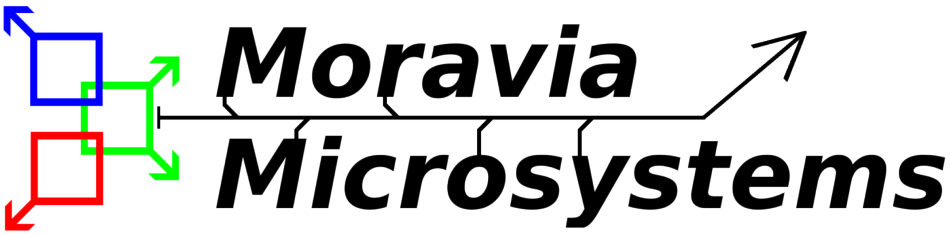

Figure 1: *Moravia Microsystems, s.r.o.*

Coding Convention

Corporate coding convention guildelines for C++ and C.

Martin Ošmera
Brno, 2013

Contents

Chapter 1

Importance of a code convention

First of all it is important to keep in mind that this convention is not supposed to create a set of unnecessary and unproductive rules to poison your work; instead in case you are sure that your work goes better with another coding style, go ahead but please at least read this paragraph till end. Coding convention is something you probably get used to, sooner or later, and it contributes in creating unified working environment where you always know what to expect. Also it helps to write code which is relatively easily readable for someone who is familiar with the applied convention.

Code conventions are important to programmers for a number of reasons:

- About 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

Chapter 2

Detailed specification

2.1 General

2.1.1 File names

All header files should contain exactly one class, or struct, or namespace in the root namespace; name of this data type or namespace has to be the same as name of the file itself; this convention has already been already adopted by the Java programming language. Other classes, structs, and namespaces should be declared/defined only withing this one "main" class, struct, or namespace. Unless it is necessary, the header file should not contain any declarations outside the class, struct, or namespace.

Use the following file suffixes:

File Type	Suffix
C/C++ header file	.h
C++ source file	.cxx
C++ source file	.c

2.1.2 Maximum line length

- Normally, do not write lines longer than **120 characters**.
- Try to use the horizontal space (120c), do not break lines unnecessarily.
- In certain specific cases when you consider it to be prudent, make lines even longer but generally try to avoid it.

2.1.3 Indentation

- Indent with spaces, use **4 spaces** for a level.
- Do not use tabs.

2.2 Header files

2.2.1 Sections

- //// Public Constants ////
- //// Public Datatypes ////
- //// Protected Datatypes ////
- //// Private Datatypes ////
- //// Constructors and Destructors ////
- //// Public Operations ////
- //// Inline Public Operations ////
- //// Static Public Operations ////
- //// Inline Private Operations ////
- //// Static Inline Private Operations ////
- //// Protected Operations ////
- //// Public Attributes ////
- //// Protected Attributes ////
- //// Private Attributes ////
- //// Static Public Attributes ////
- //// Static Private Attributes ////

2.2.2 In-file documentation

Each header file should start with the following lines, of course exact content of the text has to be correspondent to what the file is for:

```
// ============================================================================
/**
 * @brief
 * C++ Interface: Base class for ANS.1 BER encoder.
 *
 * This class implements ...
 * ... ... ...
 * ... ... ...
 *
 * (C) copyright 2013 Moravia Microsystems, s.r.o.
 *
 * @author Your Name <your.name@email.xx>
 * @ingroup SomeGroup
 * @file FileName.h
 */
// ============================================================================
```

