ROS C++ STYLE GUIDE

Version 1.0, January 2018

1 Foreword

This document defines a style guide to be followed in writing C++ code for Robot Operating System within the Institute of Automation. The style guide is based on the Google C++ style guide and official ROS style guide. The readers are encouraged to read Google C++ style guide as a fundamental. Changes from official ROS style guide are highlighted by red color. All new packages developed in the Institute of Automation should conform to this guide.

2 Autoformatting of ROS Code

Why waste your valuable development time formatting code manually when we are trying to build amazing robots? Use a robot to format your code using "clang-format". You can find an auto formatting script for this style guide here.

3 Naming

The following shortcuts are used in this section to denote naming schemes:

CamelCased: The name starts with a capital letter, and has a capital letter for each new word, with no underscores.

camelCased: Like CamelCase, but with a lower-case first letter

under_scored: The name uses only lower-case letters, with words separated by underscores. (yes, I realize that under_scored should be underscored, because it's just one word).

ALL_CAPITALS: All capital letters, with words separated by underscores

3.1 Packages

ROS packages are under_scored.

This is not C++-specific, e.g.: youbot_gazebo.

3.2 Topics / Services

ROS topics and service names are under_scored.

This is not C++-specific, e.g.: /vel_cmd, /switch_off_motors.

3.3 Files

All files are under_scored.

Source files have the extension .cpp while header files have the extension .h. For every .cpp file, there is a corresponding .h file with the same name. In general they take the module name, see Chapter 16.

Be descriptive, e.g., instead of laser.cpp, use hokuyo_topurg_laser.cpp.

If the file primarily implements a class, name the file after the class. For example the class ActionServer would live in the file action_server.h.

3.4 Libraries

Libraries, being files, are under_scored.

Don't insert an underscore immediately after the lib prefix in the library name.

E.g.:

```
lib_my_great_thing    ## Bad
libmy_great_thing     ## Good
```

3.5 Classes / Types

Class names (and other type names) are CamelCased

E.g.: class ExampleClass;

Exception: if the class name contains a short acronym, the acronym itself should be all capitals, e.g.: class HokuyoURGLaser;

Name the class after what it is. If you can't think of what it is, perhaps you have not thought through the design well enough. Compound names of over three words are a clue that your design may be unnecessarily confusing.

See also: Google:Type Names

3.6 Variables

In general, variable names consist only of nouns and adjectives. They are under_scored and begin with specified qualifier to identify the data type, e.g.: ui_voltage_level, f_max_acc. Possible qualifiers are listed in Table 3.1.

Table 3.2: Prefix for data type

Data type	**Qualifier of Data type**
usigned char	uc
signed char	sc
char	c
int	i
unsigned int	ui
unsigned long	ul
long	l
unsigned char *	puc
signed char *	psc
char *	pc
unsigned long *	pul
long *	pl
fload	f
double	d
long double	ld
float *	pf
double*	pd

long double *	pld
struct *	ps
function pointer	pfn
void	v

Be reasonably descriptive and try not to be cryptic. Longer variable names don't take up more space in memory.

Integral iterator variables can be defined without data type qualifier, such as i, j, k. i is always on the outer loop, j on the next inner loop and so on.

The pointer-qualifier "*" must stay near the name, not the data type, e.g.: unsigned int *ui_voltage_level.

STL iterator variables should indicate what they're iterating over, e.g.:

```
std::list<int> pid_list;

std::list<int>::iterator pid_it;
```

Alternatively, an STL iterator can indicate the type of element that it can point at, e.g.:

```
std::list<int> pid_list;

std::list<int>::iterator int_it;
```

3.7 Constants

Constants are in principle not allowed, define macros with ALL_CAPITALS instead of defining constants.

E.g.:

```
#define SAMPLE_CONST 7
```

Member variables

Variables that are members of a class (sometimes called fields) are under_scored, with a trailing underscore added.

E.g.:

```
int i_example_int_;
```

3.8 Global variables

Global variables should almost never be used (see below for more on this). When they are used, global variables are under_scored with a leading g_ added.

E.g.,:

```
// I tried everything else, but I really need this global variable

int g_i_shutdown;
```

3.9 Namespaces

Namespace names are under_scored and based on module/package name.

3.10 Function / Methods

In general, function and class method names are camelCased, their arguments are under_scored (variables). The name begins with a prefix to identify the returned data type (refer to Table 3.3), e.g.:

```
int iExampleMethod(int i_example_arg);
```

Functions and methods usually perform an action, so their names should consist of a verb and a noun. The name must clearify what they do:

```
int iCheckForErrors()       //instead of iErrorCheck()

void vDumpDataToFile()      //instead of vDataFile().
```

Classes are often nouns. By making function names verbs and following other naming conventions programs can be read more naturally.

Table 3.4: some preferred verbs for actions

Verb	**possible meaning of the action**
Add	add an item to a list
Adjust	align a value
Apply	assign a new value
Calc	execute an algorithm to get a value
Check	verify sth.
Clear	reset to default state
Ctrl	control sth.
Cvrt	convert
Decr	reduce the value
Get	read date from another module
Read	read date from hardware
Init	initialization
Lookup	get a predefined value in a table
Manage	
Reset	data or state reset
Start	start a process
Stop	stop a process
Test	run a test
Update	change value or state
Propagate	move on to next step
Is	ask for a logical status

Most variables and functions should be declared inside classes. The remainder should be declared inside namespaces.

