Coding Rules for *FORTE*[*]

Alois Zoitl, alil@users.sf.net, Rene Smodic, smodic@acin.tuwien.ac.at
Martin Melik Merkumians melik-merkumians@acin.tuwien.ac.at

October 19, 2023

Contents

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

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.

> 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 File Headers

Every source-file must contain a file header as follows:

```
/*******************************************************************************
 * Copyright (c) 2007 4DIAC - consortium.
 * This program and the accompanying materials are made available under the
 * terms of the Eclipse Public License 2.0 which is available at
 * http://www.eclipse.org/legal/epl-2.0.
 *
 * SPDX-License-Identifier: EPL-2.0
 *******************************************************************************
```

An example for the file header used in an full header file is given in Appendix A.1 of this document.

1.2 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

For the *FORTE*-development we distinguish between three main kinds of data types:

1. Standard C++ data types:

   These data types should be used in all places where no special demands on the used data type are required. Especially for standard integers the `int` or `unsigned int` should be considered, as these are on most machines the fastest and often also smaller assembler code is produced.

2. IEC 61131-3 data types:

   FORTE provides a set of classes resembling the data types defined in IEC 61131-3. These classes can be found in the `src/core/datatypes` directory. The class names are the IEC 61131-3 data type name prefixed with `CIEC_`. They are used for the FB interfaces and for internal variables of Basic FBs. There they are also used for the transition conditions and the algorithms. When using these data types one should be aware about the overhead involved in them.

3. Data types of given size:

   Table 1 contains the definitions of important standard data types. This is done to ensure a machine independent definition of the bit-width of the standard data types. For *FORTE*-development these definitions are in the file: `src/arch/datatypes.h`

Table 1: Size constrained data types for *FORTE*-development

defined data type	bit-width / description
TForteByte	8 bit unsigned
TForteWord	16 bit unsigned
TForteDWord	32 bit unsigned
TForteInt8	8 bit signed
TForteInt16	16 bit signed
TForteInt32	32 bit signed
TForteUInt8	8 bit unsigned
TForteUInt16	16 bit unsigned
TForteUInt32	32 bit unsigned
TForteFloat	single precission IEEE float (32 bit)
TForteDFloat	double precission IEEE float (64 bit)

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 (i.e. camel-case) have to be used and the appropriate prefixes have to be inserted where necessary.

3.1 Variables

Variables have to be named self explanatory and written in lower camel case (camelCase). If a prefix has to be used, this is the first loweer-case letter, and the name of the variable will start with an upper-case letter (e.g., mMemberVariable). If no prefix has to be used (e.g., local variables), then the variable name will start with a lower-case letter (e.g., localVariable). For loop variables names like i, j, k, or similar are allowed, however if possible a meaningful name is preferred. 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 applicable, variables shall be initialized at their declaration. Additionally, try to minimize the visibilty of a variable to its necessary minimum.

Global non constant 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++)

Scope	
m	for member variables of classes
cm	for a constant member
s	for static variables
pa	for function parameters
sm	static member
scm	static constant member
cg	for a global constant

Examples

```
class CFunctionBlock;
int number;
int *pointer = &number;
char key;
bool gIsInitialized;
float mPi = 3.1415;
int numbers[10];
```

3.3 Constants and Const-Expressions

With C++ it is prohibited to declare constants with the #define statement (const has to be used instead). A prefix cm, scm, or cg, depending on the cope of the constant or constexpr shall be used. Never ever use ”magic numbers” (e.g. `if (x == 3){...}`), always 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. Public
2. Protected
3. Private

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 spaces. 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. For the purpose of indention, the access specifiers `public`, `protected`, and `private` are considered to be blocks, so definitions need to be indented relative to the access specifier 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.

Single-line if statements are not allowed. Curly brackets have to be used for all if, else, else if, for, while statments even when they contain only a single statement.

An example how to format blocks is given in the appendix A.2 of this document.

5.3 if-Statements

Within if-statements you should consider the following rules:

- Put spaces around your operators (e.g., `if(i > 0){`)
- If you have several expressions in an if put parenthesis around each of them in order to avoid ambiguous interpretation of the compiler (e.g., `if((i > 0) && (i < 5)){`).
- Always use curly brackets for the if body, so do `if(i > 0){ return i; }` and not `if(i > 0) return i;`

5.4 Loops

Use the appropriate loop construct for your task. If you have an interator or you are using a plain counting variable with a predefined end, then use a `for` loop. If there is a certain termination condition, use `while(<condition>)` or `do ... while(<condition>)` constructs. If an infinite loop is needed use `while(true)`.

Regardless of what type of loop you use, always use curly brackets for the loop body.

5.5 Eclipse

For users of the IDE Eclipse with the CDT plugin we provide a style file that correctly formats your code to this rules. The file can be found in FORTE's main directory and is called `fortestyle.xml`. This file can be imported into your FORTE project under the Menu Project/Properties and there in the tree element C/C++ General/Code Style. With the FORTE style file you can simple correctly format your file by pressing `<ctrl>+<shift>+f`.

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”

7 Performance and Size Considerations

7.1 Pointers vs. References

If applicable prefer references over pointers. References have no issues with possibly being `nullptr` and its usage is usually simpler.

7.2 Parameter Passing

Again, prefer references over pointers when passing parameters. Additionally, if your data structures are rather small (¡ 16 bytes), then pass by-value, as this is usually more efficient compared to by-reference passing.

7.3 Function Inlining

Our experience showed that functions shorter than 4 to 3 line should be inlined. This nearly always reduces the size of FORTE and therefore should increase its performance. However your mileage may vary. So please use it wisely and make tests and measurements

7.4 Local Static Variables

Do not use local static variables... ever... period.

They introduce all kinds of problems, such as reentrant issues, thread-safety, etc. so do not use them.

A Examples

A.1 File Header

```
/*******************************************************************************
 * Copyright (c) 2007 4DIAC - consortium.
 * This program and the accompanying materials are made available under the
 * terms of the Eclipse Public License 2.0 which is available at
 * http://www.eclipse.org/legal/epl-2.0.
 *
 * SPDX-License-Identifier: EPL-2.0
 *******************************************************************************
#ifndef _FILENAME_H_
#define _FILENAME_H_

//! short class description
/*! long class description
 */
class CFooSpace {
  public:
    //! short description
    int foo(void); /*!< long description
```

```
                     */
  protected:
    //! short description
    void bar(void); /*!> long description
                      */
  private:
    //! short member var description
    int mIsBar; /*!> long description
                    */
};

#endif
```

A.2 Indention and Blocks

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

  if(!mIsBar){
        notBar();
  }

  return 0;
}
```