2.2.3 Example header file

Name of header file is "PicoBlazeStack**.h**", name of implemetation file is "PicoBlazeStack**.cxx**".

```
// ================================================================================================================================
/**
 * @brief
 * C++ Interface: PicoBlaze simulator subsystem for stack.
 *
 * Implements processor stack, requires no configuration, default stack size is 31, the stack in initialized to contain
 * only zeros after MCU reset. Fast operation of the subsystem was a priority. The stack subsystem simulator shares the
 * common memory interface, like register file, scratch pad RAM, program memory, etc.
 *
 * (C) copyright 2013 Moravia Microsystems, s.r.o.
 *
 * @author Martin Ošmera <martin.osmera@gmail.com>, (C) 2013
 * @ingroup PicoBlaze
 * @file PicoBlazeStack.h
 */
// ================================================================================================================================

#ifndef PICOBLAZESTACK_H
#define PICOBLAZESTACK_H

// Forward declarations
class DataFile;

#include "../MCUSim.h"

/**
 * @brief
 * @ingroup PicoBlaze
 * @class PicoBlazeStack
 */
class PicoBlazeStack : public MCUSim::Memory
{
    ////    Public Datatypes    ////
    public:
        /**
         * @brief Events generated by the subsystem.
         */
        enum Event
        {
            EVENT_STACK_OVERFLOW,  ///< Stack capacity was already exhausted by the previous push.
            EVENT_STACK_UNDERFLOW  ///< A value was been popped from the stack while the stack was already empty.
        };

        /**
         * @brief Subsystem configuration.
         */
        struct Config
```

```
        {
            /// Sets configuration to default setting, i.e. for KCPSM3.
            Config()
            {
                m_size = 31;
            }

            /// Stack capacity.
            unsigned int m_size;
        };

    ////    Constructors and Destructors    ////
    public:
        /**
         * @brief Initializes memory used by the stack.
         */
        PicoBlazeStack();

        /**
         * @brief Cleans up memory dynamically allocated for the stack.
         */
        ~PicoBlazeStack();

    ////    Public Operations    ////
    public:
        /**
         * @brief Link the subsystem with other simulator subsystems which this subsystem necessarily needs to function.
         * @param[in,out] eventLogger Simulator event observer.
         * @return This object.
         */
        PicoBlazeStack * link ( MCUSim::EventLogger * eventLogger );

        /**
         * @brief Reset the subsystem in the specified mode.
         *
         * It is supposed to be called by mainly when the entire processor simulator is reseted.
         *
         * @param[in] mode There are various modes of reset, e.g. accept new configuration, processor master reset, etc.
         */
        void reset ( MCUSim::ResetMode mode );

        /**
         * @brief Read data directly from the stack bypassing its normal mode of operation.
         * @param[in] addr Source address, starts with 0; push increments, pop decrements.
         * @param[out] data Address to the program memory read from the specified address.
         * @return This method doesn't generate simulator events that's why it uses return code to report final status.
         */
        MCUSim::RetCode directRead ( unsigned int addr,
                                     unsigned int & data ) const;
```

```
    /**
     * @brief Write data directly to the stack bypassing its normal mode of operation.
     * @param[in] addr Target address, starts with 0; push increments, pop decrements.
     * @param[in] data Address to the program memory to store at the specified address.
     * @return This method doesn't generate simulator events that's why it uses return code to report final status.
     */
    MCUSim::RetCode directWrite ( unsigned int addr,
                                  unsigned int data );

    /**
     * @brief Change stack capacity.
     * @warning This operation does NOT preserve the memory contents, i.e. the stack is (re-)initialize to zeros.
     * @param[in] newSize New size of the stack in number of values which can be stored in the stack at once.
     */
    void resize ( unsigned int newSize );

    /**
     * @brief Load contents of the stack from a file, like Intel 8 HEX file or something of that sort.
     * @param[in] file File data container.
     */
    void loadDataFile ( const DataFile * file );

    /**
     * @brief Store contents of the stack to a file, like Intel 8 HEX file or something of that sort.
     * @param[in,out] file File data container.
     */
    void storeInDataFile ( DataFile * file ) const;

////    Inline Public Operations    ////
public:
    /**
     * @brief Get stack capacity.
     * @return Stack in number of values which can be stored in the stack at the same time, not bytes.
     */
    unsigned int size() const
    {
        return m_config.m_size;
    }

    /**
     * @brief Push value onto stack.
     *
     * This method is NOT supposed to be used outside the simulator's own subsystems, like from GUI.
     *
     * @param value Address to the program memory to be stored in stack.
     */
    inline void pushOnStack ( unsigned int value );

    /**
     * @brief Pop value from the stack.
```

```
         *
         * This method is NOT supposed to be used outside the simulator's own subsystems, like from GUI.
         *
         * @return Address to the program memory retrieved from the stack.
         */
        inline unsigned int popFromStack();

    ////    Inline Private Operations    ////
    private:
        /**
         * @brief Reset the subsystem to a state in which real hardware would be after power-up.
         */
        inline void resetToInitialValues();

        /**
         * @brief Instruct the subsystem to accept new configuration.
         */
        inline void loadConfig();

        /**
         * @brief Reset the subsystem to a state in which real hardware would be after master reset.
         */
        inline void mcuReset();

    ////    Public Attributes    ////
    public:
        /// Subsystem configuration.
        Config m_config;

    ////    Private Attributes    ////
    private:
        /// Array holding the values pushed onto the stack.
        unsigned int * m_data;

        /// Stack pointer, starts with 0; push increments, pop decrements.
        unsigned int m_position;
};

// ------------------------------------------------------------------------------
// Inline Function Definitions
// ------------------------------------------------------------------------------

inline void PicoBlazeStack::pushOnStack ( unsigned int value )
{
    if ( m_config.m_size == m_position )
    {
        logEvent(EVENT_STACK_OVERFLOW, m_position, value);
        m_position = 0;
    }
    m_data[m_position++] = value;
```

```
}

inline unsigned int PicoBlazeStack::popFromStack()
{
    if ( 0 == m_position )
    {
        logEvent(EVENT_STACK_UNDERFLOW, m_position, -1);
        m_position = m_config.m_size;
    }
    return ( 0x3ff & m_data[--m_position] );
}

#endif // PICOBLAZESTACK_H
```

