Team 5

C-erious Coding Conventions

<Version 2.0>

Revision History

Date	Version	Description	Author	Reviewers
20–10-2016	1.0	Initial Report	Sagrika Rastogi	Sai Rahul
22–10-2016	1.1	Corrected formatting errors	Sagrika Rastogi	Vaishnav Vishal

Table of Contents

1 INTRODUCTION

1.1 PURPOSE

The goal of these guidelines is to create uniform coding habits among software developers so that reading, checking, and maintaining code written by different persons becomes easier

When a project adheres to common standards, the following are facilitated:

- Programmers can go into any code and figure out what's going on, so maintainability, readability, and reusability are increased.
- Code walk-throughs become easy.
- New people can get up to speed quickly. People new to a language are spared the need to develop a personal style.
- People new to a language are spared making the same mistakes over and over again, so reliability is increased.
- People make fewer mistakes in consistent environments

A mixed coding style is harder to maintain. So it's important to apply a consistent coding style across a project. When maintaining code, it's better to conform to the style of the existing code.

1.2 SCOPE

This document describes general software coding standards that the team will follow to develop the desired application.

The guidelines described in this document can be applied to the back-end as well as the GUI alike.

If these standards - when used as directed - fail to perform as expected, they can be edited and adapted to changing environments. The spirit of this document, not it's rules, should dictate the place of standards and consistency within the project.

1.3 GLOSSARY

- The term "**program unit**" (or sometimes simply "**unit**") means a single function, procedure, subroutine or, in the case of various languages, an include file, a package, a task, etc.
- A "**function**" is a program unit whose primary purpose is to return a value.
- A "**procedure**" is a program unit which does not return a value (except via output parameters).
- A "**subroutine**" is any function or procedure.
- An "**identifier**" is the generic term referring to a name for any constant, variable, or program unit.
- A "**module**" is a collection of "units" that work on a common domain.

1.4 REFERENCES

- Recommended C Style and Coding Standards, Indian Hill C Style and Coding Standards Updated by authors from Bell Labs, (http://www.cs.huji.ac.il/papers/cstyle/cstyle.html)
- C style and Coding Standards for the SDM Project, ROUGH DRAFT, July 3, 1995 (http://www.c8.lanl.gov/sdm/DevelopmentEnv/SDM_C_Style_Guide.html)

2 FILE GUIDELINES

2.1 HEADER FILES

Code files should have the following ordering:

- Beginning comments
- Package and Import statements
- Method Headers and Declarations

2.2 FILE NAMING GUIDELINES

A consistent naming convention for files and for directories shall be developed and used on a perproject basis. A file naming convention makes project files easily

distinguishable from other projects, and it helps associate different file types within the same project. Directories and subtrees can be used to link portions of a project together

3 SUBROUTINES

Repetitive sections of code should be made subroutines so that parallel maintenance of several copies of the same code can be avoided, and so that maintainers can be sure of the similarity of the passages.

3.1 SUBROUTINE DECLARATION GUIDELINES

- The subroutine's return type should be declared (do not allow the compiler to select a default value).
- Each parameter type should be declared (do not allow the compiler to select a default value).
-

3.2 LAYOUT GUIDELINES

- Each subroutine must begin with a comment that lists the precondition, postcondition, functionality of the subroutine.
- The general ordering should be
    - Subroutine Header
    - Variable declarations
    - Comments and Code
- Try to have different names for the function parameters and arguments passed upon invocation.

4 COMMENTS

4.1 COMMENT BLOCK STANDARDS

Code is more readable when comments are presented in paragraph form prior to a block of code, rather than a line of comment for a line or two of code.

- Comments should be preceded by and followed by a single blank line.

- Indent block comments to the same level as the block being described.
- Highlight the block in a manner that clearly distinguishes it from the code
  For example:
  #============================
  # This is an example of a blocked comment that is easily
  #distinguished from the rest of this document,
  #and is easily edited.
  #============================

4.2 COMMENT BLOCK STANDARDS

Inline comments adding important information are encouraged.

For example:

int gStateFlag; # This state variable is defined here, initialized in Main

A comment shall accompany each definition of a constant, type, or variable, giving its purpose. The comment shall either be on the same line as the definition (to the right), or on the previous line(s). If in the same line, it should be 2 tab spaces away from the line of code.

4.3 COMMENTING CONTROL CONSTRUCTS

- For conditional statements 'if' and 'switch-case':
  - State via an inline comment the purpose of the condition
  - Explain each Boolean condition

- For looping statements 'if' and 'switch-case':
  - State via an inline comment the purpose of the loop
  - State where the initialization is made, condition is validated, update is made.

5 CODE LAYOUT

5.1 ONE STATEMENT PER LINE

There shall normally be no more than one statement per line. This includes control statements such as “if”, “while” and “end” statements.

5.2 INDENTATION

A consistent use of indentation makes code more readable and errors easier to detect.

Use Tabs instead of spaces. 1 tab = 4 spaces is recommended per indent. Statements that affect a block of code (i.e. more than one line of code) must be separated from the block in a way that clearly indicates the code it affects.

Use vertical alignment of operators and/or variables whenever this makes the meaning of the code clearer

5.3 BRACKETS

Use in-line opening bracket and closing bracket in the line below the last line of code in the same vertical column as the beginning of the block.

For example:

```
if (TestFlag = = TRUE) {
        Run_Machine( )
 }
```