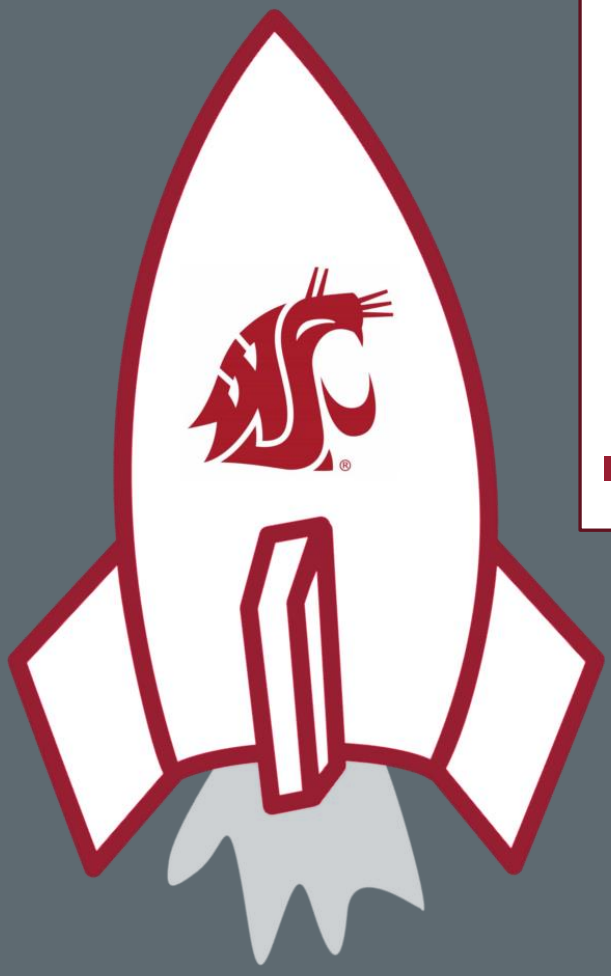
This document explains the various standards followed by the software in Cougs in Space

Code

Conventions and Guidelines

Revision: 1.0.1



Bradley Davis

Table of Contents

1 Introduction

This document describes the Cougs in Space recommended style for writing programs, where code with "good style" is defined as that which is

- Organized
- Easy to read
- Easy to understand
- Maintainable
- Efficient

The languages relevant to this document are C/C++ and Java. These are the primary languages utilized in Cougs in Space. Other languages may be used but are discouraged.

2 Readability and Maintainability

This section summarizes principles that maximize readability and maintainability of code. This is important as it allows collaborators to easily and quickly improve the software.

Eclipse has an automatic formatter (ctrl + shift + f). There are Cougs in Space formatter profiles for both C/C++ and Java whose options reflect this document. The formatter works most of the time; however, follow this document for the correct formatting.

2.1 Line Length

A line of code is limited to 80 characters long.

Exceptions

- A comment may exceed the rule if it is not feasible to split the line without harming readability. e.g. if a line contains an example command or URL longer than 80 characters
- A raw string literal may exceed 80 characters
- An #include or package line with a long path may exceed 80 characters
- A header guard may exceed 80 characters

2.2 White Space

Adding white space, such as spaces, indents, and blank lines, easily improve the readability of code without impacting performance.

2.2.1 Blank Lines

Use blank lines to separate code paragraphs. Use a single blank line to separate paragraphs. Use blank lines carefully, too many can decrease readability.

Example: C paragraphing

```
#define ONE 1
#define TWO 2
#define THREE 3

int main() {
  uint8_t number;

  for(uint8_t i = 0; i < THREE; i++) {
    number = i;
    printf("%d", number);
  }
}
```

2.2.2 Spacing

Appropriate spacing enhances readability especially in variables and operators. Use a single space to separate elements. Spaces improve readability and, in the second example, fixes a compilation error. A space shall be placed after a comma.

Example: Good Spacing

```
uint8_t number = *count / *index;
```

Example: Poor Spacing

```
uint8_t number=*count/*index; //Error: "/*" starts a comment
```

Example: Comma Spacing

```
uint8_t number = addNumbers(count, index)
```

2.2.3 Indentation

An indent shall be 2 spaces long. Use indents to show the logical structure of the code such as execution level. Set your editor to emit spaces when tab is hit

Example: Indentation

```
int main() {
  uint8_t c;
  while((c = getchar()) != EOF) {
    putchar(c);
  }
}
```

2.3 Comments

Use comments to provide information that a person could not easily discern from reading the code. Assume that your comments are for a student who has completed CPTS 122 and understands object oriented programming. Comments follow the Doxygen documentation standard. To create a Doxygen comment, type "/**" then hit [Enter].

