AA 050 Coding Guidelines

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Prepared: |  | Claudio Nold |  | Unterschrift in den Akten |  | signed: 2017-10-25 |  |
|  |  | System Architect |  | Signature |  | yyyy-mm-dd |  |
| Verified: |  | Christoph Kündig |  | Unterschrift in den Akten |  | signed: 2017-10-25 |  |
|  |  | System Architect |  | Signature |  | yyyy-mm-dd |  |
| Approved: |  | Rico Zoss |  | Unterschrift in den Akten |  | signed: 2017-10-25 |  |
|  |  | Head of Development |  | Signature |  | yyyy-mm-dd |  |
| Für dieses Dokument und den darin dargestellten Gegenstand behalten wir uns alle Rechte vor. Vervielfältigungen, Bekanntgabe an Dritte oder Verwertung seines Inhaltes sind ohne unsere ausdrückliche Zustimmung verboten. | | |  | Identity Number: |  | AX903000001 |  |
|  | Version: |  | 00 |  |
|  | Status: |  | Freigegeben |  |
|  | Date (yyyy-mm-dd): |  | 2017-10-25 |  |

Table of Contents

[1 Profile 7](#_Toc496701156)

[2 Acknowledgements 7](#_Toc496701157)

[3 Terms and Definitions 7](#_Toc496701158)

[4 Overview 7](#_Toc496701159)

[5 Coding Standards 8](#_Toc496701160)

[5.1 General Recommendations 8](#_Toc496701161)

[Recommendation 01: Don’t optimize 8](#_Toc496701162)

[Rule 02: Incomplete guidelines 8](#_Toc496701163)

[5.2 Source Files and Project Structure 8](#_Toc496701164)

[Rule 03: .h 8](#_Toc496701165)

[Rule 04: .sig 8](#_Toc496701166)

[Rule 05: .cpp /.c 8](#_Toc496701167)

[Rule 06: Naming files 8](#_Toc496701168)

[Recommendation 07: One class per file, one file per class 9](#_Toc496701169)

[Rule 08: “include” directory 9](#_Toc496701170)

[Rule 09: “source” directory 9](#_Toc496701171)

[Rule 10: Directory structure and groups of classes 9](#_Toc496701172)

[Recommendation 11: “doc” directory 9](#_Toc496701173)

[5.3 Header Files 10](#_Toc496701174)

[Rule 12: Include guards 10](#_Toc496701175)

[Recommendation 13: Forward declarations 10](#_Toc496701176)

[Rule 14: Relative and absolute include paths 10](#_Toc496701177)

[Rule 15: Include everything needed 10](#_Toc496701178)

[Rule 16: Own header file 11](#_Toc496701179)

[Rule 17: #include “ownfile.h” 11](#_Toc496701180)

[Rule 18: #include <systemfile.h> 11](#_Toc496701181)

[Rule 19: Unnecessary include files 11](#_Toc496701182)

[5.4 Classes 11](#_Toc496701183)

[Rule 20: Virtual destructor 11](#_Toc496701184)

[Rule 21: Initialize order 11](#_Toc496701185)

[Rule 22: Virtual functions in constructor 12](#_Toc496701186)

[Recommendation 23: Global objects in constructor 12](#_Toc496701187)

[Rule 24: Const functions 12](#_Toc496701188)

[Recommendation 25: Keep inheritance simple 12](#_Toc496701189)

[Recommendation 26: Function overloading 12](#_Toc496701190)

[Rule 27: No circular dependencies 12](#_Toc496701191)

[5.5 Arguments and Return Types 13](#_Toc496701192)

[Rule 28: No functions with anonymous or variadic arguments 13](#_Toc496701193)

[Recommendation 29: Number of arguments 14](#_Toc496701194)

[Recommendation 30: Avoid passing pointers as arguments 14](#_Toc496701195)

[Rule 31: Const and passed-by-reference 14](#_Toc496701196)

[Rule 32: Argument names 15](#_Toc496701197)

[Rule 33: (removed) 15](#_Toc496701198)

[Rule 34: Local variables 15](#_Toc496701199)

[Rule 35: Const member return 15](#_Toc496701200)

[5.6 Constants 15](#_Toc496701201)

[Rule 36: Use ‘const‘ keyword 15](#_Toc496701202)

[Rule 37: No numeric values – “magic numbers” 16](#_Toc496701203)

[Rule 38: No std::string constants 16](#_Toc496701204)

[5.7 Variables 17](#_Toc496701205)

[Rule 39: Smallest possible scope 17](#_Toc496701206)

[Rule 40: Initialize variables before use 17](#_Toc496701207)

[Recommendation 41: Prefer assignment over initialization 17](#_Toc496701208)

[5.8 Pointers, References and Resources 18](#_Toc496701209)

[Rule 42: Allocate memory with AF\_NEW, free memory with SAFE\_DELETE 18](#_Toc496701210)

[Rule 43: Pointer usage 18](#_Toc496701211)

[Rule 44: Check passed pointers 18](#_Toc496701212)

[Rule 45: Initialize pointers with NULL but don’t compare to NULL 19](#_Toc496701213)

[Recommendation 46: Avoid pointers to pointers 19](#_Toc496701214)

[Recommendation 47: Avoid function pointers 19](#_Toc496701215)

[Rule 48: All memory must be freed 19](#_Toc496701216)

[5.9 Type Conversions 19](#_Toc496701217)

[Rule 49: Only use C++ type conversions 19](#_Toc496701218)

[Rule 50: Type conversion rules 20](#_Toc496701219)

[5.10 Flow Control 20](#_Toc496701220)

[Rule 51: switch 20](#_Toc496701221)

[Rule 52: switch/default 21](#_Toc496701222)

[Rule 53: No ‘goto’ 21](#_Toc496701223)

[Recommendation 54: break 21](#_Toc496701224)

[Rule 55: Use of “{ }” 21](#_Toc496701225)

[5.11 Error Handling 21](#_Toc496701226)

[Recommendation 56: Exception vs. return value 21](#_Toc496701227)

[Recommendation 57: Use exceptions only for errors 22](#_Toc496701228)

[Rule 58: No exceptions used as ‘goto’ 22](#_Toc496701229)

[Rule 59: Derive from CgeneralException 22](#_Toc496701230)

[Rule 60: Destructor never throws 22](#_Toc496701231)

[Recommendation 61: The constructor never throws 22](#_Toc496701232)

[Recommendation 62: Throw by value / catch by reference 22](#_Toc496701233)

[Rule 63: No exception specification 23](#_Toc496701234)

[Rule 64: Catch all exceptions 23](#_Toc496701235)

[5.12 Portability 24](#_Toc496701236)

[Rule 65: Endianness and which data types to choose 24](#_Toc496701237)

[Recommendation 66: Use caution when using plain ‘char’ 25](#_Toc496701238)

[5.13 Miscellaneous 26](#_Toc496701239)

[Recommendation 67: Prefer C++ STL 26](#_Toc496701240)

[Rule 68: String handling only with length checking functions 26](#_Toc496701241)

[Recommendation 69: Minimize number of temporary objects 26](#_Toc496701242)

[Recommendation 70: Avoid long and complex functions 26](#_Toc496701243)

[Recommendation 71: Implement access functions inline 27](#_Toc496701244)

[Recommendation 72: Limit the use of inline functions 27](#_Toc496701245)

[Recommendation 73: Avoid ‘friend’ declarations 28](#_Toc496701246)

[Recommendation 74: Define templates with caution 28](#_Toc496701247)

[Recommendation 75: Avoid global objects 28](#_Toc496701248)

[6 Style Standards 29](#_Toc496701249)

[6.1 Whitespaces 29](#_Toc496701250)

[Rule 76: Separate elements by two empty lines 29](#_Toc496701251)

[Rule 77: Line Endings and the last line 29](#_Toc496701252)

[Recommendation 78: No space with unary operators 29](#_Toc496701253)

[Rule 79: Connect \* and & to the type, not to the name 30](#_Toc496701254)

[Rule 80: Indentation and use of tabs/spaces 30](#_Toc496701255)

[6.2 Names 33](#_Toc496701256)

[Rule 81: Avoid global scope 33](#_Toc496701257)

[Rule 82: Variables and functions: start with lowercase letter 33](#_Toc496701258)

[Rule 83: Class names: prefix ‘C’ 33](#_Toc496701259)

[Rule 84: Use camel-case 33](#_Toc496701260)

[Rule 85: Global names: don’t start with underscores 33](#_Toc496701261)

[Rule 86: No underscores in names 33](#_Toc496701262)

[Rule 87: Avoid similar type names 33](#_Toc496701263)

[Rule 88: Member variables start with ‘m\_’, ‘k\_’ or ‘s\_’ 34](#_Toc496701264)

[Rule 89: Names of constants, enumerations and macros 34](#_Toc496701265)

[Recommendation 90: Use self-descriptive names 35](#_Toc496701266)

[Recommendation 91: Names should be pronounceable 35](#_Toc496701267)

[Recommendation 92: Avoid not generally accepted abbreviations 35](#_Toc496701268)

[Rule 93: Use of the Hungarian Notation 35](#_Toc496701269)

[6.3 Classes 35](#_Toc496701270)

[Rule 94: Order of definition: members, methods, inner classes 35](#_Toc496701271)

[Rule 95: Order of sections: public, protected, private 35](#_Toc496701272)

[Rule 96: Names of accessors and mutators: “get” and “set” 36](#_Toc496701273)

[Rule 97: Template keyword 36](#_Toc496701274)

[6.4 Expressions 36](#_Toc496701275)

[Recommendation 98: Use parentheses 36](#_Toc496701276)

[Recommendation 99: Prefer ++it over it++ 37](#_Toc496701277)

[Rule 100: No combined assignment / comparison. No assignment in sub-expression 37](#_Toc496701278)

[6.5 Namespaces 37](#_Toc496701279)

[Rule 101: Namespaces 37](#_Toc496701280)

[Rule 102: ‘using namespace’ declaration 38](#_Toc496701281)

[6.6 Miscellaneous 38](#_Toc496701282)

[Rule 103: One declaration per line 38](#_Toc496701283)

[Rule 104: Integer types usage 38](#_Toc496701284)

[7 Commentary Standards 40](#_Toc496701285)

