UINT32 e3 = (UINT32) i + (UINT32) j; /* correct = 65545 */
```

2.13. Control statement expressions

Rule 21301 (M) Tests of a value against zero should be made explicit, also for Boolean. C C++

Rule 13.2 in [1] makes an exception for Booleans but this exception is overruled by this rule.
Example
:

```
Boolean flag = FALSE;
if (!flag) … /* not correct */
if (flag == FALSE) … /* correct */
```

2.14. Control flow

Rule 21401 (M) Always use inclusive lower limits and exclusive upper limits. C C++

It is best to use inclusive lower and exclusive upper limits. Instead of saying that x is in the interval x>=23 and
x<=42, use the limits x>=23 and x<43. The following important claims then apply:
• The size of the interval between the limits is the difference between the limits.
• The limits are equal if the interval is empty.
• The upper limit is never less than the lower limit.
By being consistent in this regard, many difficult errors will be avoided.
Example: Good and bad ways of setting limits for loop variables:

```
INT32 a[10];
INT32 ten = 10;
INT32 nine = 9;
INT32 i;
INT32 j;
/* Good way to do it: */
for (i = 0; i < ten; i++) /* Loop runs 10-0=10 times */
{
   a[i] = 0;
}
/* Bad way to do it: */
for (j = 0; j <= nine; j++) /* Loop runs 10 times, but 9-0=9 !!!*/
{
   a[j] = 0;
}
```

Rule 21402 (M) All functions shall have bounded execution. C C++

A function shall never be able to enter an “infinite” loop. There shall always exist a provable or explicit finite exit criterion. For example, a function comparing two null-terminated text strings shall have a parameter in the parameter list that specifies the maximum number of characters to compare. If the compare has not been finished when the last characters have been compared, the function has failed. If it can be proved that the function will stop after a finite number of iterations, there is no need to provide an explicit limit.
There are two exceptions to this rule:
• The function represents a task that shall run forever.
• An infinite loop is entered intentionally, e.g. entering a “fail safe” state in a task, which shall never be exited.
However, in this case it would probably be better to halt or kill the task.

2.15. Switch statements

No complementary rules.

2.16. Functions

Rule 21601 (M) A function shall be described at least where it is declared. The description shall comprise functionality, input- and output arguments as well as return values. The description shall be done according to the following DOXYGEN example. C C++

Example:

```
/***********************************************************************************/
/** Receive UDP data.
 *  The caller must provide a sufficient sized buffer.
 *  If the supplied buffer is smaller than the bytes received,
 *  *pSize will reflect the number of copied bytes and the call should be repeated
 *  until *pSize is 0 (zero). If the socket was created in blocking-mode (default),
 *  then this call will block and will only return if data has been received or the
 *  socket was closed or an error occured.
 *  If called in non-blocking mode, and no data available, VOS_NODATA_ERR will be returned.
 *
 *  @param[in]      sock            socket descriptor
 *  @param[out]     pBuffer         pointer to applications data buffer
 *  @param[in,out]  pSize           pointer to the received data size
 *  @param[out]     pIPAddr         source IP
 *  @param[out]     pIPPort         source port
 *
 *  @retval         VOS_NO_ERR      no error
 *  @retval         VOS_PARAM_ERR   sock descriptor unknown, parameter error
 *  @retval         VOS_IO_ERR      data could not be read
 *  @retval         VOS_NODATA_ERR  no data
 *  @retval         VOS_BLOCK_ERR   Call would have blocked in blocking mode
 */

EXT_DECL VOS_ERR_T vos_sockReceiveUDP (
    INT32    sock,
    UINT8    *pBuffer,
    UINT32   *pSize,
    UINT32   *pIPAddr,
    UINT16   *pIPPort);
```

Rule	**21602**	**(M)**	**Always pass pointers to structures. Never pass structures directly.**	**C**	**C++**

Rule	**21603**	**(M)**	**Never return structures directly. Use pointers to structures.**	**C**	**C++**

2.17. Pointers and arrays

Rule	**21701**	**(R)**	**Pointers, pointing to different types of objects should be avoided.**	**C**	**C++**

Rule	**21702**	**(R)**	**Avoid pointers of type void.**	**C**	**C++**

Usage of void pointers must always be checks during review but might be clearer than other casts.

Rule	**21703**	**(M)**	**Use a typedef to simplify program syntax when declaring function pointers.**	**C**	**C++**

Typedef is a good way of making code more easily maintainable and portable. Another reason to use typedef is that the readability of the code is improved. If pointers to functions are used, the resulting code can be almost unreadable. By making a type declaration for the function type, this is avoided.
Function pointers can be used as ordinary functions; they do not need to be dereferenced.
Example:Syntax simplification of function pointers using a typedef:

```
#include <math.h>
/* Ordinary messy way of declaring pointers to functions: */
Real (*mathFunc) (Real) = &sqrt;
/*
* With typedef, life is filled with happiness (chinese proverb):
*/
typedef Real MathFuncType(Real);
MathFuncType *pfMathFunc = &sqrt;
void main()
{
    /* You can invoke the function in an easy or complicated way */
    Real returnValue1 = pfMathFunc(23.0);    /* Easy way */
    Real returnValue2 = (*pfMathFunc)(23.0); /* No! Correct, but complicated */
}
```

2.18. Structures and unions

Rule 21801 (M) Structures (and unions) should always be defined by a typedef declaration. C C++

Typically, the keyword struct appears on the first line with optional structure tag. The opening brace appears on the next line, followed by the elements of the structure, each placed on a separate line with the appropriate indentation and comment. If necessary, the comments can extend over more than one line; The definition is concluded by a line containing the closing brace, and the ending semicolon.
Example: Structure definition:

```
typedef struct tag_BlockDev /* BLOCK_DEV */
{
    FUNCPTR pfBlockRead; /* function to read blocks */
    FUNCPTR pfBlockWrite; /* function to write blocks */
    FUNCPTR pfIoctl; /* function to ioctl devices */
} BlockDev;
```

This format is used for other composite type definitions such as union and enum.

2.19. Preprocessing directives

Rule 21901 (R) Avoid C-macros. C C++

2.20. Standard libraries

Rule 22001 (M) Use only the string operation functions with length parameter. C C++

Example:
use strncpy instead of strcpy, use snprintf instead of sprintf, …

2.21. Run-time failures

No complementary rules.

3. Style related rules

This chapter contains rules related to programming style. The rules are subjective and there are almost as many styles as there are programmers but the rules defined here are considered to do the code easy to read, understand and maintain.

3.1. File headers and common layout of files

Rule 30101 (M) Every text-file shall be documented with an C C++ introductory comment (header) that provides information about the file.

The following sections shall be included in the following order:
• **Copyright:** Contain the appropriate copyright information.
• **Component:** The name of the component this file is a part of.
• **File:** The name of the file.
• **Requirements:** Requirements this file implements (fully or partly).
• **Abstract:** A short description of the file contents.
• **History:** History over changes made in the file. The history shall also contain the revision number of the file and preferably be generated automatically by the Configuration Management tool.
Note that depending on the tools used in the development environment, some of the information above can be excluded from the files if it can obtained in another way. E.g. the history for a file can perhaps be obtained from the Configuration Management tool used or the requirements can be traced with a Requirements Management tool or a Requirements Matrix.

Rule 30102 (M) Every source file shall follow the layout in the C C++ appendices.

The conventions in this section define the standard file layout that shall come at the beginning of every source file following the standard file heading.
The file shall consist of the blocks described below; the blocks shall be separated by one or more blank lines. If there are no declarations to write in a block, the block heading shall be kept empty for future use.
Header file:
• **Includes**: The includes block consists of one or more C pre-processor #include directives. This block groups all header files included in the file in one place.
• **Defines:** The defines block consists of one or more C pre-processor #define directives. This block groups all definitions made in the file in one place.
• **Typedefs:** The typedefs block consists of one or more C typedef statements, one per line. The block groups all type definitions made in the file in one place.
• **Global Variables:** The global variables block consists of one or more declarations, one per line. This block groups together all declarations in the file that are intended to be visible outside the file.
• **Global Function Declarations:** The global functions block consists of one or more ANSI C functions. This block groups together all functions in the file that are intended to be visible outside the file.
C-code file:
• **Includes**: The includes block consists of one or more C pre-processor #include directives. This block groups all header files included in the file in one place.
• **Defines:** The defines block consists of one or more C pre-processor #define directives. This block groups all definitions made in the file in one place.
• **Typedefs:** The typedefs block consists of one or more C typedef statements, one per line. The block groups all type definitions made in the file in one place.
• **Local Function Declarations:** The local function declarations block consists of prototypes of one or

more ANSI C functions defined in the local function definitions block.

- **Local Variables:** The local variables block consists of one or more C declarations, one per line. This block groups together all declarations in the file that are intended not to be visible outside the file.
- **Global Variables:** The global variables block consists of one or more declarations, one per line. This block groups together all declarations in the file that are intended to be visible outside the file.
- **Local Function Definitions:** The local function definitions block consists of one or more ANSI C functions. This block groups together all functions in the file that are intended not to be visible outside the file.
- **Global Function Definitions:** The global function definitions block consists of one or more ANSI C functions. This block groups together all functions in the file that are intended to be visible outside the file.

Rule 30103 (M) Every function definition shall be documented with an introductory comment. C C++

These comments shall encompass the following information:

- **Abstract:** A complete description of what the function does and how to use it.
- **Return value:** Description of the function return value.
- **Global variables:** Description of global variables used in this function.

See also Rule 21601 regarding documentation of input and output arguments and Rule 30403 regarding function declarations.

3.2. Coding in general

Rule 30201 (M) Write ANSI compatible code. C C++

Rule 30202 (M) Braces ('{', '}') shall enclose all blocks even if a block only consists of one statement. C C++

Rule 3202

Example: Braces enclosing blocks:

```
/* NO!!*/
for (i = 0; i < xLength; i++)
x[i] = 2 * i;
/* Yes */
for (i = 0; i < xLength; i++)
{
    x[i] = 2 * i;
}
```

Rule 30203 (M) If some bits of bit fields are not used they shall be defined by dummy names (reservedx). C C++

Rule 30204 (M) Not used (reserved) parts of structures or bit fields shall be set to 0. C C++

3.3. Indentation and spacing

Rule 30301 (M) Use spaces instead of tabs. C C++

Ordinary spaces shall be used instead of tabs. Since different editors treat tab characters differently, the work in perfecting a layout may have been wasted if another editor is later used. Tab characters can be replaced by spaces by the editor. Code will then have a uniform appearance regardless of who has written it.

Rule 30302	**(M)**	**One indentation level shall consist of four (4) characters**	**C**	**C++**

Rule 30303	**(M)**	**Indent one indentation level after function declarations, conditionals, looping constructs, switch statements, case labels, and structure definitions in a typedef**	**C**	**C++**

Rule 30304	**(M)**	**The file and function headings and the function declarations shall start in column one.**	**C**	**C++**

Rule 30305	**(M)**	**Continuation lines shall line up with the part of the preceding line they continue. This applies also for structures.**	**C**	**C++**

Example: Layout for continued lines:

```
a = (b + c) *
(d + e);
status = fooList(foo, a, b,
c, d, e);
if ((a == b) &&
(c == d))
```

Exception: If the left part of the expression is exceptionally long (including indentation) the continuation line can line up with the left part of the expression but indented one level:

```
theExceptionallyLongLeftPartOfExpression = firstVariable +
secondVariable + thirdVariable + fourthVariable +
fifthVariable + sixthVariable;
```

Rule 30306	**(M)**	**In a function call always write the left parenthesis directly after a function name. In a function declaration always write a space between the function name and the left parenthesis.**	**C**	**C++**

Example: The left parenthesis always directly after the function name:

```
void foo (void); /* Declaration! */
void foo(void); /* Call! */
```

Rule 30307	**(M)**	**Braces ('{', '}') which enclose a block shall be placed in the same column, on separate lines directly before and after the block.**	**C**	**C++**

It is easier to check whether an opening brace has a matching closing brace.

Example: Braces enclosing blocks:

```
while (i < iLength)
{
    cout += i;
}
if (condition)
{
    statements
}
```

```
else if (condition)
{
    statements
}
else
{
    statements
}
switch (input)
{
case 'a':
...
break;
case 'b':
...
break;
default:
...
break;
}
```

Rule	**30308**	**(M)**	**Do not use spaces around '.' or '->', nor between unary operators and operands.**	**C**	**C++**

Code is more readable if spaces are not used around the . or -> operators. The same applies to unary operators (those that operate on one operand), since a space may give the impression that the unary operator is actually a binary operator.

Example:

```
foo.index = aNumber;
pFoo->index = aNumber;
++unaryOperation;
aNumber = 3 – (-2);
```

Rule	**30309**	**(M)**	**Put spaces around binary operators (+, *, /, %, -), bitwise operators (&, |, <<, >>), logical operators (&&, ||), equality operators (==, !=), assignment operators (=, *=, /=, %=, -=,<<=, >>=, &=, ^=, |=), after commas. Do not put spaces before opening brackets of array subscripts.**	**C**	**C++**

Example: Horizontal spacing:

```
status = fooGet(foo, i + 3, &value);
fooArray[(max + min) / 2] = aBinaryValue << aNumber;
string[0] = aCharacter;
if (a != b)
{
    …
}
```

3.4. Comments

It is necessary to document source code. By properly choosing names for variables and functions and by properly structuring the code, there is less need for comments within the code.

Rule	**30401**	**(M)**	**All comments shall be written in English.**	**C**	**C++**

Rule	**30402**	**(M)**	**Use strategic comments.**	**C**	**C++**

A strategic comment describes what section of code is intended to do, and is placed before this code with the same indentation as the following code.

Example:

```
/* A short strategic comment */
{
    /*
    * A longer strategic comment over several lines. The comment can
    * e.g. explain a following bubble sort algorithm and why it is used
    */
    aVariable = anotherVariable;
    …
}
```

Rule 30403 (M) Comments shall not be nested. C C++

C does not support the nesting of comments. After a / * begins a comment, the comment continues until the first * / is encountered, with no regard for any nesting which has been attempted.

3.5. Naming conventions

In this chapter, it is important to distinguish between identifiers and names. The name is that part of an identifier that shows its meaning.
An **identifier** is defined to consist of (in the specified order):
[<tag_prefix>][<scope_prefix>][<ptr_prefix>|*<func_ptr_prefix>]<name_part>[<suffix>]*
where the prefixes and the suffix are optional.
<tag_prefix>
tag_ preceding the tag identifier in a *struct, union* or *enum* typedef, see Rule 3503 and Rule 3507
<scope_prefix>
The possible scope prefixes are:
priv_ a variable declared *static* at file scope with internal linkage
<component_name>_
The name or abbreviation of a component followed by an underscore. Usually, the component name will be the name of the header file (excluding the file extension) exporting the interface. The length of the *<component_name>_* should not exceed 8 characters, therefore an abbreviation may be used (which shall be defined and documented within the project) to avoid this. Since file names shall be in lower case letters (see Rule 20103), the component name shall also be in lower case letters. The component name is used for types and objects exported from a component, see Rule 30504 and Rule 30506.
<ptr_prefix>|*<func_ptr_prefix>*
The possible prefixes are:
p pointer to a variable with one level of indirection
pp pointer to a variable with two level of indirection
pf pointer to a function with one level of indirection
ppf pointer to a function with two level of indirection
prefixes such as b (bool), f(flag), sem(semaphore), en(enum) could be helpful in special places.
<name_part>
Shows the meaning of the identifier.
<_suffix>
Currently, the naming convention does not specify any suffix. However, suffixes may be introduced by a project, e.g. as a result of design patterns. The suffix shall be preceded by an underscore to be able to distinguish it from the name part.
An **object name**, is an identifier that represents an object with an unqualified or qualified type, i.e. variables, pointers, functions, and function pointers.
The purpose of naming rules is to make programs more readable for all members of a project. When a programmer sees a name, it might be out of context. A name that seems cute or easy to type can cause trouble to someone trying to decipher code. Remember, code is read many more times than it is written. One rule of thumb is that a name which cannot be pronounced is a bad name. A long name is normally better than a short, cryptic name.
Abbreviations can always be misunderstood. Global variables, functions and constants shall have long enough names to avoid name conflicts, but not too long.

Rule 30501 (M) Names shall be written in English. C C++

Rule 30502 (M) All identifiers being an object name shall be in camel casing, i.e. the words are written together with the first character capitalized in each word except for the first word. The prefixes <ptr_prefix>, or <func_ptr_prefix>, if present, comprises the first word. C C++

Example:

```
UINT32 *pMessageBuffer;              /* Local pointer */
UINT32 messageLength;                /* Local variable */
UINT32 messageBufferIndex;           /* Local variable */
UINT32 TLC_bufferAdd(UINT32 newItem);/* Global function in TLC component*/
```

Rule 30503 (M) Type identifiers, i.e. defined by use of typedef, shall be written in camel casing with the first character capitalized. C C++

Example:

```
typedef UINT32 (*FooPtr)(U32 arg1, U32 arg2);
typedef struct tag_MessageHeader
{
    UINT32 messageId;
    UINT32 length;
} MessageHeader;
```

Rule 30504 (M) Type identifiers defined and exported by a component shall have a <*scope_prefix*> *::=<component_name>_* in lower case letters. In C++ namespaces shall be used instead. C C++

Example:

```
typedef struct tag_log_MessageHeader
{
    UINT32 messageId;
    UINT32 length;
} log_MessageHeader;
```

Exception:

Common type identifiers with no relation to any specific component do not need to have any <*scope_prefix*>.

Typically, common type identifiers could be type identifiers replacing basic numerical types (see e.g. [1], rule 6.3 and Rule 20601).

Rule 30505 (M) Variables declared at file scope with internal linkage, i.e. declared *static*, shall be prefixed by <*scope_prefix*> ::= *priv_*. The <*scope_prefix*> is not considered part of the first word. C C++

Example: Variables with internal linkage:

```
static UINT32 *priv_pMessageBuffer;
static UINT32 priv_logBuffer[MAX_BUFFER_SIZE];
```

The rationale behind the trailing underscore in the scope prefix is to avoid ambiguities when the scope prefix needs to be combined with any other prefixes.

Rule 30506 (M) Global object names, i.e. objects declared at file scope with external linkage shall be prefixed in C by <*scope_prefix*> ::= <*component_name*>_ in lower case C C++

	letters. The *<component_name>_* is not considered part of the first word. In C++ namespaces shall be used instead.		

```
extern BOOL log_enabledFlag;
extern UINT32 log_sendMessage(const Char * const msg);
```

The rationale behind the trailing underscore in the component name prefix is to avoid ambiguities when the component name prefix needs to be combined with any other prefixes.

Rule 30507 (M)	**The names of constants (via #define), enumeration constants (members of enumerations), macros (via #define), and preprocessor names are to be written completely with uppercase letters with underlines separating the words in the name.** **If the name shall be globally accessible, it shall be prefixed with a *<scope_prefix> ::= <COMPONENT_NAME>_* (i.e. the *<component_name>_* but with all letters in upper case).**	**C**	**C++**

Example:

```
#define CC_PRE_CONDITION(expr) (...)
#define USING_OBJECT_STORE 1
typedef enum
{
    MONITOR_STATE_IDLE,
    MONITOR_STATE_ACTIVE,
    MONITOR_STATE_INACTIVE,
    MONITOR_STATE_ERROR
} MONITOR_STATES_T;
#define MONITOR_ENABLED (TRUE)
```

Rule 30508 (M)	**Pointer variable names shall have a *<ptr_prefix> ::= p* for each level of indirection.**	**C**	**C++**

Example: Pointer prefixes:

```
FooNode *pFooNode;
FooNode **ppFooNode;
```

Rule 30509 (M)	**Pointers to functions shall have a *<func_ptr_prefix> ::= pf* , with one additional p for each additional level of indirection.**	**C**	**C++**

Example: Pointer prefixes:

```
FuncPointer *pfFooFunc;
FuncPointer **ppfFooFunc;
```

4. Portability related rules

Portability across operating systems and compilers may not be the major concern of a project. But the appearance of new versions of operating systems or C compilers, or the move to another hardware platform might cause some problems if some elementary conventions are not observed.
The following topics without rule number are already mentioned in the other rules.

Avoid the direct use of pre-defined data types in declarations.

Do not assume that an int and a long have the same size and do not assume that an int is a 32 bits long (it may be only 16 bits long on another platform). See rule 6.3 in [1].

An excellent way of transforming your world to a "vale of tears" is to directly use the pre-defined data types in declarations. If it is later necessary, due to portability problems, to change the return type of a function, it may be necessary to make changes at a large number of places in the code. One way to avoid this is to declare a new type name using typedefs to represent the types of variables used. In this way, changes can be more easily made. This may be used to give data a physical unit, such as kilogram or meter. Such code is more easily reviewed. (For example, when the code is functioning poorly, it may be noticed that a variable representing meters has been assigned to a variable representing kilograms). It should be noted that a typedef does not create a new type, only an alternative name for a type. This means that if you have declared typedef int Error, a variable of the type Error may be used anywhere that an int may be used. It is recommended to use PC-lint's support for strong type checking. For example:

```
/*lint -strong(AcJX, Speed)
typedef unsigned long Speed;
```

This means that variables of type *Speed* can be only assigned values of type *Speed*, ignoring assignment of constants. For more information regarding strong type checking in PC-lint, please see chapter 9 of [2].

Be careful not to make pointer conversions from a "shorter" type to a “longer" one. See Rule 21003.
A processor architecture often forbids data of a given size to be allocated at an arbitrary address. For example, a word must begin on an "even" address for MC680x0. If there is a pointer to a char which is located at an "odd" address, a type conversion from this char pointer to an int pointer will cause the program to crash when the int pointer is used, since this violates the processor's rules for alignment of data.

Rule 40001	**(M)**	**Do not assume that you know how an instance of a data type is represented in memory.**	**C**	**C++**

Exception: When using dual ported RAM and memory mapped IO, it is necessary to understand and enforce the reprentation in memory.

Rule 40002	**(M)**	**Do not assume that longs, floats, doubles or long doubles may begin at arbitrary addresses.**	**C**	**C++**

The representation of data types in memory is highly machine-dependent. By allocating data structures to certain addresses, a processor may execute code more efficiently. Code which depends on a specific representation is, of course, not portable.

Rule 40003	**(M)**	**Do not assume that the operands in an expression are evaluated in a definite order.**	**C**	**C++**

Rule 40004	**(M)**	**Do not assume that you know how the invocation mechanism for a function is implemented.**	**C**	**C++**

Rule	**40005**	**(M)**	**Do not assume that static objects are initialized in any special order.**	**C**	**C++**

5. Metrics related rules

This chapter contains rules related to metrics that need to be applied.

5.1. Files

Rule	**50101**	**(R)**	**The maximum number of lines of code per file should be 1500.**	**C**	**C++**

All lines are counted. This includes comments, header and history.

5.2. Functions

Rule	**50201**	**(R)**	**The nesting of blocks should not exceed 5 levels excluding function block.**	**C**	**C++**
Rule	**50202**	**(R)**	**The maximum number lines of code per function should be 250.**	**C**	**C++**
Rule	**50203**	**(R)**	**The maximum number of arguments passed to a function should be 7.**	**C**	**C++**
Rule	**50204**	**(R)**	**The maximum Cyclomatic Complexity of a function should be 30.**	**C**	**C++**
Rule	**50205**	**(R)**	**The maximum number of functions that may be called from one function is 10.** **If the same function is called more than once, this counts as one call.**	**C**	**C++**

5.3. Lines

Rule	**50301**	**(M)**	**The maximum number for statements per line is 1.**	**C**	**C++**

The only exceptions are the for statement, where the initial, conditional, and loop statements may be written on a
single line, and the switch statement where the actions are short and nearly identical.
The if statement is not an exception: the executed statement always goes on a separate line from the conditional
expression(s).
Example:

```
/* The general form of the switch statement is: */
switch (input)
{
case 'a':
...
break;
case 'b':
```