Comments can take three forms: block, short, and inline. Examples of each follow below. Use block comments before in the document and function headers. Use short comments appended to a line of code to describe its function. Use inline comments before a line of code when an equivalent short comment is too long. All short comments in a paragraph shall be indented to align their start.

Example: Block Comment

```
/** Initializes all subsytems of the satellite
 * @param time - current time
 */
void init(Time time);
```

Example: Short Comment

```
uint8_t i = (random() % 10) + 1; // Generates a number on [1,10]
```

Example: Inline Comment

```
// Gets the green channel of the pixel at (col, row) of image
uint8_t green = (image.getRGB(col, row) >> 8) & 0xFF;
```

2.3.1 Document and Function Headers

Each document shall have a header that provides details about its contents, other information. See section 3.1 for more details.

Each function shall have a header that provides details about its function, arguments, return parameter, and other information. See section 4.1 for more details.

2.4 Meaningful names

Choose names for file, functions, constants, or variables that are meaningful and readable.

Naming Guidelines

- Choose names with meanings that are precise and use them consistently
- Avoid abbreviations that form letter combinations that may suggest unintended meanings. e.g. don't abbreviate "inch" to "in" as "in" suggests input
- Generally, use camel case. See that appropriate following section following for proper capitalization techniques.
- Assign names that are unique
- Use longer names to improve readability and clarity, but not too long
- Do not rely on letter case to make a name unique
- Acronyms in names shall be upper case. Except if the acronym is the beginning of a camel case name, the acronym shall be lower case.

Some standard short names are listed below and are acceptable for use if its meaning is clear. Using a longer, more explicit, name is recommended.

Standard Short Names

- c character
- i, j, k indices
- n counter
- p pointer
- s string
- buf buffer

2.4.1 Files

Files shall be descriptive of its contents. Files shall be organized into logical folders. A file located inside a folder should not have the folder's name in

its name. Files that contain classes shall be named the same as the class, using title case. All other files should be named descriptively, using camel case.

Example: Path and File Name

```
/src/tools/SWODebugPrint.cpp
/src/Events/pmicMessage.cpp
```

2.4.2 Classes and Objects

Classes and objects shall be named using title case. An object of a class, if singleton, shall be named identical to the class except using camel case. Typedef names shall be the same for the type specifier and declarator.

Example: Classes and Objects Naming

```
class OVCamera {};
typedef struct CameraReg {} CameraReg;
PMIC pmic;
I2C i2c;
```

2.4.3 Functions

Functions, expect constructors and destructors, shall be named using camel case.

Example: Function Naming

```
uint8_t getPowerLevel();
void pmicMessage(uint8_t *data, uint16_t dataLength);
```

2.4.4 Variables

Variables shall be named using camel case. When naming internal variables, do not duplicate the name of global variables. Expect in constructors, and mutators where the parameter's names should duplicate the global variable. In separate functions, variables with the same name may be used only if they share the same meaning.

Example: Variable Naming

```
uint8_t id;

void setId(uint8_t id) {
  this.id = id;
}

void generateID() {
  uint8_t tempID = random();
  id = tempID;
}
```

2.4.5 Constants

Constants, including #define and const variables, shall be named with upper snake case.

Example: Constants Naming

```
const uint8_t UNIQUE_NUMBER = 2;
#define I2C_ADDR_PMIC 0x69
```

3 File Organization

3.1 File Header

A file header introduces the file to the reader. Every file must have a header. The file header is separated into two parts: license boilerplate, and file details. The file header template used by Cougs in Space follows below. If significant changes are performed on a file by an author not listed in the file header, add the collaborator's name or consider removing the author tag.

See section 8.1 for a copyable version of the license boilerplate.