[7.1 Comments 40](#_Toc496701286)

[Rule 200: Include an introductory comment 40](#_Toc496701287)

[Rule 201: (removed) 40](#_Toc496701288)

[Rule 202: Language: English 40](#_Toc496701289)

[Rule 203: Write comments for classes, functions, etc. 40](#_Toc496701290)

[Rule 204: Comment: Duplicate in header / source 40](#_Toc496701291)

[Rule 205: Use the Doxygen format 40](#_Toc496701292)

[Rule 206: Blocks of overloaded methods 42](#_Toc496701293)

[Rule 207: Use ‘//’ for comments within methods 43](#_Toc496701294)

[Recommendation 208: Write helpful comments 44](#_Toc496701295)

[Rule 209: Commented code 44](#_Toc496701296)

[Rule 210: Mark up comments 46](#_Toc496701297)

[8 Appendix A: Hungarian Notation 47](#_Toc496701298)

[8.1 Type Prefixes 47](#_Toc496701299)

[8.2 Variable and Object Prefixes 47](#_Toc496701300)

[8.3 Container 48](#_Toc496701301)

[9 Appendix B: List of Common Abbreviations 50](#_Toc496701302)

[10 Appendix C: List of Rules Applying to .sig Files 51](#_Toc496701303)

[11 Referenced Documents 52](#_Toc496701304)

[12 Revision History 52](#_Toc496701305)

[13 Review Group 52](#_Toc496701306)

Table of Figures

- No named figures in this document -

Table of Tables

- No named table in this document -

# Profile

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Summary: | This document specifies rules and recommendations for coding, style and commentary standards for OFRIS C++, and, where applicable, C source code. | | | |
|  |  | | | |
| Scope: | This document applies to OFRIS C++ source code developed by the ANNAX Group and its suppliers. | | | |
|  |  | | | |
| Input data: | Software requirements and design | | | |
|  |  | | | |
| Output data: | Source code and source code documentation | | | |
|  |  | | | |
| Process owner: | Development | | | |
|  |  | | | |
| Impact on occupational safety: | No |  | Impact on environment | No |

# Acknowledgements

The rules and recommendations in this document are based on the Coding Guidelines by Applied Informatics found on the Internet at <http://www.appinf.com/download/CppCodingStyleGuide.pdf>

# Terms and Definitions

The following terms and definitions are used in this document:

OFRIS Object oriented Framework for RAILVOX Information Systems; C++ framework used in software of ANNAX passenger information systems

Rule A rule must be followed. If for some reason this is not possible and an exception is made, this must be clearly documented in the code and, if applicable, in the design documents.

Recommendation A recommendation should be followed whenever this is possible with reasonable effort.

# Overview

This document is organized as follows:

* Section 5 - Coding Standards
* Section 6 - Style Standards
* Section 7 - Commentary Standards
* Appendix A: Hungarian Notation
* Appendix B: List of Common Abbreviations

# Coding Standards

## General Recommendations

Recommendation 01: Don’t optimize

Optimize code only if you know for sure that you have a performance problem. Think twice before you begin. Then **measure** and think again. C++ compilers are pretty good at optimizing these days. So, source code that “looks fast” does not necessarily produce faster object code. It only is harder to understand and maintain.

As C.A.R. Hoare once said: “*Premature optimization is the root of all evil.”*

Rule 02: Incomplete guidelines

If these guidelines leave anything unclear, see how it has been done in the existing code base of the OFRIS framework and do it accordingly.

## Source Files and Project Structure

Rule 03: .h

Header (include) files in C++ always have the file name extension ".h".

Rule 04: .sig

OFRIS signal definition files always have the file name extension ".sig". They may only contain C code with includes, structs, enums, defines, unions and typedefs.

Note: The list of rules applying to signal definition files can be found in “Appendix C: List of Rules Applying to .sig Files”.

Rule 05: .cpp /.c

Implementation files in C++ always have the file name extension ".cpp" and C implementation files always have the file name extension “.c”.

Rule 06: Naming files

Always give a file a name that is unique in as large a context as possible.

A header file for a class must have the file name of the form <class name **without prefix “C”**> + extension. Use uppercase and lowercase letters in the same way as in the source code.

Also see rule 10 for how to group multiple files and in which scope they have to be unique.

Example: File names according to the class name:

* Rio2Driver.h
* Rio2Driver.cpp

namespace fahel

{

class **CRio2Driver** : public CProcess,…

Recommendation 07: One class per file, one file per class

As a general recommendation, a file should only contain the definition and implementation of a single class. Exceptions are:

* Inner classes should be in the same file as the outer class.
* Abstract interface definitions do not have to be in separate files.
* A file may contain many classes of a very similar kind, e.g. implementing communication protocol objects.

The implementation of a class should not be spread over several files.

Rule 08: “include” directory

*Public* header files always go into the include directory called “*include*”. Header files which are only used by the module itself (= non-public) go into the “source” directory.

Rule 09: “source” directory

All implementation files and non-public header files go into the *“source”* directory.

Rule 10: Directory structure and groups of classes

All project files (except CMake file) and other support files not containing source code go into the project directory.

Files which belong together have to be grouped and are stored in a subdirectory of “include” and “source”. The criteria for grouping files cannot be generally defined and each case has to be discussed within the project team or with the system architects. Only one additional directory level is allowed below “include” and “source”. Use camel-case naming beginning with an uppercase letter.

Example of GeneralDeviceDriver (base class from which several drivers are derived):

OfrisLibModules/include/**GeneralDeviceDriver**/AxisCameraDriver.h

OfrisLibModules/source/**GeneralDeviceDriver**/AxisCameraDriver.cpp

Modules which use the header file in the example use the relative path:

#include "**GeneralDeviceDriver**/AxisCameraDriver.h"

Note: The namespace rules (rules 101 and 102) allow having multiple classes with the same name but in different namespaces. Prefixes in file and class names which have previously been used for grouping (like “Sst” or “Mmi”) may no longer be used.

Files which cannot be grouped remain in the “include” and “source” directory without a sublevel.

Recommendation 11: “doc” directory

If there is extra documentation for a project (e.g. specifications or standard documents), this documentation goes into a directory named “*doc”*.

Example for a public library:

LibSstDezKis/include

LibSstDezKis/source

LibSstDezKis/doc

Example for a typical product:

ProdLion/2.x/doc

ProdLion/2.x/ProdLionDriver/source

ProdLion/2.x/ProdLionUnits/source

## Header Files

Rule 12: Include guards

Every header file must contain a mechanism that prevents multiple inclusions of the file.

#ifndef RIO2DRIVER\_H

#define RIO2DRIVER\_H

…

#endif

Recommendation 13: Forward declarations

Definitions of classes that are only accessed via pointers (\*) or references (&) shall not be included as header files. Forward declarations shall be used instead.

namespace fahel

{

// forward declaration

class CViewConditionHandler; // #include “ViewConditionHandler.h” is not needed

class CAbstractViewClient

{

public:

// …

virtual CViewConditionHandler\* getViewConditionHandler() = 0;

};

Rule 14: Relative and absolute include paths

Never specify relative paths containing “.” And “..” in *#include* directives.

Never specify absolute include paths.

Bad example:

#include “../interfaces/AffectRes.h”

Rule 15: Include everything needed

Every implementation file (.cpp) must include the relevant files that contain:

* Declarations of types and functions used in the functions that are implemented in the file.
* Declarations of variables and member functions used in the functions that are implemented in the file.
* Exception: do not repeat #includes declared in the header file of the same class.

#include <map> // for map<string, CGroup\*>::iterator

#include <string> // for map<string, CGroup\*>::iterator

// include header file of “CGroup”

#include <sstream> // for ostringstream

// “endl” needs <iostream> but we have <sstream> already which is enough

void CSstMatrixDisplayDriver::display()

{

map<string, CGroup\*>::iterator itGroup;

ostringstream osGroup;

for(itGroup = m\_mapGroups.begin(); itGroup != m\_mapGroups.end(); ++itGroup)

{

osGroup << “GROUP “ << itGroup->first << endl;

}

}

Rule 16: Own header file

Every implementation file must, before any other header files, include its corresponding header file. This ensures that header files are self-sufficient.

Rule 17: #include “ownfile.h”

Use the directive *#include* “*filename.h*” for user-prepared include files.

Rule 18: #include <systemfile.h>

Use the directive *#include* <*filename.h*> for system and compiler-supplied header files only.

Rule 19: Unnecessary include files

Unnecessary include files must be omitted. If a file is included in an include file, then every implementation file that includes the second include file must be re-compiled whenever the first file is modified. A simple modification in one include file can make it necessary to re-compile a large number of files.

## Classes

Rule 20: Virtual destructor

All classes that have at least one virtual function, must define a virtual destructor.

class CSstIntercomDriver

{

public:

CSstIntercomDriver(const char\* szParamSet, const char\* szName);

**virtual** ~CSstIntercomDriver();

**virtual** void initialize(); **// virtual function -> requires a virtual destructor**

};

Rule 21: Initialize order

Define and initialize member variables in the same order. Prefer initialization to assignment in constructors.

CSstIntercomDriver::CSstIntercomDriver(const char\* szParamSet, const char\* szName)

: m\_pxResHndLLInt(NULL), // good: initialization

…

{

pxResHndLLInt = NULL; // bad: assignment instead of initialization

Rule 22: Virtual functions in constructor

Do not call a virtual function from a constructor or destructor.

CSstIntercomDriver::CSstIntercomDriver(const char\* szParamSet, const char\* szName)

{

initialize(); // bad: initialize() is virtual -> may lead to problems

Recommendation 23: Global objects in constructor

Avoid the use of global objects in constructors and destructors as they may not yet be initialized.

Rule 24: Const functions

Declare member functions that do not affect the state of an object (its instance variables) as const; typically a “getter” or similar functions. If there is a mutex required to protect the read-only data, declare the mutex member variable as “mutable”.

class CDiagnosticsModule

{

private:

**mutable** systec\_oss::CMutexLock m\_xMutexLock;

public:

virtual const char\* getModuleTypeName() **const** // function does not change anything

{

CMutexHandler xMutexHandler(m\_xMutexLock); // create a lock for other threads

…

return m\_szTypeName;

}

Recommendation 25: Keep inheritance simple

Avoid inheritance for parts-of relations. Prefer composition to inheritance.

