Coding Rules for *FORTE*[*]

Version 1.0

Alois Zoitl[†] Rene Smodic[‡]

25.06.2007

Contents

1 Comments

A sufficient amount of comments has to be written. There are never too many comments, whereas invalid comments are worse than none — thus invalid comments have to be removed from the source code. Comments have to be written in English.

[*] Framework for Distributed Industrial Automation and Control—Run-Time Environment

[†] zoitlacin.tuwien.ac.at

[‡] smodicacin.tuwien.ac.at

Comments for class, function, ... definitions have to follow the conventions of *Doxygen* to allow the automated generation of documentation for the sourcecode.

For documenting the implementation it is allowed to indicate Single-line comments with `//` ahead of the command or in the same line right after the command. All other comments have to be located ahead of the command or block. Generally comments have to be tagged with `//` to allow the temporarily commenting out of code with `/*...*/`. Comments have to be meaningful, to describe to program and to be up to date.

1.1 Fileheaders

Every source-file must contain a fileheader as follows:

```
/*******************************************************************************
 * Copyright (c) 2007 4DIAC - consortium.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *
 * Contributors:
 *    <author>, <author email> - changes
 *******************************************************************************/
```

Each author needs to explain his changes in the code. An example for the fileheader used in an full header file is given in Appendix A.1 of this document.

1.2 Revision History

The revision history has to be done in a style usable by Doxygen. This means that the history is independent of the files, but all classes are documented.

1.3 Keywords

The following Keywords should be used in the source code to mark special comments:

- **TODO:** For comments about possible or needed extensions
- **FIXME:** To be used for comments about potential (or known) bugs

2 Datatypes

The following table contains the definitions of important standard datatypes. This is done to ensure a machine independant defintion of the bit-width of the standard data types. For *FORTE*-development these definitions are in the file: `src/arch/datatypes.h`

3 Naming of Identifiers

Every identifier has to be named in English. The first character of an identifier must not contain underscores (there are some compiler directives which start with underscores and this could lead to conflicts). Mixed case letters has to be used and the appropriate prefixes have to be inserted where necessary.

defined data type	bit-width / description	used C-datatype
T_BYTE	8 bit unsigned	char
T_WORD	16 bit unsigned	unsigned short
T_DWORD	32 bit unsigned	unsigned long
T_INT8	8 bit signed	signed char
T_INT16	16 bit signed	short
T_INT32	32 bit signed	long
T_UINT8	8 bit unsigned	char
T_UINT16	16 bit unsigned	unsigned short
T_UINT32	32 bit unsigned	unsigned long
T_FLOAT	single precission IEEE float (32 bit)	float
T_DFLOAT	double precission IEEE float (64 bit)	double
T_BOOL8	byte variable as boolean value	bool

3.1 Variables

Variables have to be named self explanatory. The names have to be provided with the appropriate prefixes and they have to start with an uppercase letter. In case of combining prefixes, the use of ranges, arrays, pointer, enumerations, or structures is at first, followed by basic data types or object prefixes. The only exception are loop variables (thereby the use of i, j, k is allowed). Only one variable declaration per line is allowed. Pointer operators at the declaration have to be located in front of the variable (not after the type identifier). If possible initializations have to be done directly at the declaration.

Global variables are prohibited!

3.2 Prefixes

The following prefixes have to be applied to identifiers:

Type Definitions

S	for structures
C	for class
I	for interface
E	for enum
T_	for types (e.g. typedef in C++)

Variable Types

a	for arrays
p	for pointers
r	for references
en	for enumerations
st	for structures

Objects

o	for meaningless objects
lst	for list objects
v	for vector objects
s	for string objects

Ranges

m_	for member variables of classes
s_	for static variables
pa_	for function parameters
sm_	static member
csm_	constant static member

Basic Data Types

c	for characters
b	for booleans
n	for integers
f	for all floating point numbers

Examples

```
class CFunctionBlock;
int nNumber;
int *pnNumber = &nNumber;
char cKey;
```

```
bool g_bIsInitialized;
float m_fPi = 3.1415;
int anNumbers[10];
```

3.3 Constants

Constants have to be named with block letters (only upper case letters). If a name consists of more words, underscores for separation are allowed. With C++ it is prohibited to declare constants with the #define statement (const has to be used instead). Constants with a validity exceeding a file have to be centralized in a separate file with the prefix "Const". Never ever use "magic numbers" (e.g. `if (x == 3){...}`). Instead use constants.

4 Classes

In addition to the type–prefix the class identifiers have to start with a capital letter.

4.1 Class Structure

The declaration of the class content has to be done in the following order:

1. Private
2. Protected
3. Public

4.2 Functions/Methods

Function– and method–identifiers have to start with a lower case letter. Functions with a return value of a Boolean type should have a name which points to the result (relate the name to the more likely result) and the name should start with the prefix "is". Set and get methods have to start with the appropriate prefix. Methods which are not modifying the state of the object have to be declared as a const method (keyword const).

4.3 Parameters

Parameters which are keeping their value within a method have to be declared as const parameters.

5 Code Formatting

5.1 Indentation

The tabulator width has to be set to 2. Instead of tabulator characters spaces have to inserted (usually there is an option for this in the IDE called: "replace tabs"). A new block has to be started at the same line as its initial statement. An example is given in the appendix A.2 of this document.

5.2 Blocks

The left parenthesis of a block has to be in the same line as the construct. The right parenthesis has to be in an own line. No single-lined if statements need to be put into two lines (1 for the if statement 1 for the expression). An example how to format blocks is given in the appendix A.2 of this document.

6 Exceptions for IEC 61499 Elements

6.1 Naming of IEC 61499 Objects

All identifiers corresponding to IEC 61499 objecets (ressources) should be named as defined in the IEC 61499 Standard. So they are execepted from the rules in sections 3 to 6. This has two advantages:

- No parsing/substitution of names in the code files is needed
- It helps to differentiate between ”runtime-code” and ”user-code”

A Examples

A.1 File Header

```
/*******************************************************************************
 * Copyright (c) 2007 4DIAC - consortium.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *
 * Contributors:
 *    <author>, <author email> - changes
 *******************************************************************************/
#ifndef _FILENAME_H_
#define _FILENAME_H_
 //! short class description
 /*! long class description
  *  @version: <date>/<author>: <description>
  */
class CFooSpace {
  private:
    //! short member var description
    int m_nIsBar; /*!> long description
                   */
  protected:
    //! short description
    void bar(void); /*!> long description
                     */
  public:
    //! short description
    int foo(void); /*!< long description
                    */
};
#endif
```

A.2 Indention and Blocks

```
int CFooSpace::foo(void){
  if(m_nIsBar){
    bar();
    return 1;
  }
  else
    return 0;
```

```
    if (!m_nIsBar)
        notBar();
}
```