```
...
break;
default:
...
break;
}
/*
* If the actions are very short and nearly
* identical in all cases, an alternate form of
* the switch and if/elseif statement is acceptable:
*/
switch (input)
{
case "a": x = aVar; break;
case "b": x = bVar; break;
default: x = dVar; break;
}
if      (a) { callx(); }
else if (b) { cally(); }
else if (c) { callz(); }
else        { callError(); }

if ((a == b) || (c == d)) e = f; /* wrong, not execution
statement on same line */
if ((a == b) || (c == d))
{
    e = f; /* correct */
}
```

Rule 50302 (M) The maximum number of Variables defined per line is 1. C C++

Example: Definition of several variables in the same statement:

```
/* NOT RECOMMENDED!!! */
INT8 *pI, j; /* pI defines a pointer to char, while j is a char */
/* RIGHT WAY TO DO IT! */
INT8 *pI;
INT8 j;
```

Rule 50303 (M) The maximum number of assignments per line is 1. C C++

Example:

```
a = b = c; /* no multiple assignments allowed */
```

Rule 50304 (M) The maximum number of characters per line is 120. C C++

6. Appendices

6.1. Include file (.h)

```
/**********************************************************************************************/
/**
 * @file            trdp_types.h
 *
 * @brief           Typedefs for TRDP communication
 *
 * @details
 *
 * @note            Project: TRDP prototype stack
 *
 * @author          Bernd Loehr, NewTec GmbH
 *
 * @remarks All rights reserved. Reproduction, modification, use or disclosure
 *          to third parties without express authority is forbidden,
 *          Copyright Bombardier Transportation GmbH, Germany, 2012.
 *
 *
 * $Id: trdp_types.h 5405 2012-03-02 17:23:50Z bloehr $
 *
 */

#ifndef TRDP_TYPES_H
#define TRDP_TYPES_H

/***********************************************************************************************
 * INCLUDES
 */

#include <stddef.h>
#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

/**********************************************************************************************
 * DEFINES
 */

#define MAX_URI_SIZE    102
#define MAX_SOCKET_CNT  80

#ifndef UINT8
#define UINT8   uint8_t
#define UINT16  uint16_t
#define UINT32  uint32_t
#define INT8    int8_t
#define INT16   int16_t
#define INT32   int32_t
#define BOOL    int
#endif