Example: File Header License Boilerplate

```
/*****************************************************************************
 * Copyright (c) 2017 by Cougs in Space - Washington State University         *
 * Cougs in Space website: cis.vcea.wsu.edu                                  *
 *                                                                           *
 * This file is a part of flight and/or ground software for Cougs in Space's *
 * satellites. This file is proprietary and confidential.                    *
 * Unauthorized copying of this file, via any medium is strictly prohibited. *
 *****************************************************************************/
```

Example: File Header Details

```
/**
 * @file OVCamera.cpp
 * @author Bradley Davis
 * @date 8 Dec 2017
 * @brief Configures and grabs frames from a camera
 *
 * Configures an OmniVision camera using I2C. Grabs frames using
 * DCMI interface provided by STM32 libraries. Writes raw pixel
 * data to a file.
 */
```

3.2 Guard Code

Each header file shall have preprocessor guard code to prevent multiple inclusion. The guard code shall be #ifndef style to be compatible across editors. The #define is [PATH]_[TO]_[FILE]_H_.

Example: Guard Code

```
#ifndef SRC_DRIVERS_OVCAMERA_H_
#define SRC_DRIVERS_OVCAMERA_H_
…
#endif /* SRC_DRIVERS_OVCAMERA_H_ */
```

3.3 Include Directives

Includes and imports are located directly after the guard code. The order of includes are as follows.

Example: Include Statements

```
#include <mbed.h>
#include <other system headers>
#include "user header files"
```

3.4 Defines, Typedefs, and Constants

After including header files, define constants, types, and macros that should be accessible to the rest of the file. Include the following, in the sequence shown:

- Enums
- Typedefs
- Constant macros (#define identifier token-string)
- Function macros (#define identifier (identifier, identifier, ...) token-string)
- Simple constants (const modifier)

3.5 External Data Declarations and Definitions

External data declarations are located directly after function macros.

3.6 Sequence of Functions

This section provides guidelines on how functions shall be arranged within a file. See section 4 for the organization of information within a function.

- If a file contains the main function, it should be the first function of the file
- Class constructors and destructors should be the first functions of the file, following the main function if present
- Arrange functions in a breadth-first approach (functions on similar levels of abstraction are together) rather than depth-first (functions defined as soon as possible before or after their calls.
- Mutators and accessors are the last functions in a file

4 Function Organization

This section describes guidelines for organization of information within a function.

4.1 Function Header

Every function should have a header, in Doxygen format, that introduces the function. It shall contain, in the following order, a description, details on arguments, and details on the return value. The function header shall only appear above the function's definition.

Example: Function Header

```
/**
 * Sends a frame exposure request for the specified time
 * @param ms of exposure duration
 * @return 0 on success, 1 on failure
 */
uint8_t OVCamera::activiateShutter(uint16_t ms) {…}
```

4.2 Function Arguments

Define function arguments in a logical order. A pointer to an array of variables precedes the length argument, see example below.

Example: Function Arguments

```
uint8_t pmicMessage(uint8_t *data, uint16_t dataLength);
```

4.3 Internal Variable Declaration

Define internal variables, also known as local variables, directly after opening a function. Except loop indices which may be defined in the loop call.

Example: Function Arguments

```
uint8_t pmicMessage(uint8_t *data, uint16_t dataLength) {
  uint8_t buf[64];

  for uint8_t i = 0; i < dataLength; i++) {
    …
  }
  …
}
```

4.4 Statement Paragraphing

Separate logical groups of lines with a single empty line to improve readability.

4.6 Return Statement

Use a single return statement at the end of the function to return a value to the function call. Multiple returns may be used for error checking or if the use of a single return statement results in convoluted logic.

Example: Return Statement

```
uint8_t pmicMessage(uint8_t *data, uint16_t dataLength) {
  if(dataLength = 0) {
    return ERROR;
  }
  …
  return result;
}
```

5 Data Types, Operators, and Expressions

This section describes the proper way to format constants and variables definitions and declarations, and data encapsulation techniques. The general guidelines are:

- Define one variable per line
- Use descriptive and unique names
- Group related variables and constants together

5.1 Variables

When declaring variables of the same type, declare each on separate lines. Unless the variables are self-explanatory are related. Add a comment after a declaration if the name is does not give sufficient information about the contents of the variable.

Example: Variable Declaration

```
uint8_t day, month, year;
uint8_t row, column;

FILE *rawImage; //File pointer to image containing raw pixel data
```

All integer type variables shall use the declaration as follows:

Example: Integer Types

```
[unsigned]int[number of bits]_t
uint8_t a; //a can store 0x00 to 0xFF
int16_t b; //b can store -0x8000 to 0x7FFF
```

5.2 Constants

When declaring constants, use snake case. Use enumeration types when declaring constants in the same group. Use const modifier instead of #define for declaring simple constants. #define tells the preprocessor to substitute a token for an identifier which can cause issues, see below.