A common mistake is to use multiple inheritance for parts-of relations (when an object consists of several other objects, these are inherited instead of using instance variables). This can result in strange class hierarchies and less flexible code. In C++ there may be an arbitrary number of instances of a given type; if inheritance is used, direct inheritance from a class may only be used once.

Example: If a notebook is composed of a CPU, a hard drive, a keyboard and a screen, store the parts as variables of the class and don’t inherit from them.

Recommendation 26: Function overloading

When overloading functions or operators, all variations should have the same semantics (be used for the same purpose).

Overloading of functions can be a powerful tool for creating a family of related functions that only differ as to the type of data provided as arguments. If not used properly (such as using functions with the same name for different purposes), they can, however, cause considerable confusion.

Template functions can be a good alternative to overloading.

Rule 27: No circular dependencies

Avoid circular dependencies, e.g. include, type or class dependencies.

Two or more classes, which depend on each other, are forbidden. This also applies to inherited classes. Keep the dependencies simple: A -> B and not A -> B -> A for instance. For such dependencies, use the common OFRIS “Client” / “ClientInt” architecture. Exceptions to this rule are inner classes which may access to the outer object.

Bad example:

// File A.h

class B;

class A

{

public:

unsigned char\* m\_rgData;

void do(B\* pxB);

};

void A::do(B\* pxB)

{

pxB->display(); **// class A accesses to class B**

}

// File B.h

#include <A.h>

class B

{

A\* m\_pxA; // this is a pointer to class A

public:

void display();

};

void B::display()

{

sendRequest(m\_pxA::m\_rgData); **// Problem: class B accesses to class A**

};

## Arguments and Return Types

Unless otherwise stated, the following rules apply to member functions as well as functions without a class.

Rule 28: No functions with anonymous or variadic arguments

Do not use unspecified function arguments (ellipsis notation). Do not use anonymous arguments.

class CSstIntercomDriver

{

public:

CSstIntercomDriver(const char\*, const char\*); // bad example: argument names missing

setOptions(int32\_t a, ...);

Recommendation 29: Number of arguments

Avoid functions with too many (rule of thumb: more than five) arguments. Functions having long argument lists look complicated, are difficult to read, and can indicate poor design. In addition, they are difficult to use and to maintain.

Recommendation 30: Avoid passing pointers as arguments

Prefer references over pointers. Use a pointer only if the argument could be null or if using a reference would always require dereferencing a pointer when calling the function. If you have to pass a pointer, describe very precisely, who the owner of the object is and how long it’s valid. See also the Rule 43 which is related to this recommendation.

{

// avoid the following situation – or at least, describe it very carefully

CHandler xHandler(**AF\_NEW** CData(“text”)); // is a reference possible here?

}

Rule 31: Const and passed-by-reference

The following function argument handling is mandatory:

1. Read-only function arguments have the keyword ‘const’. This ensures that the argument cannot be modified within the method
2. All classes or arrays are passed by reference (“non-POD data types”)
3. Writable function arguments are also passed by reference, even if it’s a simple (“POD”) data type
4. Avoid passing pointers (see Recommendation 30)
5. A function that changes the value of a pointer that is provided as an argument should declare the argument as having the type reference to pointer (e.g. *char\*&*).

Good examples:

bool readFile(**const** std::string**&** strFileName, char\* rgBufResult, int32\_t**&** iSizeResult);

void parseString(**const** char cSep, **const** int32\_t iIdx, std::vector<std::string>**&** vecArgs);

**Const Keyword Order**

Writing “const SParam\* pxParamStruct“ has the same effect as “SParam const\* pxParamStruct”: The object pointed to by pxParamStruct cannot be modified.

We prefer having the “const” keyword *first*.

You may also add a second “const” after the “\*” sign. The effect is that the pointer itself may not be changed within the method, e.g. in

bool constExample(**const** SParam\* **const** pxParamStruct);

pxParamStruct cannot be deleted within the method.

Recommendation: Writing two “const” would be the safest way to avoid any overwriting but gives very long argument lines. So it is allowed to just write the first “const” to show that an argument is read only even it is not the safest way.

Rule 32: Argument names

The names of formal arguments to functions must be specified and must be the same both in the function declaration and in the function definition.

The names of formal arguments may be specified in both the function declaration and the function definition in C++, even if the compiler ignores those in the declaration. Providing names for function arguments is a part of the function documentation. The name of an argument clarifies how the argument is used and reduces the need to include comments in a function definition. It is also easier to refer to an argument in the documentation of a class if it has a name.

Rule 33: (removed)

Rule 34: Local variables

A public function must never return a reference or a pointer to a local object.

If a function returns a reference or a pointer to a local variable, the memory to which it refers will already have been deallocated when this reference or pointer is used. The compiler may or may not give a warning for this.

string& CClass::getName() const

{

string strName = “HEADER”;

return strName; // bad: a local variable is returned -> undef. Behavior

}

Rule 35: Const member return

A public member function must never return a non-const reference or pointer to member data.

A public member function must never return a non-const reference or pointer to data outside an object, unless the object shares the data with other objects.

By allowing a user direct access to the private member data of an object, these data may be changed in ways not intended by the class designer. This may lead to reduced confidence in the designer’s code: a situation to be avoided.

class CName

{

private:

std::string strName;

public:

// Return the name as const, otherwise everyone can write into

// the private member!

**const** std::string& getName() const { return strName; }

## Constants

Rule 36: Use ‘const‘ keyword

Constants in C++ are defined using const; never using *#define*.

The preprocessor performs a textual substitution for macros in the source code that is then compiled. This has a number of negative consequences. For example, if a constant has been defined using *#define*, the name of the constant is not recognized in many debuggers. If the constant is represented by an expression, this expression may be evaluated differently for different instantiations, depending on the scope of the name. In addition, macros are, at times, incorrectly written.

Rule 37: No numeric values – “magic numbers”

Avoid the use of numeric values in code; use constants instead (“enums” or “const int32\_t/float/…”). Exceptions: 0, 1.

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 a portability point of view, absolute values may be the cause of more subtle problems. The type of a numeric value is dependent on the implementation. Normally, the type of a numeric value is defined as the smallest type able to hold the value.

Magic literals which do not appear multiple times for the same purpose, can be marked with “// ML: ” and explained on the same line like in the following example:

}

else if((iArgCount == 2) && (arg[0] == “trainno”)) // ML: argument 2: train number expected

{

“// ML: “ marks an exception for the static code analysis. Be careful explaining the reason for the magic literal – the reason for the given number is only clear to the writer, so nonsense comments like “// ML: two” are not acceptable.

Rule 38: No std::string constants

Use C strings for constant strings instead of C++ strings, because the order of initialization is undefined if someone from the outside reads the constant. You can never be sure that the string class has been initialized. Besides, std::string consumes much more memory than really needed.

Example:

class CProperty

{

public:

static const std::string k\_strDefaultValue1; // **not ok**

static const char\* k\_szDefaultValue2; // **ok**

}

const std::string CProperty::k\_strDefaultValue1 = “Default”;

const char\* CProperty::k\_szDefaultValue2 = “Default”;

// different source file

class CConstantUser

{

public:

// fails if called before initialization of string constant:

std::string getRandomConstant() { return CProperty::k\_strDefaultValue1; }

// works correctly:

const char\* getCorrectConstant() { return CProperty::k\_szDefaultValue2; }

}

// different source file – which translation unit will be constructed first?

static CConstantUser s\_xUser1;

std::string strName = s\_xUser1.getRandomConstant(); // may crash depending on the order

Note: Static variables in different translation units are constructed in an undefined order. This is so terrible it has its own name: the static initialization order fiasco. This affects all classes but is mainly a problem of std::string constants.

## Variables

Rule 39: Smallest possible scope

Variables are declared with the smallest possible scope.

A variable ought to be declared with the smallest possible scope to improve the readability of the code and so that variables are not unnecessarily allocated. When a variable that is declared at the beginning of a function is used somewhere in the code, it is not easy to directly see the type of the variable. In addition, there is a risk that such a variable is inadvertently hidden if a local variable, having the same name, is declared in an internal block.

Many local variables are only used in special cases which seldom occur. If a variable is declared at the outer level, memory will be allocated even if it is not used. In addition, when variables are initialized upon declaration, more efficient code is obtained than if values are assigned when the variable is used.

Example:

// do not declare variables outside of the loop, if not required

ostringstream osOutput;

for(vector<int32\_t >::const\_iterator it = vecIds.begin(); it != vecIds.end(); ++it)

{

// declare variables in this scope, if they are only used in the loop

**int32\_t** iVal = \*it;

osOutput << (iVal – 1) << “,”;

}

Rule 40: Initialize variables before use

Every variable must be initialized before use.

Normally, the compiler gives a warning if a variable is uninitialized. It is then sufficient to take care of such cases. Instances of a class are usually initialized even if no arguments are provided in the declaration (the empty constructor is invoked).

Recommendation 41: Prefer assignment over initialization

Prefer assignment (int32\_t iLine = 0;) instead of initialization (int32\_t iLine(0);).

Reason: Compilers have been optimizing this for a long time and the assignment is better readable.

## Pointers, References and Resources

Rule 42: Allocate memory with AF\_NEW, free memory with SAFE\_DELETE

Always allocate memory with the macro AF\_NEW and free it with SAFE\_DELETE or SAFE\_ARR\_DELETE. Do not use *malloc*( ), *realloc*( *), calloc( ), delete( )* or *free*( ). The SAFE\_DELETE macro will make sure the pointer is not NULL and will assign NULL after freeing the memory.

CDigitalInputInt\* pxDin = AF\_NEW CDigitalInputInt(…);

SAFE\_DELETE(pxDin);

Rule 43: Pointer usage

Never save a pointer passed by a constructor or a method from outside as a member in the own class. The lifetime of a pointer is difficult to track and leads to segmentation faults.