#ifndef TRUE
#define TRUE    (1)
#define FALSE   (0)
#endif

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

```
/** Various flags for PD packets */
typedef enum
{
    TRDP_FLAGS_NONE         = 0,
    TRDP_FLAGS_REDUNDANT    = 0x1,    /**< Redundant                    */
    TRDP_FLAGS_MARSHALL     = 0x2,    /**< Optional mrshalling/unmarhalling in TRDP stack */
    TRDP_FLAGS_CALLBACK     = 0x4     /**< Use of callback function     */
    TRDP_FLAGS_TCP          = 0x8     /**< Use TCP protocol instead of UDP */
} TRDP_FLAGS_T;


/** Various flags general TRDP options */
typedef enum
{
    TRDP_OPTION_NONE            = 0,
    TRDP_OPTION_NON_BLOCK       = 0x1, /**< Non blocking execution    */
    TRDP_OPTION_TRAFFIC_SHAPING = 0x2  /**< Optional traffic shaping  */
} TRDP_OPTION_T;


/** Message Types */
typedef enum
{
    TRDP_MSG_PD = 0x5072,     /**< 'Pr' PD Request                     */
    TRDP_MSG_PR = 0x5064,     /**< 'Pd' PD Data (Reply)                   */
    TRDP_MSG_PE = 0x5065,     /**< 'Pe' PD Error                       */
    TRDP_MSG_MN = 0x4D6E,     /**< 'Mn' MD Notification (Request without reply) */
    TRDP_MSG_MR = 0x4D72,     /**< 'Mr' MD Request with reply             */
    TRDP_MSG_MP = 0x4D70,     /**< 'Mp' MD Reply without confirmation        */
    TRDP_MSG_MQ = 0x4D71,     /**< 'Mq' MD Reply with confirmation           */
    TRDP_MSG_MC = 0x4D63,     /**< 'Mc' MD Confirm                     */
    TRDP_MSG_ME = 0x4D65,     /**< 'Me' MD Error                       */
} TRDP_MSG_T;

/**   Timer value compatible with timeval / select.
 * Relative or absolute date, depending on usage
 */
typedef struct
{
    UINT32  tv_sec;      /**< full seconds                                   */
    UINT32  tv_usec;     /**< Micro seconds (max. value 999999            */
} TRDP_TIME_T;


/**   Message info from received telegram; allows the application to generate responses.
 *
 * Note: Not all fields are relevant for every message type!
 */
typedef struct
{
    UINT32      srcIpAddress;  /**< source IP address for filtering    */
    UINT32      dstIpAddress;  /**< destination IP address for filtering  */
    UINT32      seqCount;      /**< sequence counter                     */
    UINT16      protVersion;   /**< Protocol version                     */
    TRDP_MSG_T  msgType;       /**< Protocol ('PD', 'MD', ...)            */
    UINT32      comId;         /**< ComID                        */
    UINT32      topoCount;     /**< received topocount                 */
    BOOL        subs;          /**< substitution                    */
    UINT16      offsetAddress; /**< offset address for ladder architecture   */
    UINT32      replyComId;    /**< ComID for reply (request only)  */
    UINT32      replyIpAddress;/**< IP address for reply (request only)   */
    TRDP_ERR_T  resultCode;    /**< error code, user stat ???          */
} TRDP_PD_INFO_T;


typedef struct
{
    UINT32              srcIpAddress; /**< source IP address for filtering        */
    UINT32              dstIpAddress; /**< destination IP address for filtering */
    UINT32              seqCount;     /**< sequence counter                       */
    UINT16              protVersion;  /**< Protocol version                       */
    TRDP_MSG_T          msgType;      /**< Protocol ('PD', 'MD', ...)           */
    UINT32              comId;        /**< ComID                          */
    UINT32              topoCount;    /**< received topocount                */
    UINT16              userStatus;   /**< user status code, use 0 for OK    */
    TRDP_REPLY_STATUS_T replyStatus;  /**< reply status                       */
    UINT32[4]           sessionId;    /**< for response                   */
    UINT32              replyTimeout; /**< reply timeout given with the request */
    UINT8               destURI[32];  /**< URI user part from MD header         */
    UINT8               srcURI[32];   /**< from MD header                    */
    UINT32              noOfReplies;  /**< actual number of replies for the request */
    void                *userRef;     /**< User reference given with the local call */
    TRDP_ERR_T          resultCode;   /**< TRDP result code, 0 for OK           */

} TRDP_MD_INFO_T;


struct
{
    UINT8   qos;
    UINT8   ttl;

} TRDP_SEND_PARAM_T


typedef UINT32 TRDP_UUID_T[4];

/* regarding the former info provided via SNMP the following information was left out/can be
implemented additionally using MD:
 *   - task table        name, prio, cycle time (not needed)
 *   - PD subscr table:  ComId, sourceIpAddr, destIpAddr, cbFct?, timout, toBehaviour, counter
 *   - PD publish table: ComId, destIpAddr, redId, redState cycle, ttl, qos, counter
 *   - PD join table:    joined MC address table
 *   - MD listener table: ComId  destIpAddr, destUri, cbFct?, counter
 *   - Memory usage
 */
typedef struct
{
    UINT32 version;             /**< TRDP version  */
    TRDP_TIME_T timeStamp;      /**< actual time stamp */
    UINT32 upTime;              /**< time in sec since last initialisation */
    UINT32 statisticTime;       /**< time in sec since last reset of statistics */
    UINT32 ownIpAddr;           /**< own IP address */
    UINT32 virtualIpAddr;       /**< virtual IP address */
    UINT32 processPrio;         /**< priority of TRDP process */
    UINT32 processCycle;        /**< cycle time of TRDP process in microseconds */
    UINT32 pktRcv;              /**< total number of received packets */
    UINT32 pktRcvCrcErr;        /**< number of received packets with CRC err */
    UINT32 pktRcvProtErr;       /**< number of received packets with protocol err */
    UINT32 pdDefQos;            /**< default QoS for PD */
    UINT32 pdDefTtl;            /**< default TTL for PD */
    UINT32 pdPktRcv;            /**< number of received PD packets */
    UINT32 pdPktRcvTopoErr;     /**< number of received PD packets with wrong topo count */
    UINT32 pdPktRcvNoSubs;      /**< number of received PD packets without subscription */
    UINT32 pdTimeout;           /**< number of PD timeouts */
    UINT32 pdPktSnd;            /**< number of sended PD  packets */
    UINT32 mdDefQos;            /**< default QoS for MD */
    UINT32 mdDefTtl;            /**< default TTL for MD */
    UINT32 mdPktRcv;            /**< number of received MD packets */
    UINT32 mdPktRcvTopoErr;     /**< number of received MD packets with wrong topo count */
    UINT32 mdPktRcvNoListener;  /**< number of received MD packets without listener */
    UINT32 mdReplyTimeout;      /**< number of reply timeouts */
    UINT32 mdConfirmTimeout;    /**< number of confirm timeouts */
    UINT32 mdPktSnd;            /**< number of sended MD packets */
} TRDP_STATISTICS_T;


/********************************************************************************************/
/**   Callback for receiving indications, timeouts, releases, responses.
 *
 *  @param[in] *pRefCon   pointer to user context ???????
 *  @param[in] *pMsg      pointer to received message information
 *  @param[in] *pData     pointer to received data excl. padding and FCS !!!!
 *  @param[in] dataSize   size of received data pointer to received data excl. padding and FCS
 */
typedef void (*TRDP_MD_CALLBACK_T)( void *pRefCon, TRDP_MD_INFO_T *pMsg,
                                    UINT8 *pData, UINT32 dataSize);
typedef void (*TRDP_PD_CALLBACK_T)( void *pRefCon, TRDP_PD_INFO_T *pMsg,
                                    UINT8 *pData, UINT32 dataSize);


/********************************************************************************************/
/**   Callback for receiving indications, timeouts, releases, responses.
 *
 *  @param[in] *pRefCon   pointer to user context
 *  @param[in] *pMsg      pointer to received message information
 *  @param[in] *pData     pointer to received data excl. padding and FCS !!!!
 *  @param[in] dataSize   size of received data pointer to received data
 */
typedef void (*TRDP_MD_CALLBACK_T)( void *pRefCon, TRDP_MD_INFO_T *pMsg,
                                    UINT8 *pData, UINT32 dataSize);


/**********************************************************************************************
*************************/
/**   Callback for receiving indications, timeouts, releases, responses.
 *
 *  @param[in] *pRefCon   pointer to user context
 *  @param[in] *pMsg      pointer to received message information
 *  @param[in] *pData     pointer to received data excl. padding and FCS !!!!
 *  @param[in] dataSize   size of received data pointer to received data
*/
typedef void (*TRDP_PD_CALLBACK_T)( void *pRefCon, TRDP_PD_INFO_T *pMsg,
                                    UINT8 *pData, UINT32 dataSize);



/**********************************************************************************************
*************************/
/**   Function types for marshalling .
 * The function must know about the dataset's alignment etc.
 *
 *  @param[in]     *pRefCon   pointer to user context
 *  @param[in]    comId      ComId to identify the structure out of a configuration
 *  @param[in]    *pSrc      pointer to received original message
 *  @param[in]     *pDst      pointer to a buffer for the treated message
 *  @param[in/out]  *pDstSize  size of the provide buffer / size of the treated message
 *
 *  @retval         TRDP_NO_ERR      no error
 *  @retval         TRDP_MEM_ERR    provide buffer to small
 *  @retval         TRDP_COMID_ERR  comid not existing
 *
 */

typedef TRDP_ERR_T (*TRDP_MARSHALL_T)(
    void        *pRefCon,
    UINT32       comId,
    const UINT8 *pSrc,
    UINT8       *pDst,
    UINT32      *pDstSize);


/******************************************************************************************/
/**   Function types for unmarshalling.
 * The function must know about the dataset's alignment etc.
 *
 *  @param[in]     *pRefCon   pointer to user context
 *  @param[in]      comId      ComId to identify the structure out of a configuration
 *  @param[in]     *pSrc      pointer to received original message
 *  @param[in]     *pDst      pointer to a buffer for the treated message
 *  @param[in/out] *pDstSize  size of the provide buffer / size of the treated message
 *
 *  @retval         TRDP_NO_ERR      no error
 *  @retval         TRDP_MEM_ERR    provide buffer to small
 *  @retval         TRDP_COMID_ERR  comid not existing
 *
 */

typedef TRDP_ERR_T (*TRDP_UNMARSHALL_T)(
    void        *pRefCon,
    UINT32       comId,
    const UINT8 *pSrc,
    UINT8       *pDst,
    UINT32      *pDstSize);



#ifdef __cplusplus
}
#endif

#endif
```

