**FREEDM Distributed Grid Intelligence (DGI) Subthrust**

**Coding Standards  
Document**

Bruce McMillin

Draft 9

April 2012

# **Table of Contents**

Table of Contents 2

Introduction 3

Variables 3

Constants and Definitions 3

Files 4

Comment Block 4

Rules 4

Structures 5

Classes 5

Comment Block 5

Rules 5

Functions 7

Comment Block 7

Rules 8

Switch Statements 9

Loops 9

Comment Block 9

Rules 9

Example of a Fully Developed Loop Comment 10

Miscellaneous 10

Indentation 11

Braces 11

Correctness Considerations 11

Overall Coding Practices 11

Compiler Conventions 12

# **Introduction**

This document describes the requirements that must be met by all code written for the FREEDM DGI. The use of these coding standards is to promote reusable, reliable, readable, extensible, easily maintainable, cross-platform code that is well commented and documented. The commenting standards presented here are intended to be compatible with the Doxygen documentation tool. This document is a modified version of the coding standards document used for the Caffeine and AI Puzzle framework projects.

In this file, two types of standards will be used. Mandatory standards will be signified by the word “must”, whereas coding suggestions will be given by the word “should”.

# **Variables**

1. All variable names must be in title case, with the exception of the first letter of the name, which should be in lower case. Typically, loop counters do not have to follow this rule.  
   Ex: minVelocity, maxRecordedVelocity, speed, etc.
2. Variable names must be logical and descriptive.
3. Global variables must be avoided. If a value is needed globally, it should be encapsulated in a singleton or contained within a namespace.
4. There should be no static variables in class member functions.
5. Variable declarations should have some commenting to describe their functionality unless it is very obvious.
6. All variable declarations must be made at the beginning of their scope and initializations all done at this point (C style) except for temporary variable usage in C++.
7. Variables should be declared in alphabetical order for easiest reading.

# **Constants and Definitions**

1. All constants must be uppercase.
2. Limit the use of #define except for header file #ifndef and #define NDEBUG statements. Constants should be defined as C++ const variariables or enums rather than with the preprocessor.
3. “Magic numbers” should not be used.

# **Files**

## ***Comment Block***

Every file must contain the following Doxygen header comment block:

///////////////////////////////////////////////////////////////////////////////

/// @file Header.hpp

///

/// @author List of names and emails of significant contributors

///

/// @project FREEDM DGI

///

/// @description A brief description of the file

///

/// @functions List of functions and external entry points

///

/// These source code files were created at the Missouri University of Science

/// and Technology, and are intended for use in teaching or research. They may

/// be freely copied, modified and redistributed as long as modified versions

/// are clearly marked as such and this notice is not removed.

///

/// Neither the authors nor Missouri S&T make any warranty, express or implied,

/// nor assume any legal responsibility for the accuracy, completeness or

/// usefulness of these files or any information distributed with these files.

///

/// Suggested modifications or questions about these files can be directed to

/// Dr. Bruce McMillin, Department of Computer Science, Missouri University of

/// Science and Technology, Rolla, MO 65409 <ff@mst.edu>.

///////////////////////////////////////////////////////////////////////////////

## ***Rules***

1. Files should be no more than 1000 lines long.
2. Each line of code must not exceed 80 characters to prevent line wrapping.
3. Indents must be four spaces. Tabs must not be used.
4. File names should be in title case. E.g. CBoardState.cpp

# **Structures**

1. Structures should follow the same comment block convention as classes, presented below.
2. Structure names must follow the class naming convention, except start with “S” as opposed to “C”.
3. Structures must not contain functions. If a function is needed, the structure should be made into a class.

# **Classes**

## ***Comment Block***

Classes should have a Doxygen comment with a brief description immediately before the description. This description should be less than eighty characters and be a “one-liner” of what the class does.

/// This is my class, there are many like it.

class MyClass

Classes must always begin with the following comment block, which goes immediately after the opening French brace:

//////////////////////////////////////////////////////////

/// @description A brief description of the class

///

/// @limitations Any limitations for the class, or None

/////////////////////////////////////////////////////////

## ***Rules***

1. Class names must always begin with a ‘C’. Ex: CBook
2. Interface class names must begin with an ‘I’. Ex: ISocket
3. Class names must be in title case.
4. Class member variable names must be prefixed with “m\_”. Ex: m\_minValue
5. Accessor functions names must always be preceded with either Get or Set.
6. A “Get” accessor must not take any arguments.
7. A “Set” accessor should take a single argument and return a void type. The only exception to this is if you want to return a bool to verify the data is valid.
8. All public accessors should be const.
9. All classes that use heap memory must have the copy constructor and assignment operator defined (unless marked private).
10. All dynamic memory allocated in a class must be deleted by the same class.
11. Public class functions should never return a pointer or reference to dynamically created memory.
12. Prototypes for class methods and member declarations must be preceeded with a Doxygen comment describing its purpose in less than 80 characters. These should be used in addition to the long function comment block that appears before the implementation. Example:

/// Returns a count of the sheep

int getMySheep();

/// How many sheep this class has.

int m\_sheep;

Full example:

///A shepard class

class CLittleBoPeep

{

//////////////////////////////////////////////////

/// @description The legendary little shepard

/// @limitations If she loses the sheep she won't know /// where to find them.

//////////////////////////////////////////////////

public:

/// Returns a count of the sheep

int getMySheep();

private:

/// How many sheep this class has.

int m\_sheep;

}

# **Functions**

## ***Comment Block***

Function prototypes must be proceeded by a brief Doxygen comment explaining its purpose in a single line less than 80 characters.

/// Returns a count of the sheep

int getMySheep();

Function implementation must be preceded by the following comment block:

////////////////////////////////////////////////////////////

/// GetArea

///

/// @description A brief description of the function

/// @I/O Enumerate specific device communication (A/D,D/A)

/// @Peers For concurrent programs, processes this

/// function communicates with (can also be

/// replicated copies of itself) and threads and

/// interprocess/interthread communication

/// mechanism

/// @ErrorHandling enumerate exceptions and recovery

/// actions

/// @RealTime For Real-Time programs, indicate

/// periodicity, lower bound, upper bound

/// @Citations: For algorithms and mathematical

/// computations, cite the resource.

///

/// @pre Function's precondition in terms of program

/// variables and process statuses.

/// @post Function's postcondition in terms of program

/// variables and process statuses.

///

/// @param paramname Description of parameter

/// …

/// @param paramname2 Description of param

///

/// @return

///

/// @limitations

///

/////////////////////////////////////////////////////////

1. If no parameters are present, omit param lines entirely.
2. If there is no I/O to a physical device (non-network), omit I/O.
3. If the class uses network communication or authors messages, list the common recipient classes under peers.
4. If a function can throw an exception, list those exceptions under ErrorHandling and the circumstances when they can occur; otherwise, omit ErrorHandling.
5. If a function does not have real time requirements, omit RealTime.
6. If the algorithm or code is presented in a paper, cite it using Citation; otherwise, it may be omitted.
7. If the function returns void, or has no return (such as for a constructor), omit the return line entirely.
8. A function should have only one return statement. Obviously in some situations this is cumbersome, but if it is not, try to follow this convention.
9. Do not omit limitations. If the function has no limitations, simply write “none”.
10. Preconditions and postconditions must be stated within the function's description as shown above.

## ***Rules***

1. Function names should always follow variable naming conventions, with the exception that the first letter of the function name should be title case.  
   Ex: GetArea(...)
2. Functions should be no more than 400 lines of code and comment.
3. Functions should be const whenever possible.
4. Function parameters should be made const whenever possible.
5. Function bodies should be well commented. All major design decisions should be noted in the implementing code. These comments should explain why the given decision was made and possibly what other design options were considered. Code based on formal text should be cited whenever possible in the appropriate block.
6. Function parameters should be passed by reference (or pointer), not by value. Built-in data types, including STL iterators and std::string, should be passed by value.

# **Switch Statements**

1. Any time a break is left out of a case block, the comment “//break intentionally omitted” must be included.
2. All switch statements should have a default case.

**Loops**

## ***Comment Block***

Complex loops (those that do not simply iterate over a set of items) should be preceded by the following comment block:

///////////////////////////////////////////////////////////

/// High level description of what the loop does.

///

/// Loop Precondition: Loop's precondition

/// Loop Postcondition: Loop's postcondition

///

/// Invariant: Loop's invariant

///

/// Proof: Outline of Initialization,

/// Maintenance, and Termination proof

///////////////////////////////////////////////////////////

## ***Rules***

1. for loops should not modify their control variables inside the statement block unless this greatly simplifies the loop and an explanation is provided.
2. Invariants should logically imply their postconditions.
3. Proofs are needed for complicated loops, typically involving array copy operations that are not obvious. See the following for an example.

# ***Example of Fully Developed Loop Comment***

////////////////////////////////////////////////////////////////////////////////

/// Build F matrix from mismatch values where the mismatch equations are those

/// determined by a trapezoidal linearization of the ODEs associated with the

/// states of the power system being modeled (structure preserving classical

/// model).