Exceptions to that rule are the following special cases:

* Inner classes may save a pointer to the outer object
* Pointer to client interfaces (“call-back”) may be saved
* Pointers used for logging (CextMsgLoopInt)
* Pointers used for message handling (CmsgLoopInt)
* Pointers handled by the shared\_ptr class

Bad example:

class CData;

class CBadExample

{

private:

CData\* m\_pxData; // Oops, a pointer is stored here

public:

void saveData(CData\* pxData)

{

m\_pxData = pxData; // Bad example: avoid storing a pointer

}

};

main()

{

CBadExample xExample;

CData xData;

xExample.saveData(&xData); // Very dangerous!

// What is the lifetime of xData?

}

Rule 44: Check passed pointers

Always check if a pointer passed as an argument is not NULL before using it. An exception to this rule is the OFRIS signal handling pointer CExtMsgLoopInt (if it really has to be a pointer instead of a reference).

Void CExample::doSomething(const CData\* pxData)

{

if(!pxData)

{

// Report an error if this is really an error

return;

}

if(pxData->empty()) …

}

Rule 45: Initialize pointers with NULL but don’t compare to NULL

Always initialize a pointer with *NULL*.

Do not compare a pointer to *NULL* when used in a Boolean expression.

CDigitalInputInt\* pxInput = NULL;

…

if(pxInput)

{

…

}

Recommendation 46: Avoid pointers to pointers

Pointers to pointers should whenever possible be avoided. Use std::vector instead and pass it in function arguments by reference.

Recommendation 47: Avoid function pointers

Avoid function pointers. Better use OO concepts for that.

Rule 48: All memory must be freed

All objects allocated with AF\_NEW must be freed. Special attention must be given in situations where exceptions are thrown:

try

{

CDigitalInputInt\* pxDin = AF\_NEW CDigitalInputInt(…);

if(bErrorOccured)

{

throw CGeneralException();

}

SAFE\_DELETE(pxDin);

}

catch(…)

{

**// memory leak!**

}

## Type Conversions

Rule 49: Only use C++ type conversions

Use C++ style casts (*dynamic\_cast<>, static\_cast<>, reinterpret\_cast<>, const\_cast<>*) instead of old-style C casts for all pointer conversions. This makes type conversions more explicit.

*Note:* This rule only defines which type casts are replacing the C style casts. Rule 50 defines when to use which type!

|  |  |  |
| --- | --- | --- |
| Usage | C++ cast | old-style C cast – don’t use |
| Type cast | EType eType = **static\_cast**<EDisplayElementType>(i); | EType eType = (EDisplayElementType)i; |
| Inheritance cast | **dynamic\_cast**<CDSDigitalInput\*> (it->second)->setAutoMode(xTmpLst); | (CDSDigitalInput\*)(it->second) ->setAutoMode(xTmpLst); |
| Constness | **const\_cast**<char\*>(szName) | (char\*)szName |
| Unclear data types | **reinterpret\_cast**<int32\_t\*>(rgBuf[0]); | (int32\_t\*)rgBuf[0] |

Rule 50: Type conversion rules

The use of casts often leads to design weaknesses, therefore consider improving the interfaces and try to avoid type conversions. If a type conversion cannot be avoided, the following rules must be followed:

* Use **static\_cast** wherever possible – but **only for basic datatypes** like int8/16/32, float and char and not for casting any classes. Doing so may crash the running process.
* Use reinterpret\_cast only in low-level code interfacing other systems (for instance a serial port communication)
* dynamic\_cast is not allowed. It is expensive and may even throw exceptions or invoke an undefined behavior – anything may happen. Ask the system architects if you still plan to use it.
* Removing the constness by using const\_cast is not allowed. Ask the system architects if you still plan to do so.

Example of the usage of reinterpret\_cast:

void CMessageDecoder::parseBuffer(const uint8\_t\* rgBuf, const uint32\_t uiBufLen,

std::vector<int32\_t>& vecParsedValues)

{

// To parse a received buffer, you have no other chance

// than to cast the buffer. Avoid little/big endian problems!

uint32\_t uiNameLen = reinterpret\_cast<uint32\_t\*>(rgBuf[0]);

// check uiNameLen according to uiBufLen for avoiding

// memory access violations

// parse the buffer here…

}

## Flow Control

Rule 51: switch

The code following a case label must always be terminated by either a *break* statement, a *return* statement or a *throw* statement.

Multiple case statements directly following each other (‘fall through’) are allowed.

In exceptional cases and if this simplifies the code significantly ‘fall throughs’ are allowed even with (few) statements between the case statements.

The comment “// fall through“ is mandatory in case that the ‘break’ statement was omitted intentionally.

Example:

**switch**(eDestState)

{

**case** *eDSt\_StateUnready*: // fall through

bFlag = true;

**case** *eDSt\_StateReady*:

m\_xOut.changePriority(**false**);

**break**;

**default**:

**break**;

}

Rule 52: switch/default

A switch statement must always contain a default branch that handles unexpected cases. The *default* label is always the last label in a *switch* statement.

Rule 53: No ‘goto’

Never use goto.

*Goto* breaks the control flow and can lead to code that is difficult to comprehend. In addition, there are limitations for when *goto* can be used. For example, it is not permitted to jump past a statement that initializes a local object having a destructor.

Recommendation 54: break

Use break to exit a loop if this helps to avoid the use of flags.

Rule 55: Use of “{ }”

Use “{ }” in compound statements and if-else constructs.

// not allowed:

if(bState == true)

executeCmd();

nextStep();

// correct:

if(bState == true)

{

executeCmd();

}

nextStep();

## Error Handling

Recommendation 56: Exception vs. return value

Exceptions should only be used if this simplifies the code *for the caller*. Keep in mind that for each exception the caller will need to have a catch statement. In some cases reporting an error may be sufficient.

class CTimeoutInt

{

public:

void cancel()

{

throw systec\_oss::CTimeoutException();

}

// Note: every caller has to catch exceptions! Keep classes easy to use…

Recommendation 57: Use exceptions only for errors

Do not throw an exception for things which are “normal” and happen often – for instance a lost connection or any user interaction.

Rule 58: No exceptions used as ‘goto’

Do not use exceptions for flow control within a method (throw and catch in the same method). In fact, this is a “goto”.

Try

{

throw CGeneralException(); // bad example: this is a goto

}

catch(CGeneralException& e)

{

…

}

Exception: It is allowed to catch an exception locally to report an error and to re-throw the exception.

Try

{

throw CGeneralException();

}

catch(CGeneralException& e)

{

reportError();

throw;

}

Rule 59: Derive from CgeneralException

Only throw exceptions that are derived from CGeneralException or one of its subclasses.

Rule 60: Destructor never throws

Destructors, deallocation and swap must never fail (throw exceptions).

Recommendation 61: The constructor never throws

Constructors should never fail. Reason: An exception thrown by the constructor of a member object cannot be caught.

Recommendation 62: Throw by value / catch by reference

Always throw by value and catch by reference.

Rule 63: No exception specification

Do not use exception specifications (unless you are forced to). Reason: If a function throws an exception which is not specified, the program will be terminated.

Document all potentially thrown exceptions in the Doxygen comment.

Example:

/\*\*

\* Does nothing.

\* @exception CBadMoodException If in bad mood.

\* @exception CGeneralException If doing nothing fails.

\*/

void lazyfunc()

{

if(m\_bBadMood)

{

throw CBadMoodException();

}

doNothing(); // may throw a CGeneralException

}

Bad example:

void func1(const int32\_t i) throw(CException1) **// NOT CException2!**

{

if(i == 1)

{

throw CException2(“Error 2”); **// cannot be caught -> program terminates!**

}

}

int main(int argc, char\* argv[])

{

try

{

func1(argc-1);

}

catch(CException1& e)

{

cout << “Exception: “ << e.what() << endl;

}

catch(CException2& e) **// program terminates anyway!**

{

cout << “Exception: “ << e.what() << endl;

}

}

Rule 64: Catch all exceptions

Catch all CGeneralException and all exceptions which derive from it.

Catching std::exception is not generally required. Consider the following criteria:

* Within interfaces to other systems (“outside of OFRIS”) you have to carefully catch all exceptions which may occur – also when using vector and other STL classes. This ensures the robustness and protection of the software against attacks over the interface.
* For all other cases you can let the runtime system handle the exception which leads to a program termination and a reboot later. This has the advantage that a callstack backtrace can be generated which helps to find the source of the problem.
* Catching std::exception on a high level (e.g. in the method run()) hides the origin of the exception and is thus counterproductive in finding the source.

## Portability

Rule 65: Endianness and which data types to choose

Use common techniques from the framework for solving endianness problems (htonl() etc., ZDR/XDR, existing Macros).

Portability Rule: Always use the original’s data type – in many cases this is size\_t

Notes:

* If you compare an ‘int’ on a 64bit system to string::npos it could not match like expected
* ‘size\_t’ is dynamic and could be a 32bit or a 64bit integer – let the system decide
* ‘ssize\_t’ is only a POSIX standard and not a C++ standard so try to avoid it if possible

BAD: (u)int32\_t iPos = str.find(‘,’); // Note: str is of type std::string or CStr

GOOD: size\_t uiPos = str.find(‘,’);

BAD: (u)int32\_t iSize = str.size();

GOOD: size\_t uiSize = str.size();

BAD: int32\_t iWritten = ::send(m\_iSocketFd, pxDataBufCnt, iRemaining, 0);

GOOD: ssize\_t iWritten = ::send(m\_iSocketFd, pxDataBufCnt, iRemaining, 0);

Portability Rule: Do not use ‘long’ and ‘long long’

Notes:

* ‘long’ is 32 bit or 64 bit and has not the same length like an ‘int’ which is always 32 bit even on a 64 bit system

Strategy:

1. For database keys use the new type “dbkey\_t” which will be set to int32\_t after migration
2. For resource IDs use the new type “resid\_t” which will be set to int32\_t after migration
3. Use “long” if you are forced to – for instance for a library call
4. But never use “long” for data exchange in an interface; this will break the compatibility on switching to 64 bit
5. “long long” is misleading too. Only use it if you are forced to. Otherwise use “int64\_t”.

