Coding Rules for *OpENer* — Open Source EtherNet/IP^{TM} Adapter Stack

Version 2.0

Martin Melik Merkumians*

2015-11-15

Contents

1 Introduction **1**

2 Comments **2**
2.1 Fileheaders 2
2.2 Revision History 2
2.3 Keywords 2

3 Datatypes **2**

4 Naming of Identifiers **3**
4.1 Pre- & Postfixes 3
4.2 Variables 4
4.3 Constants 4
4.4 Functions 4
4.5 Structs 4
4.6 Enums 5

5 Code Formatting **5**

6 Assertions **5**

1 Introduction

This document describes the coding rules, which has to be used in the OpENer project. These rules are mainly the Google C++ style rules, with some extensions specific to C and the OpENer project. If something is not covered by the rules given in this document, please check the official Google C++ style guide, available at `http://google.github.io/styleguide/cppguide.html`. Additional code style examples can be found at `https://gist.github.com/davidzchen/9187878`.

As the OpENer code style aims to be as close as possible to the established Google C++ code style, please file an issue if anything in this guide contradicts the Google C++ code style.

*melik-merkumians@acin.tuwien.ac.at

2 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 function, structure, ... definitions have to follow the conventions of *Doxygen* to allow the automated generation of documentation for the source code. Hereby Java-style Doxygen comments shall be used. Doxygen comments shall therefore start with slash and two starts, and use the @ symbol to indicate Doxygen keywords. For enums, variables, and strucutres inline documentation with /**< shall be used. Autobrief behavior shall not be assumed for Doxygen comments. See the example below.

```
/** @brief function, structure, enum, etc. to comment
 *
 *  Detailed explanation, spanning multiple lines if needed.
 *  @param parameter1 Parameter1 description
 *  @return Return value description
 */
int foo(char bar) {
    ...
}

const int g_kFooBar = 1; /**< Global constant which needs documentation */
```

Comments have to be meaningful, to describe to program and to be up to date.

2.1 Fileheaders

Every source-file must contain a fileheader as follows:

```
/******************************************************************************
 * Copyright (c) 2009, Rockwell Automation, Inc.
 * All rights reserved.
 *
 * Contributors:
 *     <date>: <author>, <author email> - changes
 *****************************************************************************/
```

Each author needs to explain his changes in the code.

2.2 Revision History

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

2.3 Keywords

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

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

3 Datatypes

Table 1 on the following page contains the definitions of important standard datatypes. This is done to ensure a machine independant defintion of the bit-width of the standard data types. For *OpENer*-development these definitions are in the file: `src/typedefs.h`

These data types shall only be used when the bit size is important for the correct operation of the code, whereby Eip-prefixed data types shall be used for communication functions, and Cip-prefixed data types shall be used for CIP related functions and objects. If not we advice to

Table 1: Data types used in OpENer

defined data type	bit-width / description	used C-datatype
EipByte	8 bit unsigned	uint8_t
EipInt8	8 bit signed	int8_t
EipInt16	16 bit signed	int16_t
EipInt32	32 bit signed	int32_t
EipInt64	64 bit signed	int64_t
EipUint8	8 bit unsigned	uint8_t
EipUint16	16 bit unsigned	uint16_t
EipUint32	32 bit unsigned	uint32_t
EipUint64	64 bit unsigned	uint64_t
EipFloat	single precission IEEE float (32 bit)	float
EipDfloat	double precission IEEE float (64 bit)	double
EipBool8	byte variable as boolean value	unit8_t
CipOctet	unspecified type	uint8_t
CipBool	byte variable as boolean value	uint8_t
CipByte	8 bit unsigned	uint8_t
CipWord	16 bit unsigned	uint16_t
CipDword	32 bit unsigned	uint32_t
CipUsint	8 bit unsigned	uint8_t
CipUint	16 bit unsigned	uint16_t
CipUdint	32 bit unsigned	uint32_t
CipSint	8 bit signed	int8_t
CipInt	16 bit signed	int16_t
CipDint	32 bit signed	int32_t
CipReal	single precission IEEE float (32 bit)	float
CipLreal	double precission IEEE float (64 bit)	double
CipLint	64 bit signed	int64_t
CipUlint	64 bit unsigned	uint64_t
CipLword	64 bit unsigned	uint64_t

use the type `int` or `unsigned int` for most variables, as this is the most efficient data type and can lead on some platforms (e.g., ARM) even to smaller code size.

4 Naming of Identifiers

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

4.1 Pre- & Postfixes

The following prefixes have to be applied to identifiers:

- *"g_"* shall be prefixed for global variables.
- *"_"* shall be postfixed for member variables. These are usually CIP object variables with file-global scope.