Example: Constants Declaration

```
typedef enum ADCSState {
  STOPPED,
  MOVING
} ADCSState;

#define VERSION “1.2.3”
#define QUEUE_LENGTH 32
```

Example: #define vs. const Modifier

```
#define WIDTH 10 + 10
const uint8_t SIZE = 10 + 10; //Evaluates to 20
…
area = WIDTH * WIDTH; //Evaluates to 120 = 10 + 10 * 10 + 10
area = SIZE * SIZE;   //Evaluates to 200 = (10 + 10) * (10 + 10)
```

5.3 Variable Declaration and Definition

Floating point numbers shall have at least one number on each side of the decimal point. Floats shall end with a f (lowercase f) while doubles have no suffix. Hexadecimal numbers shall be capitalized and begin with 0x (zero, lower case x). Add leading zeros such that the number of bytes is communicated. Long constants end with a L (uppercase L).

Example: Numerical Constants

```
0.5, 5.0f, 1.0e+99
0xFFFF, 0x00A0
124512L
```

Variables may be initialized either where it is declared or just prior to use. Variables that are used within a page of their declaration shall be initialized where it is declared. Variables that are used several pages after their declaration are better initializes just prior to use.

Example: Variable Initialization

```
uint8_t sampleCount = 0;
uint8_t maxTemp;

sampleCount = thermometer.getCount();
…
//Several pages later
maxTemp = 0;
for(uint8_t i = 0; i < sampleCount; i++) {
  uint8_t temp = thermometer.getTemp();
  if(temp > maxTemp){
    maxTemp = temp;
  }
}
```

5.4 Type Conversion

Type conversion happen by default when using mixed types in an expression. Use the cast operator to make type conversion explicit rather than implicit.

Example: Type Conversion

```
float power = 0.0f;
uint16_t rawPower = powerMeter.getRawPower();
power = (float) rawPower * RATIO_POWER;
```

5.5 Pointer Types

When declaring a pointer, put the pointer qualifier (*) directly before the variable rather than with the type.

Example: Pointer Types

```
uint8_t value = 0;
uint8_t *address = &value;
```

5.6 Pointer Conversions

Programs should not contain pointer conversions, except for the following:

- **NULL** may be assigned to any pointer type
- **Allocation Functions** (i.e. malloc) return a void* and thus should be properly casted before assigning to any pointer. Ensure to use sizeof when passing the number of bytes to allocate

5.7 Operator Formatting

- Do not put space around the primary operators: ->, ., and [ ]

```
a->b  c.d  e[i]
```

- Do not put a space before parenthesis following function names. Within parenthesis, do not put spaces between the expression and parenthesis

```
function(a, b)
```

- Do not put a space between unary operators and their operands

```
!a  ~b  ++i  *c  &d
```

- Exception, do put a space between a cast and its operand

```
(float) integerValue
```

- Always put a space around assignment operators

```
a = b
```

- Always put a space around conditional operators

```
a = (a >= b) ? b : a;
```

- Commas shall have one space after them

```
function(a, b, c, d, e)
```

- Semicolons shall have one space after them

```
for(uint8_t i = 0; i < COUNT; i++)
```

- For other operators, generally put one space before and after the operator

```
a = (b + c) * d;
```

- Use parenthesis to liberally indicate the precedence of operations

```
a = ((b % c) + (c % b)) * d;
```

- Split a string of conditional operators that will not fit onto one line, breaking after the logical operators

```
if((a->b != c.b) &&
    (currentIndex < COUNT) &&
    c.isValid(a->b)) {
  c.setB(a->b);
}
```

5.8 Assignment Operators and Expressions

The assignment operator "a = b" has a value that can be used in expressions. It is recommended to use this feature sparingly. The most common use follows:

Example: Assignment Operator Expression

```
while((c = getChar()) != EOF){
  putChar(c);
}
```

5.9 Conditional Expressions

The conditional expression, which condenses if then else statements, should be used only if the resultant expression is easy to understand and maintain

Example: Conditional Expressions

```
if( a > b) {
  z = a;
```

```
} else {
  z = b;
}

//Condensed version
z = (a > b) ? a : b;

//Avoid condensed version here
z = (a > b) ? a + f(b) : f(a) + b;
```