BAD: long lActionId;

GOOD: dbkey\_t iActionId;

BAD: void resetResource(long lResId) {}

GOOD: void resetResource(resid\_t iResId) {}

**Portability Rule: use ‘%zu’ and ‘%zd’ in printf()/REPORTMSG format string**

Notes:

* ‘%ld’ and ‘%lu’ is only for 32 bit
* size\_t can be 32 bit or 64 bit long

BAD: REPORTMSG(... “size of string: **%lu**\n”, str.size()); // size\_t is expected 32 bit

GOOD: REPORTMSG(... “size of string: **%zu**\n”, str.size()); // size\_t could be 32 bit or 64 bit

BAD: REPORTMSG(... “substring size: **%ld**\n”, str.find(...)-str.find(...)); // difference is expected 32 bit (< 0 is possible)

GOOD: REPORTMSG(... “substring size: **%zd**\n”, str.find(...)-str.find(...)); // difference could be 32 bit or 64 bit (< 0 is possible)

Portability Rule: Never *implicitly* cast from and to long

BAD: int32\_t iId = lId;

GOOD: int32\_t iId = static\_cast<int32\_t>(lId); // lId always fits into an int32 because...

Portability Recommendation: Never implicitly cast signed numbers to unsigned numbers and vice versa

It is not very obvious to compare for instance an INVALID\_ID (=-1) to an unsigned number. Are you sure it will behave like you would expect?

BAD: if(uiVal == INVALID\_ID) // INVALID\_ID is -1 -> expected behavior?

GOOD: if(iVal == INVALID\_ID) // better: keep original *signed* integer

BAD: uint32\_t uiVal = iValue; // implicit cast

GOOD: uint32\_t uiVal = static\_cast<uint32\_t>(iValue); // iValue is never < 0 because...

Recommendation 66: Use caution when using plain ‘char’

See “Rule 104: Integer types usage”. Be aware that the treatment of the type “char” is platform dependent. The standard does not specify if plain “char” is signed or unsigned. Use static\_cast for omitting platform specific behavior:

Example:

char rgBuf[] = { 0xff };

int32\_t iLen = rgBuf[0]; // Attention: is iLen == -1 or 255?

// better do it this way – if you can’t change the type of rgBuf:

uint32\_t uiLen = static\_cast<uint8\_t>(rgBuf[0]); // uiLen is 255

// best way: declare uint8\_t

uint8\_t rgBuf[] = { 0xff };

uint32\_t uiLen = rgBuf[0]; // uiLen is 255

## Miscellaneous

Recommendation 67: Prefer C++ STL

Prefer the use of STL (Standard Template Library) classes versus old C types.

Rule 68: String handling only with length checking functions

Only use C library string functions with length check. For example do not use strcpy, sprintf, strcat, etc.. When using strncpy, the terminating zero must be re-added manually.

Void setGroup(const std::string& strGroup)

{

strncpy(m\_szGroup, strGroup.c\_str(), eDcG\_DisplayMaxGroupLen); // good: length check

m\_szGroup[eDcG\_DisplayMaxGroupLen-1] = ‘\0’; // good: set terminating zero

}

Recommendation 69: Minimize number of temporary objects

Minimize the number of temporary objects that are created as return values from functions or as arguments to functions.

Temporary objects are often created when objects are returned from functions or when objects are given as arguments to functions. In either case, a constructor for the object is first invoked; later, a destructor is invoked. Large temporary objects make for inefficient code. In some cases, errors are introduced when temporary objects are created. It is important to keep this in mind when writing code. It is especially inappropriate to have pointers to temporary objects, since the lifetime of a temporary object is undefined.

Void setName(const std::string& strName)

{

m\_strName = strName;

}

const char\* szName = „C String“;

setName(szName); // A temporary std::string will be created here.

Recommendation 70: Avoid long and complex functions

Avoid long and complex functions.

Avoid long or complex constructors. Use init() methods instead.

Long functions have disadvantages:

1. If a function is too long, it can be difficult to comprehend. Generally, it can be said that a function should not be longer than 150 lines.
2. If an error situation is discovered at the end of an extremely long function, it may be difficult for the function to clean up after itself and to “undo” as much as possible before reporting the error to the calling function. By always using short functions, such an error can be more exactly localized.

Complex functions are difficult to test. If a function consists of 15 nested if statements, then there are 2^15 (or 32768) different branches to test in a single function.

Nesting of control structures should not exceed 4 levels.

Note for OFRIS debug methods: separate each debug command and pass it to a method which handles the debug function. The method prefix is “debug” followed by the function name.

Note for specific OFRIS methods: the following callback methods may become large, but not complex so static code analysis is disabled for them:

* + newParameter()
  + newAliasParameter()
  + newState()
  + lostResource()

It is explicitly allowed to leave methods with more than one return statement in order to reduce the complexity of a method. The validity checks should be done at the beginning and when everything is as expected, only one return statement at the end of the method should appear.

bool CAsnParser::parseBuffer(const uint8\_t\* rgBuf, const size\_t uiSize)

{

if(!rgBuf || (uiSize < eASN\_HeaderSize))

{

return false; // this return decreases the complexity of the method

}

if(!rgBuf[0] != k\_u8Pattern)

{

return false; // this return decreases the complexity of the method

}

// everything is as expected, start processing…

// from here, only the return statement at the end is expected

return true;

}

Recommendation 71: Implement access functions inline

Access functions (accessors) that simply return the value of a member variable should be inline. See rule 35 for an example.

Recommendation 72: Limit the use of inline functions

Only use inline functions when it is really necessary and when they only contain a small number of statements.

Inline functions have the advantage of often being faster to execute than ordinary functions. The disadvantage in their use is that the implementation becomes more exposed, since the definition of an inline function must be placed in an include file for the class, while the definition of an ordinary function may be placed in its own separate file.

As a result changing the implementation of an inline function will require re-compiling all files that include the changed file.

The compiler is not forced to actually make a function inline. The decision criteria for this differ from one compiler to another.

class CData

{

unsigned char\* m\_rgBuffer;

int32\_t m\_iLength;

public:

// Note: inline function or in source file? Keep the line simple if it’s in the

// header file; otherwise implement it in the source file (.cpp)

int32\_t copy(const CData& xOtherData) { memcpy(m\_rgBuffer, …); return m\_iLength; }

Recommendation 73: Avoid ‘friend’ declarations

Friend declarations should be avoided.

class CNetBoxDriver

{

class CStateMachineNetBoxDriver: public CStateMachine

{

**friend** class CNetBoxDriver; // avoid ‘friend’ if possible

};

**friend** class CStateMachineNetBoxDriver; // avoid ‘friend’ if possible

}

Recommendation 74: Define templates with caution

Be careful when defining C++ templates. Ask the system architects before doing so.

template<class T>

class SortAlgorithm

{

…

template<typename U>

JSONNode listToJson(const std::list<U>& lstList, const std::string& strNodeName)

{

// implementation directly in header file required (inline)!

}

};

Recommendation 75: Avoid global objects

Avoid global objects at all.

# Style Standards

## Whitespaces

Rule 76: Separate elements by two empty lines

In a header or implementation file, introductory comment, include guards, #include block, using block and function definitions are separated by two empty lines. This makes it easier to visually distinguish the various elements.

#include “DtmfService.h”

#include <sstream>

#ifndef DTMF\_SIG\_SWITCH

#define DTMF\_SIG\_SWITCH

#endif

#include “signals.h”

**// Empty line 1**

**// Empty line 2**

// Namespaces

using namespace fahel;

using namespace systec\_oss;

using namespace std;

**// Empty line 1**

**// Empty line 2**

//-------------------------------------------------------------------------

// State machine

//-------------------------------------------------------------------------

/\*\*

\* Constructor

\* @param xOut Outer object.

\* @param xMsgLoop Reference to the message loop to use

\*/

CDtmfService::CDtmfStateMachine::CDtmfStateMachine(CDtmfService& xOut, CExtMsgLoopInt& xMsgLoop)

: CStateMachine(xMsgLoop, MAX\_STATES, MAX\_TRANS, MAX\_RULES),

m\_xOut(xOut)

{

}

**// Empty line 1**

**// Empty line 2**

Rule 77: Line Endings and the last line

New files must be created with Linux style line endings. Existing files should not be converted to Linux style line endings to preserve the white space comparability of the source file revisions.

The last line in an implementation or header file must be terminated by a newline. This is required by the C++ standard, but not enforced by all compilers.

Recommendation 78: No space with unary operators

Do not use spaces around “ .” or “->”, nor between unary operators and operands (“-1”, “++i”, “!bFlag”).

Rule 79: Connect \* and & to the type, not to the name

The dereference operator “\*” and the address-of operator “&” must be directly connected with the type names in declarations and definitions.

The characters “\*” and “&” must be written together with the types of variables instead of with the names of variables in order to emphasize that they are part of the type definition. Instead of saying that \*i is an int32\_t, say that i is an *int32\_t\**.

uint8\_t**\*** m\_rgBuf; // correct (the type is in fact a pointer)

uint8\_t **\***m\_rgBuf; // not correct

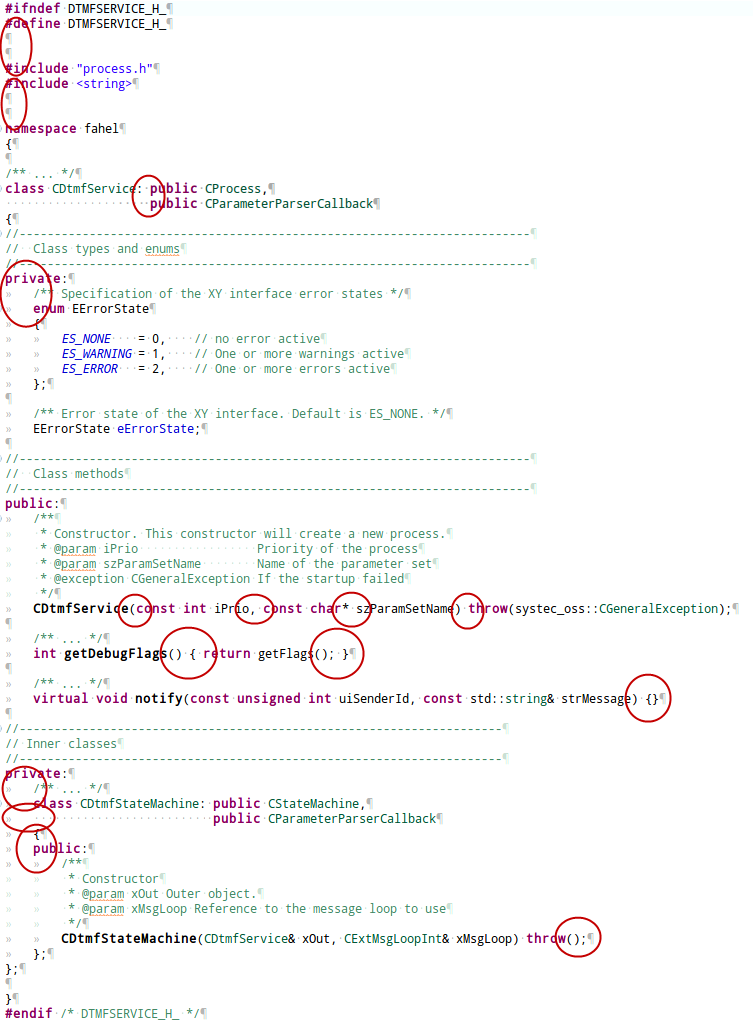
Rule 80: Indentation and use of tabs/spaces