6.2. Implementation file (.c)

```
/*
 * $Id: receiveHello.c 3844 2012-02-17 18:12:40Z bernd $
 */
/******************************************************************************/
/**
 * @file            receiveHello.c
 *
 * @brief           Demo application for TRDP
 *
 * @note            Project: Receiving Demo application for TRDP
 *
 * @author          Bernd Loehr and Florian Weispfenning, NewTec GmbH
 *
 * @remarks All rights reserved. Reproduction, modification, use or disclosure
 *          to third parties without express authority is forbidden,
 *          Copyright Bombardier Transportation GmbH, Germany, 2011.
 *
 */

/*******************************************************************************
 * INCLUDES
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <netinet/in.h>

#include "icontrol_defs.h"
#include "trdp_if_light.h"
#include "trdp_utils.h"
#include "trdp_resolver.h"

#define PD_COMID1       2000
#define PD_COMID1_CYCLE    100
#define PD_COMID1_TIMEOUT  1200

#define PD_COMID2       2001
#define PD_COMID2_CYCLE    100
#define PD_COMID2_TIMEOUT  1200

#define PD_COMID3       2002
#define PD_COMID3_CYCLE    100
#define PD_COMID3_TIMEOUT  1200

uint8_t  gBuffer[32];
```

```
/******************************************************************************/
/** callback routine for receiving TRDP traffic
 *
 *  @param[in]      pCallerRef   user supplied context pointer
 *  @param[in]       pMsg        pointer to message block
 *
 *  @retval         none
 */
void callBack(
    void          *pCallerRef,
    MSG_INFO_T    *pMsg)
{
```