5.10 Precedence and Order of Operations

There are dozens of precedence rules for every language. Instead of memorizing them, use the short version:

- * / % come before + and –
- Put ( ) around everything else

6 Statements and Control Flow

This section describes how to organize statements into logical thoughts and how to format various kinds of statements. The general guidelines are as follows:

- Use blank lines to organize statements into logical paragraphs
- Limit the complexity of statements. If it improves readability, spit complex statements into several simpler statements
- Indent to show the logical structure

6.1 Sequence Statements

6.1.1 Statement Placement

Only put one statement per line, except in for loop statements.

```
switch (adcsState) {
  case STOPPED:
    adcs.enable();
    break;
  case MOVING:
    adcs.update();
    break;
  default:
    adcs.disable();
    break;
}
```

It is recommended that explicit comparison is used even if the comparison value will never change.

```
if (adcs.move(1.2)) {
  //Resulted in errors
}
```

Rewrite as

```
if (adcs.move(1.2) != 0) {
  //Resulted in errors
}
```

6.1.2 Braces

Blocks of statements that are surrounded in braces should place the opening brace on the same line as the previous statement and place the closing brace on the next line after the last statement in the block. Lack of braces in if statements is discouraged

Example: Brace Placement

```
if (a != b) {
  a = 2 * b;
}

//Discouraged
if (a != b)
  if(a > b)
    a = b + 1;
  else
    a = b - 1;
else
  return a * b;
```

If a block is large, more than about 40 lines, comment the end brace to indicate which part of the process it is closing. Place the comment just before the closing brace if it improves readability and prefix the comment with "END:"

Example: Brace Comments

```
if (pixel == BLUE) {
  //A lot of lines
  …
  //END: pixel is BLUE
} else if (pixel == RED) {
  //A lot of lines
  …
  //END: pixel is RED
} else {
  //A lot of lines
  …
} //pixel is GREEN
```

6.2 Selection Control Statements

6.2.1 If Statements

Indent the body relative to the if statement. If and else shall be the same level of indentation. Use nested if statements if there are alternative actions, do not use nested if statements if only the clause contains actions.

Example: If Statements

```
if (a == b) {
```

```
  //Indent body
}

//Proper use of nesting
if (a == b) {
  object.setA(a);
  if (c == d) {
    b = object.getResult() + c;
  }
} else {
  object.setA(a + b);
}

//Improper use of nesting
if (a == b) {
  if (c == d) {
    if(a > c) {
      a = b + c + d;
    }
  }
} else {
  d = a;
}

//Change to a single if statement
if((a == b) && (c == d) && (a > c)) {
  a = b + c + d;
} else {
  d = a;
}
```

6.2.2 Switch Statements

Use the following guide to format switch statements. Comment fall through cases for maintainability. If a body of a case is longer that a few lines, consider replacing with a function for readability.

Example: Switch Statements

```
switch (value) {
  case a:
    //statements
    break;
  case b: //Fall through
  case c:
```

```
    doSomething();
    break;
  defaut:
    //statements
    break;
}
```

6.3 Iterative Control Statements

6.3.1 While Loop

Format while loops as seen below:

Example: While Loop

```
while (expression) {
  //statements
}
```

6.3.2 For Loop

Format for loops as seen below. If a for loop will not fit on one line, split into three lines.

Example: For Loop

```
for (expression) {
  //statements
}

for(uint32_t i = 0;
  i < SIZE && values[i] != NULL;
  i++) {
  //statements
}
```

6.3.3 Do While Loop

Format do while loops as seen below:

Example: Do While Loop

```
do {
  //statements
} while (expression)
```

6.4 Error and Exception Handling

Whenever possible, add error checking methods. Most standard functions return an integer representing the success of the execution of the function. When calling these functions, verify the function returns the success value and act accordingly. The try catch method should be used in Java when executing methods with possible exceptions.

All functions written for Cougs in Space shall return 0 (zero) for success or a non-zero number error code. Use unique values for different sources of error. List these values in the function header. The program should never abort as it puts the satellite in a dangerous state. Comment before an error return statement. Consider using #define to define error codes rather than using "magic numbers".