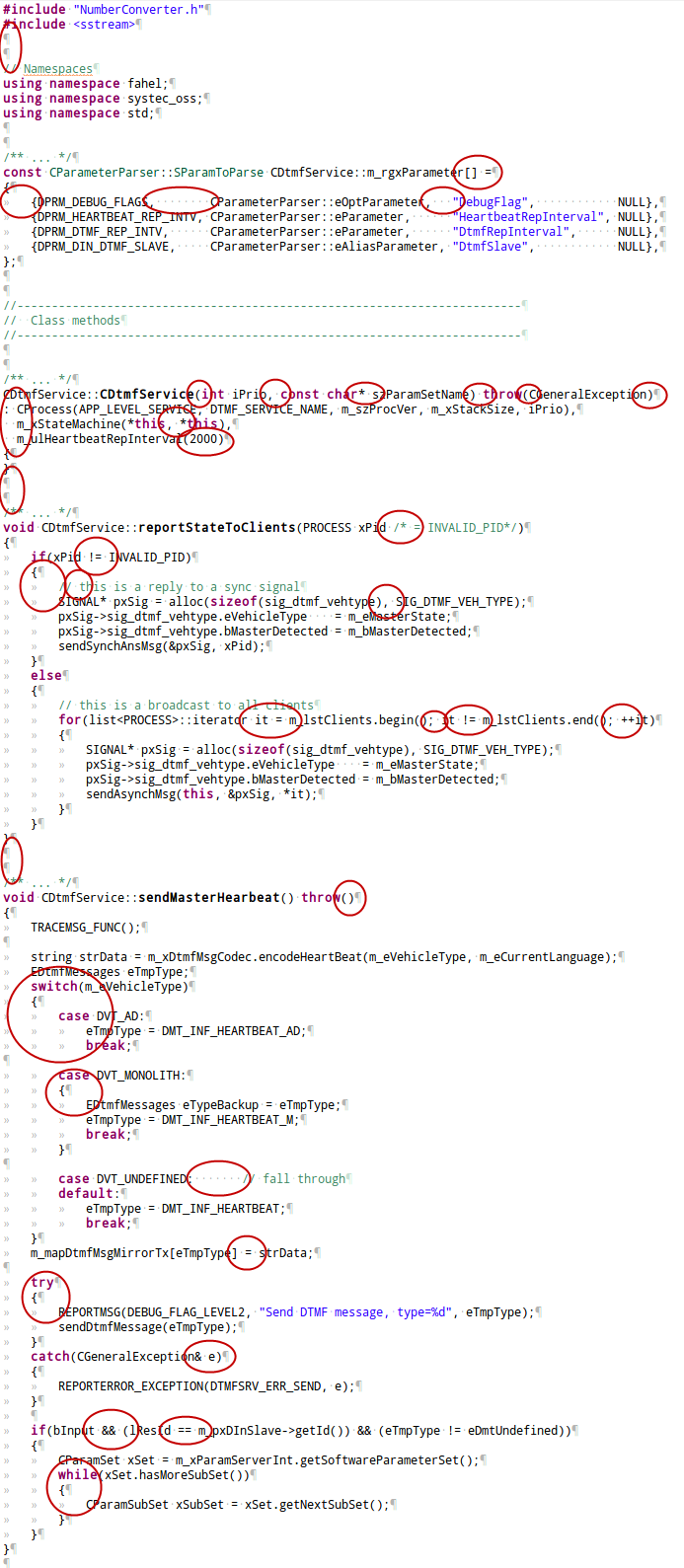
The indentation at the beginning of the line is done using tabs.

The content of each new block started with an opening bracket (“{“) is indented with an additional tab. Exception: No indentation must be done after the namespace declaration.

All other whitespaces must use spaces, not tabs.

See the following example for the whitespaces. Note: The example is *only* significant for whitespaces!





## Names

Rule 81: Avoid global scope

Avoid globally visible classes, enumeration types, type definitions, functions, constants, and variables outside of classes. Also avoid constants and enums in namespace only (for instance “fahel”).

namespace fahel

{

enum EGlobalState **// Avoid this: the enum is in „quasi-global” namespace**

{

eGS\_Unknown = 0,

eGS\_NotReady = 1

}

}

Rule 82: Variables and functions: start with lowercase letter

The names of variables and functions must start with a lowercase letter.

Rule 83: Class names: prefix ‘C’

The names of classes must have the prefix ‘C’. Also see rule 92 and Appendix A: Hungarian Notation.

Rule 84: Use camel-case

In names that consist of more than one word, the words are written together and each word that follows starts with an uppercase letter (“camel-case”). This also applies to abbreviations, for example “CTcpServer” or “CUdpListener”.

Rule 85: Global names: don’t start with underscores

Global names must not start with one or two underscores (“\_” or “\_\_”). This is forbidden by the C++ standard as such names are reserved for use by the compiler.

Rule 86: No underscores in names

With the exception of constants (#define and enums) and the prefixes in rule 89 there are no underscores in names.

Rule 87: Avoid similar type names

Do not use type names (including class names) that differ only by the use of uppercase and lowercase letters.

enum EUseCase

{ … }

enum EUsecase //!! Not allowed! Differs from EUseCase only by case!

{ … }

Rule 88: Member variables start with ‘m\_’, ‘k\_’ or ‘s\_’

Member variables always begin with the following prefixes:

| Prefix | Member Type | Example |
| --- | --- | --- |
| k\_ | Constant members  *Attention: also look at Rule 38 about string constants* | const char\* k\_szTypeName; |
| s\_ | Static non-const members  “k\_” has a higher priority than “s\_” | static std::string s\_strDataVersion; |
| m\_ | All other members  Does not apply to members of a struct | EerrState m\_eErrState;  CtimeoutInt m\_xTimeoutTrapSubscribe;  struct SmyStruct  {  int32\_t iIndex;  } |

Rule 89: Names of constants, enumerations and macros

The names of enumeration values have the prefix ‘e’ and are in camel-case. It is recommended to use an additional unique prefix directly following the ‘e’. This helps to prevent mixing different enum types especially when they are passed to an integer variable. The prefix is separated from the rest of the name with an underscore.

The names of macros and constants, which are outside of classes – if really needed – are written in uppercase style. If the name consists of more than one word, these words are separated by underscore characters. Example: “AVOID\_ANY\_MACROS\_AND\_GLOBALS”

The following example shows the constants and enumeration styles:

class CFzpfInterface

{

/\*\* Specification of the XY interface error states \*/

enum EErrorState

{

eFzES\_None = 0, // no error active

eFzES\_Warning = 1, // one or more warnings active

eFzES\_Error = 2, // one or more errors active

};

/\*\* Error state of the XY interface. Default is eFzES\_None. \*/

EErrorState m\_eErrorState;

}

class CName

{

/\*\* Maximal name size constant including ‘\0’ \*/

static const int32\_t k\_iNameSize;

};

/\*\* Maximal name size constant including ‘\0’ \*/

const int32\_t CName::k\_iNameSize = 20;

const CParameterParser::SParamToParse CDtmfService::k\_rgParameter[] =

{

{DPRM\_DEBUG\_FLAGS, CParameterParser::eOptParameter, „DebugFlag“, NULL}

}

Recommendation 90: Use self-descriptive names

Names should be self-descriptive yet as brief as possible. Cryptic abbreviated names are as hard to read as too long ones. Avoid names which could be easily confused with a similar name. Avoid the use of general names. Be specific.

Example: A class name “CConnection” or “CData” is too unspecific.

Recommendation 91: Names should be pronounceable

Names should be pronounceable. It is hard to discuss something that cannot be pronounced.

Recommendation 92: Avoid not generally accepted abbreviations

Names should not include abbreviations that are not generally accepted. A list of generally accepted abbreviations can be found in Appendix B: List of Common Abbreviations.

Rule 93: Use of the Hungarian Notation

Use the Hungarian Notation for variable and type names as specified in Appendix A: Hungarian Notation.

Loop indexes do not require names after the type prefix like the following example shows:

// “ui” as loop index does not need a name after the prefix

for(size\_t ui = 0; ui < strName.size(); ++ui) {…}

// The iterator variable “it” in the loop does not need a name after the prefix

for(vector<CItem>::const\_iterator it = vecIds.begin(); it != vecIds.end(); ++it)

{

size\_t\_t ui = it->strName.size(); // **prefix only is not allowed here**

}

## Classes

Rule 94: Order of definition: members, methods, inner classes

Keep the following order in class definitions (examples in rules 205 and 206):

1. type definitions, enums, structs and inner classes
2. members
3. methods

Exception: for large inner classes (e.g. state machines) the definition may be placed at the bottom because they tend not to be of interest to most readers. If there are members depending on those classes, they may be placed afterwards.

Rule 95: Order of sections: public, protected, private

The public, protected, and private sections of a class are declared in that order (the *public* section is declared before the *protected* section which is declared before the *private* section).

By placing the public section first, everything that is of interest to a user of the class is gathered in the beginning of the class definition. The protected section may be of interest to designers when considering inheriting from the class. The private section contains details that should be of least general interest.

Derived class methods from interfaces (callbacks) must be declared ‘protected’ and not ‘public’.

See rules 205 and 206 for a complete examples.