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

/// Loop Precondition:

/// F is a (2n+2M) by 1 matrix (column vector), where there are n generator

/// buses and M lines in the system.

/// Numgens is an alias for n (strictly equal)

/// Numlines is an alias for M (strictly equal)

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

/// Loop Postcondition:

/// For all j=0 through Numgens-1,

/// F[ 2\*j ][ 0 ] = mismatch value for the (j+1)th system delta

/// F[ (2\*j)+1 ][ 0 ] = mismatch value for the (j+1)th system omega

///

/// Meaning that F contains a mismatch value for each generators DELTA

/// and each generators OMEGA (thus the first 2\*Numgens values of F are

/// set to the proper mismatch values -- remainder can be 0's or trash).

/// This leaves 2\*M (or equivalently 2\*NumLines) positions

/// yet to be filled by later processing.

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

/// Loop Invariant:

/// For all j=0 through i-1,

/// F[ 2\*j ][ 0 ] = mismatch value for the (j+1)th system delta

/// F[ (2\*j)+1 ][ 0 ] = mismatch value for the (j+1)th system omega

///

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

///

/// Proof:

///

/// INITITIALIZATION (invariant):

/// i = 0, thus a null range on j

///

/// MAINTENANCE (invariant):

/// loop starts with the first (i-1) pairs of mismatch values in F.

/// Calculate and insert the next pair of values into F.

/// Increment i to account for the insertion, which keeps the

/// invariant valid.

///

/// TERMINATION (invariant):

/// i = Numgens, which causes j to take on Numgens distinct values.

/// Thus, all Numgens pairs of values have been traversed/set.

////////////////////////////////////////////////////////////////////////////////

# **Miscellaneous**

## ***Indentation***

1. Indentation is four spaces. Tab characters must not be used.
2. The code within a scope must always be indented once. Namespaces are an exception.
3. Code should not be nested any deeper than five levels of indentation – such code is a candidate for decomposition.
4. The else statement in an if-else block must be placed on its own line, as shown here:

if(condition)

{

// body

}

else

{

// body

}

## ***Braces***

1. Opening French braces should be on their own line.
2. Closing braces should have the same indentation as the associated opening brace.
3. Empty functions and namespace declarations are exempt from this rule.

## ***Correctness Considerations***

Use the assert() statements (or exceptions) for all significant preconditions, postconditions, and program and loop invariants when developing code. These may be turned off for released code only if performance is a significant issue: it’s better to have slightly slower code than incorrect code. For distributed code, additional standards may be developed for assertions on collected state.

## ***Overall Coding Practices***

1. Recursion should be avoided unless the original mathematical expression or algorithm is best expressed through recursion and speed is not an issue.
2. Set internal units should be specified and used, ideally throughout the entire system. For example, all voltage variables could be kept in millivolts. Conversion should occur only before user output, after user input, or before and after interfacing with a specific piece of hardware that does not allow using the standard internal unit. Ensure such unit conversions are well documented.
3. Using increments/decrements inside statements should not be done. For example, don’t setchannel(++a); The one exception to this is using postincrement within a for loop declaration.
4. “if” conditionals should be a Boolean statement. If you mean to see if something is zero or not, ask if it == or != zero.
5. French braces must be included around the body of an if statement or loop even if only one line is done.
6. Do not use ternary conditional expressions (x?y:z).
7. Limit the use of macros.
8. Check all return statuses!
9. In general, write modern C++. Use std::vector rather than built-in arrays, new and delete in preference to malloc and free, and std::string rather than char\*. Utilize Boost smart pointers to manage heap memory.

## ***Compiler Conventions***

1. Filename extensions:

.h: Header that can be used in C or C++ code

.hpp: Header that can only be used in C++ code

.c: C source file

.cpp: C++ source file

.tpp: C++ header implementation file for a templated class

This scheme allows for sharing header files between C and C++, which is nice when dealing with numerical code which is mostly written in C. It also avoids the ugliness of #include "foo.cpp" by separating out object code from templated class implementation.

Do not use any reserved identifier conventions. C++ reserves any identifier beginning with an underscore, and any identifier with two or more consecutive underscores anywhere in it, for the standard library implementation.

1. Flags

gcc/g++ flags should include -Wextra -Wall -pedantic. Most warnings are there for a good reason. Suppress warnings with gcc pragmas only if the warnings are fully understood, and reenable them as soon as possible. All warnings must be either corrected or suppressed.

gcc should be invoked with -std=c99 and g++ with -std=c++98. This is to prevent GNU-specific code from creeping in.