Example: Error and Exception Handling

```
/**
 * readReg
 * @param addr of register
 * @param val pointer to read register
 * @return ERROR_SUCCESS on success,
 *         ERROR_WRITE on failure to request register
 *         ERROR_READ on failure to read register
 **/
uint16_t readReg (uint8_t addr, uint8_t *val) {
  if (i2c.write(addr) != 0) {
    //Failed to request register
    return ERROR_WRITE;
  }

  if (i2c.read(val) != 0) {
    //Failed to request register
    return ERROR_READ;
  }

  return ERROR_SUCCESS;
}
```

6.4.1 Error Codes

Error codes are constrained to unsigned 8-bit integers.

```
ERROR_SUCCESS           (0x00) The operation completed successfully
ERROR_INVALID_ARGS      (0x01) The arguments are incorrect
ERROR_OUT_OF_MEMORY     (0x02) Not enough memory to complete the operation
ERROR_CRC               (0x03) Data error (cyclical redundency check)
ERROR_FILE_NOT_FOUND    (0x04) The system cannot find the specified file
ERROR_WRITE_PROTECT     (0x05) The media is write protected
ERROR_EOF               (0x06) Reached the end of the file
ERROR_NOT_SUPPORTED     (0x07) The request is not supported
ERROR_BUFFER_OVERFLOW   (0x08) The buffer is too small
ERROR_INVALID_PASSWORD  (0x09) Authentication failed
ERROR_WRITE             (0x0A) Failed during write operation
ERROR_READ              (0x0B) Failed during read operation
```

```
ERROR_NACK              (0x0C) No acknowledgement
ERROR_OPEN_FAILED       (0x0D) The system cannot open the device or file
ERROR_DISK_FULL         (0x0E) There is not enough space in the disk
ERROR_BUSY              (0x0F) The specified resource is in use
ERROR_ALREADY_EXISTS    (0x10) Cannot create a file that already exists
ERROR_QUEUE_OVERFLOW    (0x11) The queue is at maximum capactity
ERROR_WAIT_TIMEOUT      (0x12) The operation timed out
ERROR_WATCHDOG          (0x13) The watchdog was not pet properly
ERROR_INVALID_ADDRESS   (0x14) Attempt to access invalid address
ERROR_POWER_LIMITED     (0x15) Insuffient power to complete operation
ERROR_POWER_EMERGENCY   (0x16) Emergency power level reached
ERROR_COMMS_INTERRUPTED (0x17) Communication link lost
ERROR_UNKNOWN_COMMAND   (0x18) The requested command is unknown
```

7 Portability and Performance

This section explains guidelines on writing software which can easily be ported to another program or processor, and guidelines on performance of software.

7.1 Guidelines for Portability

- Write portable code first. Optimized code is often obscure and not always an optimization for another processor. Comment code that becomes obscure
- Some software cannot be ported to another device, consider using an intermediate handler to isolate the device specific code (i.e. camera class and specific camera drivers)
- Organize device-dependent and device-independent code separately so changing devices is easier
- Use sizeof( ) whenever writing the size of a variable instead of hard-coding it
- Become familiar with standard library functions. Reimplementation of these functions requires readers to analyze the new version and identify any differences
- Use #ifdef to hide nonportable code using centrally placed definitions

7.2 Guidelines for Performance

- Maintainable code is better that high-performance code
- If performance is not an issue, write easy to understand code
- When performance is important, comment possibly unclear code
- Minimize the number or open or closes and I/O operations if possible
- Free allocated memory as soon as possible
- To improve efficiency, use automatic increment ++ and decrement -- operators and the special operators *= and +=
- When passing objects or structures, use pointers or reference instead of a copy as this uses less stack space

8 Code Elements

Below are copyable elements intended to be added to code files. They are properly formatted for use in code editors.

8.1 License Boilerplate

```
/*******************************************************************************
 * Copyright (c) 2017 by Cougs in Space - Washington State University          *
 * Cougs in Space website: cis.vcea.wsu.edu                                    *
 *                                                                             *
 * This file is a part of flight and/or ground software for Cougs in Space's   *
 * satellites. This file is proprietary and confidential.                      *
 * Unauthorized copying of this file, via any medium is strictly prohibited.   *
 *******************************************************************************/
```

9 References

Nasa's C Style Document
http://homepages.inf.ed.ac.uk/dts/pm/Papers/nasa-c-style.pdf

Google's C++ Style Document
https://google.github.io/styleguide/cppguide.html

Doxygen Documentation
http://www.stack.nl/~dimitri/doxygen/