Rule 96: Names of accessors and mutators: “get” and “set”

If a member variable has an accessor or a mutator, the accessor name has a *get* prefix and the mutator name has a *set* prefix.

class CProperty

{

private:

std::string m\_strName;

std::string m\_strValue;

public:

CProperty(const std::string& strName);

CProperty(const CProperty& xProp);

CProperty& operator = (const CProperty& xProp);

const std::string& **get**Name() const;

void **set**Value(const std::string& strValue);

const std::string& **get**Value() const;

private:

CProperty();

};

Rule 97: Template keyword

The template keyword, together with the template argument list, is written on a separate line before the following class or function definition.

## Expressions

Recommendation 98: Use parentheses

Use parentheses to clarify the order of evaluation for operators in expressions.

There are a number of common pitfalls having to do with the order of evaluation for operators in an expression. Binary operators in C++ have associativity (either leftwards or rightwards) and precedence. If an operator has leftwards associativity and occurs on both sides of a variable in an expression, then the variable belongs to the same part of the expression as the operator on its left side.

In doubtful cases, parentheses are always to be used to clarify the order of evaluation.

Another common mistake is to confuse the assignment operator and the equality operator. Since the assignment operator returns a value, it is entirely permitted to have an assignment statement instead of a comparison expression. This, however, most often leads straight to an error.

C++ allows the overloading of operators, something that can easily become confusing. For example, the operators << (shift left) and >> (shift right) are often used for input and output. Since these were originally bit operations, it is necessary that they have higher priority than relational operators. This means that parentheses must be used when outputting the values of logical expressions.

Good: if((iTimeDiffMs > eNbT\_MinTimeDiffMS) && (iTimeDiffMs < eNbT\_MaxTimeDiffMS))

bad: if(iTimeDiffMs > eNbT\_MinTimeDiffMS && iTimeDiffMs < eNbT\_MaxTimeDiffMS)

good: os << “Execution failed: “ << (eReturnState != eRS\_Ok) << endl;

Recommendation 99: Prefer ++it over it++

Prefer prefix increment/decrement (++it) of iterators over the postfix variants (it++).

Iterators overload the increment and decrement operators. Using the postfix increment/decrement operators on an iterator causes the construction of a temporary object. Therefore prefer the prefix operators.

Rule 100: No combined assignment / comparison. No assignment in sub-expression

Do not call a function, assign the return value and compare the result in an if() statement all at the same time.

Do not use assignments in sub-expressions.

Bad examples:

if((iIndex = getIndex()) == -1) …

doSomething(iIdx = 0);

## Namespaces

Rule 101: Namespaces

The main namespaces to be used are “fahel” and “systec\_oss”. “fahel” remains the top-most namespace and helps to distinguish between OFRIS and external libraries. As specified in Rule 10, a new namespace must be defined when grouping classes that belong to the same entity but are spread over multiple files. Examples:

OfrisLibModules/include/**GeneralDeviceDriver**/AxisCameraDriver.h -> example: “gdd”

OfrisLibModules/include/**Infotainment**/InfotainmentPlaylist.h -> example: “infot”

ProdMinkDrivers/source/**C30Services**/PcuService.h -> example: “c30services”

ProdMinkDrivers/source/**C30Drivers**/TcmsDriver.h -> example: “c30drivers”

New namespace names have to be unique within OFRIS and its products and have to be clear enough but should not be longer than 10 characters. They consist of lowercase letters and numbers but nothing else. Discuss a new name within the project team or with system architects. Only one namespace level additionally to “fahel” is allowed.

Namespace definitions in headers are nested and do not indent an additional level:

namespace fahel

{

namespace c30drivers

{

…

class CTcmsDriver …

Rule 102: ‘using namespace’ declaration

Do not use “using namespace” declarations in a header file.

In .cpp files, the ‘using namespace’ declaration must be placed after the *#include* statements.

“using namespace” declarations are allowed only for the following namespaces:

* “fahel”, “systec\_oss” and “std”
* The own namespace as declared in the header file

No other “using namespace” declarations are allowed. Example:

using namespace fahel;

using namespace systec\_oss;

using namespace std;

using namespace fahel::c30drivers; // = own namespace

// no other declarations allowed

## Miscellaneous

Rule 103: One declaration per line

Never declare more than one variable on the same line.

Bad example:

uint32\_t iSize, iPos = 0;

Rule 104: Integer types usage

The built-in integer types like ‘int’, ‘short’, ‘char’ etc. must not be used. They are insufficient, because their actual sizes are implementation defined and may vary across different systems. Use the following types instead:

* Types defined in stdint.h (int8\_t, etc.).
* Types defined in the STL or in the C/C++ libraries (time\_t, size\_t, etc.)

Exception: The built-in type ‘char’ may be used for C-strings, e.g. ‘char szText[ePR\_TextLen]’. If a single character of a C-string is extracted and treated as a numeric value, int8\_t or uint8\_t must be used. The corresponding prefix “i8” or “u8” must be used with the variable name to indicate the size (see 8.2).

Example:

char szText[ePR\_TextLen];

snprintf(szText, sizeof(szText), "%s", strMyTxt.c\_str());

szText[ePR\_TextLen-1] = '\0';

char cThirdCharacter = szText[ePR\_ThirdPos]; // used as visible character

printf(“the 3rd character is ‘%c’”, cThirdCharacter);

...

size\_t uiPos = m\_strName.find(cToken);

uint8\_t u8AsciiValue = szText[ePR\_ThirdPos]; // used for some kind of numeric operation

int16\_t i16MyCounter = 0x1234;

int32\_t iMyCounter = 0x12345678; // no prefix for 32 or 64 bit variables

# Commentary Standards

## Comments

Rule 200: Include an introductory comment

Every file that contains source code must be documented with an introductory comment that contains copyright information and provides information on the file name and its content. The description consists of the context and the purpose of the file content.

Template:

//=============================================================================

// Copyright (C) <current year> ANNAX Schweiz AG

//

// Name <file name without extension>

//

// Description <description>

//

// Author <initial author: first name, last name>, <optional: external company>

//=============================================================================

Example:

//=============================================================================

// Copyright (C) 2018 ANNAX Schweiz AG

//

// Name AnnActionHandler

//

// Description Action handler for executing announcement actions. Receives

// action lists and plays them on the audio backend.

//

// Author Hans Muster, Muster AG

//=============================================================================

Rule 201: (removed)

Rule 202: Language: English

Generally use the English language (comments, any names, commit comments, release notes, ...).

Rule 203: Write comments for classes, functions, etc.

Write a comment for every class, function, member variable, *enum/typedef/struct* and members of structs. The standardization of comments makes it possible to automatically generate reference documentation from source code. This is used to keep source code and documentation together up-to-date.

Rule 204: Comment: Duplicate in header / source

All functions have to be documented identically in the header **and** the source files (copies).

Rule 205: Use the Doxygen format

Comments have to comply with the following example to be parsed by Doxygen:

/\*\*

\* The abstract base class of all directory generators. Explain something

\* about its environment (input from ... output to ...). Describe things inheriting

\* classes have to be aware of.

\*/

class CDirectoryGenerator

{

//-------------------------------------------------------------------------

// Class types and enums

//-------------------------------------------------------------------------

private:

/\*\*

\* Enumeration of all events which ...

\*/

enum EAAHEvent

{

eDES\_None = 0, // no error active

eDES\_Warning = 1, // One or more warnings active

eDES\_Error = 2, // One or more errors active

}

/\*\*

\* Stores announcement actions.

\*/

struct SAnnAction

{

/\*\* The announcement action's id. \*/

long lActionId;

...

};

//-------------------------------------------------------------------------

// Class data members

//-------------------------------------------------------------------------

private:

/\*\* Reference to the message loop interface \*/

CExtMsgLoopInt& m\_xMsgLoop;

//-------------------------------------------------------------------------

// Class methods

//-------------------------------------------------------------------------

public:

/\*\*

\* Constructor

\* @param xMsgLoop Reference to the message loop

\* @param pxADDServer Pointer to the ADD server

\*/

CDirectoryGenerator(CExtMsgLoopInt& xMsgLoop, CADDServer\* pxADDServer);

//---------------------------------------------------------------------

// Implementation of CDSDigitalInputHandlerClientInt

//---------------------------------------------------------------------

protected:

/\*\*

\* Returns the appropriate digital input resource object.

\* @param lResId Resource id.

\* @param lClientId Client id.

\* @param lResIntId Unambiguous identifier of the appropriate

\* resource interface at the client-side.

\* @param xOwnerInfo Owner info.

\* @return Appropriate digital input resource.

\* @exception CNotFoundException If the resource id is invalid or the client is not owner.

\*/

CDSDigitalInput& getDigitalInput(long lResId, long lClientId, long lResIntId, CResourceInfoPoolReadAccess::SOwnerInfo& xOwnerInfo);

//---------------------------------------------------------------------

// Inner classes

//---------------------------------------------------------------------

private:

/\*\*

\* Stores all necessary informations on about a resource.

\*/

class CResInfo

{

private:

/\*\* Stores whether the auto reset feature is activated. \*/

bool m\_bAutoReset;

public:

/\*\*

\* Determines whether the auto reset feature is active.

\* @return True if the auto reset feature is active.

\*/

bool isAutoReset() const { return m\_bAutoReset; }

};

};

Rule 206: Blocks of overloaded methods

Blocks of overloaded methods or implemented interfaces are separated as shown in the following example:

/\*\* ... \*/

class CDtmfService : public CProcess,

public CParameterParserCallback

{

//-----------------------------------------------------------------------------

// Class data members

//-----------------------------------------------------------------------------

protected: // Note: protected if used by subclasses, otherwise private

/\*\* ... \*/

static const char k\_szModuleName;

private:

/\*\* ... \*/

EDtmfVehicleType m\_eMasterState;

//-----------------------------------------------------------------------------

// Class methods

//-----------------------------------------------------------------------------

public: // Note: constructor and destruct first in the public section

/\*\* ... \*/

CDtmfService(const int32\_t iPrio, const char\* szParamSetName);

/\*\* ... \*/

virtual ~CDtmfService();

private:

/\*\* ... \*/

int32\_t handleSendDtmfMsg(SIGNAL \*msg);

//-----------------------------------------------------------------------------

// Implementation of CProcess

//-----------------------------------------------------------------------------

protected:

/\*\* ... \*/

int32\_t run(void);

/\*\* ... \*/

int32\_t handleMsg(union SIGNAL \*msg);

//-----------------------------------------------------------------------------

// Implementation of CParameterParserCallback

//-----------------------------------------------------------------------------

protected:

/\*\* ... \*/

void newParameter(int32\_t iNumber, bool bFound, const char\* szValue);

//-----------------------------------------------------------------------------

// Inner classes

//-----------------------------------------------------------------------------

private:

/\*\* ... \*/

class CResInfo

{

//------------------------------------------------------------------------

// CResInfo: Class data members

//------------------------------------------------------------------------

private:

/\*\* ... \*/

bool m\_bAutoReset;

//------------------------------------------------------------------------

// CResInfo: Class methods

//------------------------------------------------------------------------

public:

/\*\* ... \*/

bool isAutoReset() const { return m\_bAutoReset; }

};

};

Rule 207: Use ‘//’ for comments within methods

Use ‘//’ for all comments in source files within methods, not ‘/\* \*/’.