2.3 C/CXX files

It should look like this:

```
MCUSim::RetCode PicoBlazeStack::directRead ( unsigned int addr,
                                             unsigned int & data ) const
{
    if ( addr >= m_config.m_size )
    {
        return MCUSim::RC_ADDR_OUT_OF_RANGE;
    }

    data = ( 0x3ff & m_data[addr] );
    return MCUSim::RC_OK;
}
```

Some basic rules/recommendations:

- Use "CamelCase" for symbol names comprising of more that one word, i.e. practice of writing compound words where the first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. The only exceptions are constants, macros, and cases where usage of a different convention is inevitable. Example: use this: "`matrixEquationComputationModule`" instead of this "`matrix_equation_computation_module`".
- Local variables and class attributes should be distinguished from each other by their name in order to prevent unintentional use of the wrong variable. Use prefix "`m_`" for class, or struct, attributes.
- Names of data types and namespaces should begin with capital letter in all cases possible. Example: "`class BooleanLattice { ... };`".
- Names of all constants and macros should comprise of capital letter only, and should not use CamelCase.

  Examples:

  - "`static const int MAX_CAPACITY = 100;`",
  - "`#define GET_SECOND_BYTE(arg) ( ( arg & 0xff00 ) >> 8 )`".

- When you use "==" operator (equal to) and you compare a variable with a constant, then **put the constant on the left side** of the comparison; it prevents very unpleasant bugs when you instead of comparing two values you assign the variable a new value, in that case compiler won't complain about anything, it won't even display a warning message but the code is likely to fail to function properly, and this bug is usually very hard to find. Example: "`if ( -1 == abc ) { ... }`".

- When you use a condition such that you only check whether something is zero or not (e.g. "`if ( abc ) {...}`"), be more explicit about what "kind of zero" you expect: it might be zero as boolean value i.e. false, or NULL pointer, or integer zero, etc. It makes the code more readable when it's clear with which kind of value, possible function/method return value, are you dealing with.

  Examples:

  - use this: "`if ( NULL == abc() /* returns pointer */ ) { ... }`" instead of this: "`if ( !abc() ) { ... }`",
  - use this: "`if ( false == abc /* declared as bool */ ) { ... }`" instead of this: "`if ( !abc ) { ... }`",
  - use this: "`while ( 0 != abc /* declared as integer */ ) { ... }`" instead of this: "`if ( abc ) { ... }`".

- When you use nested expressions or you use operators with different precedence in an expression, make it absolutely clear how the expression is supposed to be evaluated. For example suppose we have an expression like this: "`return a / ( b - c ) >> d;`", white it instead like this: "`return ( ( a / ( b  - c ) ) >> d );`".

- **Never use the comma operator** (",") unless it is absolutely necessary, or it is a declaration/initialization of variables. For example never use something like this: "`if ( a > b, c ) { ... }`" but you might use something like this: "`for ( int i = 0, j = 1; i < 10; i++, j++ ) { ... }`".

- Do not try to save space at any cost, there is a plenty of space for source code on your computer, use spaces and empty lines to make the code more readable but don't use too much of white space, like sequences of several empty lines or spaces.

  Examples:

  - use this: "`if ( NULL == abc() ) { ... }`" instead of this: "`if(NULL==abc()) {...}`",
  - use this: "`for ( int i = 0; i < 10 ; i++ ) { ... }`" instead of this: "`for(int i=0; i<10; i++) { ... }`".