4.2 Variables

Variables have to be named self explanatory. The names have to be provided with the appropriate pre- or postfix and shall be all lowercase letters, and if a name consists of more than one word underscores shall be used for separating these words. The only exception are loop variables (thereby the use of i, j, k is allowed). Only one variable declaration per line is allowed. Pointer operators at the declaration have to be located in front of the variable (not after the type identifier). If possible initializations have to be done directly at the declaration.

Examples

```
int i;
int local_variable;
CipBool boolean_flag_in_cip_object_;
```

4.3 Constants

The preferred way to declare constants is to define them as*const* data types, if this is not possible constants shall be defined as pre-processor statements, via *#define*. If constants are defined as C constants the name of the constant shall start with *k*, followed by the constant name in Pascal case. If a constant is defined as a pre-processor statement the constant name shall be all upper case, separating multiple words with underscored. Avoid the using "magic numbers" (e.g. `if (x == 3){...}`). Instead use constants.

Examples

```
static int g_global_variable;
static const int g_kAGlobalConstant = 73:
const int kAnImportantConstant = 42;
#define DO_NOT_DO_THIS_IF_IT_IS_NOT_NECESSARY bad
```

4.4 Functions

Functions names shall be Pascal cased, function parameters shall be named like variables. The parameter list shall adhere to the following rules:

- Input parameters shall come first
- Input parameters shall be const
- Output parameters shall be last

Examples

```
int FooBar(const int foo, const char* const bar, double* additional_return_value)
```

4.5 Structs

The default case for structs shall be, that they are defined as anonymous structs, giving them a type name via the *typedef* keyword. Struct names shall be pascal cased. If a struct type is needed, before a typedef alias can be created (usually inside the same struct), the struct name shall be repeated in the struct type name, following the conventions for variable names, with all lowercase and words separated with underscores.

The element names inside the struct are following the normal conventions for their types.

Examples

```
typedef struct {
    int foo;
    char bar;
} TheDefaultCase;

typedef the_excpetion {
    struct the_exection *needed_the_struct_definition_already_here;
    char other_elements;
} TheException;
```

4.6 Enums

Enums shall be defined anonymous and typedef'ed to a type name. As the values inside an enum are constant, the naming scheme of constants apply for enum members. As Enums do not define their own namespace, the enum type name shall be added between the initial *k* and the constant name.

Examples

```
typedef enum {
    kImportantEnumConstant1 = 0,
    kImportantEnumConstant2 = 1
} ImportantEnum;
```

5 Code Formatting

In order to have consistent code formating the Google C++ coding style rules shall apply. When using Eclipse as development environment the coding format xml file is available at `https://github.com/google/styleguide`. By pressing `<ctrl><shift>f` the formatter will format the code according to these rules.

6 Assertions

The OPENER_ASSERT(e) macro is available for traditional assertion checks, halting the program if the expression provided as the argument e evaluates false. This macro shall *only* be used to test conditions where the only possible cause of failure is an unquestionable bug with this program, typically leading to undefined behavior or a crash if execution were permitted to continue. In other words, an assertion shall *never* fail as a result of any external input, valid or invalid, or other similar foreseeable condition, such as a memory allocation failure. These latter type of failures must be handled by normal code execution paths that yield responses with appropriate error codes or possibly terminating the program with a non-zero exit code, not an assertion failure.

The following listing of a function to set an attribute's value based on received data is an example to help illustrate proper use of assertions. The raw and len parameters refer to the received data and length, and the foo parameter points to the target attribute; the function returns true only if the attribute was set successfully.

```
bool SetAttributeFoo(const void *raw, size_t len, CipDint *foo) {

  /*
   * This function should never be called with NULL pointers, regardless of
   * what was received over the network, so assertions should be used to
   * validate the pointer arguments.
   */
  OPENER_ASSERT(NULL != raw);
  OPENER_ASSERT(NULL != foo);

  /*
```

```
    * Ensuring enough data was received to satisfy the target data type
    * must not be done with an assertion as a malformed message containing
    * insufficient data shall not halt the program.
    */
  if (sizeof(CipDint) > len) {
    return false;
  }

  CipDint new_value = &(int *)raw;

  /*
   * Here the received value is tested for conformance to acceptable values;
   * assume for the sake of this example that allowable values are nonzero.
   * Validating values received from external sources must not be done
   * with assertions.
   */
  if (0 == new_value) {
    return false;
  }

  *foo = new_value;
  return true;
}
```