An exception is commenting optional arguments within the method signature.

Example:

CResHndInt::CResHndInt(CMsgLoopInt& xmliMsgLoop, const bool bResMgmt /\* =false \*/)

Bad example:

void badExample()

{

/\*

This is a bad example because in search results the following line

will not be recognizable as comment. It will look like executed code.

callMyFuntion(arg1, arg2, arg3);

\*/

doNotCallMyFuntion();

}

Recommendation 208: Write helpful comments

Comments within methods should explain *why* something is done. Omit obvious comments and describe things about the context which help understanding the code.

Also consider commenting if-else conditions, because often they are unclear to others which try to understand the code.

Bad examples:

// subscribe for the signals **<- obvious comment**

subscribeMsg(this, (SIGSELECT\*)m\_rgSigArr);

//updateLineInformations **<- what’s that?**

updateLineInformation(eLiInValCirculation,lCirculationId);

struct LEH\_Params

{

/\*\* The path to the data files\*/ **<- what kind of data files? optional?**

char\* m\_szDataFilePath;

};

/\*\* Pointer to the EventHandlerAssigner \*/ <- **why** do we need this thing?

CEventHandlerAssigner\* m\_pxEventHandlerAssigner;

Good examples:

/\*\* Timeout for the Request in ms. If during this timeout a reply has not arrived, the request is sent again \*/

uint32\_t m\_iRequestTimeout;

if(iIndex == -1)

{

REPORTERROR\_EXT(MMI\_ERR\_PARAM\_SYNTAX, …

}

else

{

// the index is correct "de:12" **<- this is unclear without some comments**

if((uiPosStart < uiPosSep) && ((uiPosSep - uiPosStart) == 2))

{

// an ISO631 code is given -> test it if it's ok

…

}

Rule 209: Commented code

Never “deactivate” code by commenting it out without writing any comments. Describe why you did it and how you are going to fix it. Leave your name and the date of your changes (date in ISO 8601 format like YYYY-MM-DD).

Bad examples:

if (checkInitData(bUnknownCourse))

{

// m\_xLineTracerProxy.sendStateAnswerIO(m\_iLastSourceId); //old ibis **<- ???**

// other example

else

{

// get it from the cache

/\*eErrorCode = m\_xLineTracerProxy.getLongName(…);\*/

}

Good example:

…

else

{

**//!! 2013-11-14 ClNo:** Currently unclear, if the long names or the short

// names will be used. To be fixed in the CR “DTMF protocol extension”.

//// get it from the cache

// eErrorCode = m\_xLineTracerProxy.getLongName(…);

}

Rule 210: Mark up comments

Use the following tags to mark important locations in the code that need to be revisited:

* // TODO
* // FIXME
* //!!

Optionally the current date and user acronym may be added.

Example: //!! 2016-11-14 ClNo

# Appendix A: Hungarian Notation

## Type Prefixes

| Prefix | Data type | Example |
| --- | --- | --- |
| C | Class | class CPoint { ... } |
| S | Struct | struct SMessage { ... } |
| T | Typedef definition | typedef std::vector<SMessage>::iterator TMsgIt; |
| E | enum (as definition) | enum EErrState { ... } |

## Variable and Object Prefixes

| Prefix | Variable, object | Example |
| --- | --- | --- |
| b | bool | bool bFlag |
| c | (signed) char | char cSeparator |
| i | (signed) int32\_t, ssize\_t  OFRIS: dbkey\_t, resid\_t | int32\_t iIndex  for(int32\_t i = 0 ; i < sizeof(rgClients) ; ++i)  ssize\_t iBytesWritten  dbkey\_t iActionId  resid\_t iAlarmOutId |
| i8 | int8\_t | int8\_t i8Length (number from 0…255, not character) |
| i16 | int16\_t | int16\_t i16Mode |
| i64 | int64\_t | int64\_t i64Distance |
| ui | uint32\_t, size\_t | uint32\_t uiLength  size\_t uiPos |
| u8 | uint8\_t | uint8\_t u8Status = 1; |
| u16 | uint16\_t | uint16\_t u16MyCounter |
| u64 | uint64\_t | uint64\_t u64LargeValue |
| f | float | float fSpeed |
| d | double | double dLatitude |
| ld | long double | long double ldSineValue |
| e | enum (as variable) | void reportError(const EErrState eErrState); |
| e | exception | catch(CGeneralException& e) { ... } |
| sz | C string (NULL byte terminated)  character arrays | char\* szName  char szFilename[k\_uiSize] |
| rg | Ranges, arrays | int32\_t\* rgClients  static const CParameterParser::SParamToParse m\_rgParameter[]; |
| x | Classes and all other undefined data types | CErrState xErrState(this);  PROCESS xPid = get\_pid(); |
| p | Pointer to classes  (use “rg” for arrays) | CMsgLoopInt\* m\_pxMsgLoop; |
| sig | OFRIS signals | SIGNAL\* psigMsg; |
| it | iterators | vector<SMessage>::iterator itMsg  for(vector<int32\_t>::const\_iterator it = vecClients.begin(); it != vecClients.end(); ++it) |

Variables and objects of a type defined in a ‘typedef’ may use the prefix ‘x’ *or* the prefix of the base type. Example:

const OSADDRESS CEuroStarService::m\_xStackSize = 8192;

const OSADDRESS CEuroStarService::m\_uiStackSize = 8192;

## Container

| Prefix | Data type | Example |
| --- | --- | --- |
| str | str::string  CStr | string strName  CStr strIdentification |
| os | std:ostring, std::ostringstream | ostringstream osDebug;  ostringstream os; // for local string conversions |
| is | std:istring, std::istringstream | istringstream isReceivedString;  istringstream is; // for local string conversions |
| ss | stringstream  Note: prefer ostringstream | stringstream ssMyStream; |
| fs | input / output filestream | ofstream fsOut;  ifstream fsIn; |
| lst | std::list  CStrList | std::list<SDisplayContent> lstDispContent |
| vec | std::vector | std::vector<SDisplayContent> vecDispContent |
| map | std::map | std::map<int32\_t, ELanguage> mapLangPrio; |
| mmap | std::multimap |  |
| deq | std::deque | std::deque<CPdiDriverMsgInfo\*> m\_deqOpenCalls; |
| set | std::set |  |
| mset | std::multiset |  |

# Appendix B: List of Common Abbreviations

| Long description | Abbreviation | Example |
| --- | --- | --- |
| Information | inf, info |  |
| Message | msg |  |
| Heartbeat | hrtb (or heartbeat) |  |
| Identification/Identity/Identifier  (a unique identifier) | id |  |
| Index (zero based enumeration) | idx |  |
| Error | err |  |
| Interface | Int (not integer!) |  |
| Change Request | CR |  |

# Appendix C: List of Rules Applying to .sig Files

Files containing signal definitions (extension .sig) are exempted from many rules specified in this document. The following list of rules apply:

* Rule 06: Naming files
* Rule 12: Include guards
* Rule 14: Relative and absolute include paths
* Rule 15: Include everything needed
* Rule 17: #include “ownfile.h”
* Rule 18: #include <systemfile.h>
* Rule 19: Unnecessary include files
* Rule 27: No circular dependencies
* Rule 37: No numeric values - “magic numbers”
* Rule 76: Separate elements by two empty lines
* Rule 77: Terminate files by a newline
* Recommendation 78: No space with unary operators
* Rule 79: Connect \* and & to the type, not to the name
* Rule 80: Indentation and use of tabs/spaces
* Rule 82: Variables and functions: start with lowercase letter
* Rule 84: Use camel-case (does not apply to the struct name!)
* Rule 86: No underscores in name (does not apply to the struct name!)
* Rule 87: Avoid similar names
* Rule 89: Names of constants, enumerations and macros
* Recommendation 90: Use self-descriptive names
* Recommendation 91: Names should be pronounceable
* Recommendation 92: Avoid not generally accepted abbreviations
* Rule 93: Use of the Hungarian Notation
* Rule 101: Use the existing namespaces
* Rule 103: One declaration per line
* Rule 104: Allowed integer types
* Rule 200: Include an introductory comment
* Rule 202: Language: English
* Rule 203: Write comments for classes, functions, etc.
* Rule 205: Use the Doxygen format
* Recommendation 208: Write helpful comments
* Rule 209: Commented code
* Rule 210: Mark up comments

# Referenced Documents

| Doc. Reference | Description of the Document |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Revision History

This document may only be changed if agreed with the process/document owner.

Changes related to the content of the document require new release by the review group.

| Revision | Date  (yyyy-mm-dd) | Initials | Description of Changes |
| --- | --- | --- | --- |
| 00 (Draft)  00 (Draft) | 2014-01-17  2014-10-29  2017-10-16 | ClNo ChKu  ClNo | Major overhaul of the Coding Guidelines  Corrections after the first automatic tests  Many clarifications and example improvements, introduction of new namespaces |
| 00 | 2017-10-25 | ClNo | New Word template and document number; finalization |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Review Group

This document was reviewed as follows:

| Name | Location / Organization | Date | Signature |
| --- | --- | --- | --- |
|  |  |  | signed |
|  |  |  | signed |
|  |  |  | signed |
|  |  |  | signed |
|  |  |  | signed |