4 Formatting

Your editor should help you format code in a consistent style, as described in this document. It is well showed in ROS editor help, how to set up your favorite editor including Vim, Emacs, Eclipse, indent and Qt Creator.

4.1 Code Block

Please indent each code block by at least two space, never use literal tab characters (this can be done in the setting of your editor). Braces, both open and close, go on their own lines (no "cuddled braces"). E.g.:

```
if(a < b)
{
  a=b// do stuff
}
else
{
  b=a// do other stuff
}
```

Braces can be omitted if the enclosed block is a single-line statement, the block stay in the same line after the construction. However, always include the braces if the enclosed block is more complex.

Hier is a larger example:

```
 1 /*
 2  * A block comment looks like this...
 3  */
 4 #include <math.h>
 5 class Point
 6 {
 7 public:
 8   Point(double xc, double yc) :
 9     x_(xc), y_(yc)
10   {
11   }
12   double distance(const Point& other) const;
13   int compareX(const Point& other) const;
14   double x_;
15   double y_;
16 };
17 double Point::distance(const Point& other) const
18 {
19   double dx = x_ - other.x_;
20   double dy = y_ - other.y_;
21   return sqrt(dx * dx + dy * dy);
22 }
23 int Point::compareX(const Point& other) const
24 {
25   if (x_ < other.x_)
26   {
27     return -1;
28   }
29   else if (x_ > other.x_)
30   {
```

```
31      return 1;
32    }
33    else
34    {
35      return 0;
36    }
37  }
38  namespace foo
39  {
40  int foo(int bar) const
41  {
42    switch (bar)
43    {
44      case 0:
45        ++bar;
46        break;
47      case 1:
48        --bar;
49      default:
50      {
51        bar += bar;
52        break;
53      }
54    }
55  }
56  } // end namespace foo
57
```

4.2 Line Length

Maximum line length is 120 characters.

4.3 #ifndef Guards

All headers must be protected against multiple inclusion by #ifndef guards, the name of the compiler switcher builds up as _FILE_NAME_CAPTAL_H_. E.g.:

```
#ifndef _CHARGE_CONTROL_H_
#define _CHARGE_CONTROL_H_
...
#endif
```

This guard should begin immediately after the license statement, before any code, and should end at the end of the file.

5 Commentary & Documentation

The source code must be coded with comments, but it is not necessary to leave a comment for each line. The comment must declare “what” happened, “how” does it happened and how the parameters are interpreted.

In the beginning of every file, a file header comment according to Doxygen guideline have to be left. Each function receive a block comment to explain its tasks, inputs and returned values.

A comment must be there, when

- declaration of variables (if there is no Doxygen comment)
- Query datas

- loops and branches
- complex calculation

Comments, which are related to a single line, should be written either over this line or in the end of this line. Only single line comment is allowed to use "//", others must use "/* … */". If the variable is a physical value, the unit must be given. Comments are not allowed to be nested.

All functions, methods, classes, class variables, enumerations, and constants should be documented. Doxygen is used to auto-document the source code. Doxygen parses your code, extracting documentation from specially formatted comment blocks that appear next to functions, variables, classes, etc. Doxygen can also be used to build more narrative, free-form documentation.

See the rosdoc page for how to auto-generate documentation with ROS package semantics. Refer to XXX for Doxygen guideline.

6 Console Output

Instead of printf, cout or their friends, use rosconsole for all the outputting needs. It offers macros with both printf and stream-style arguments. rosconsole output goes to corresponding console window, in which the program is running. We prefer rosconsole because it is

- color-coded
- controlled by verbosity level and configuration file
- published on the topic /rosout, and thus viewable by anyone on the network.
- optionally logged to disk
- sorted by different levels (normal, warning, error etc.)

7 Preprocessor directives (#if vs #ifdef)

For conditional compilation (except for the #ifndef guard in header explained in Chapter 4.3), always use #if instead of #ifdef.

```
#if DEBUG
        temporary_debugger_break();
#endif
```

The code might be compiled with turned-off debug info lik:

```
cc  -c lurker.cpp  -DEBUG=0
```

Always use #if, if you have to use the preprocessor. This works fine and does the right thing, even if DEBUG is not defined at all.

8 Output Arguments

Output arguments to methods or functions (i.e. variables that the function can modify) are passed by pointer, not by reference. E.g.:

```
int iGiveExampleMethod(int i_input, float *pf_output)
```

By comparison, when passing output arguments by reference, the caller (or subsequent reader of the code) can't tell whether the argument can be modified without reading the prototype of the method.

9 Namespaces

Use of namespaces to scope your code is encouraged. The naming advice is given in Chapter 3.9. Use of using-directives in header files is strictly forbidden. It pollutes the namespace of all code that includes this header.

Namespaces do not add an extra level of indentation.

It is acceptable to use using-directives in a source file, but it is preferred to use using-declarations, which pull in only the names you intend to use.

E.g.; instead of this:

```
using namespace std;  // Bad, because it inports all names from std and polute the namespace
```

do this:

```
namespace std::list  // I want to refer to std::list as list
{
}  // namespace std::list
namespace std::vector  // I want to refer to std::vector as vector
{
void vFoo();
}  // namespace std::vector
```

10 Inheritance

Inheritance is the appropriate way to define and implement a common interface. The base class defines the interface, and the subclasses implement it.

Inheritance, used to provide common code from a base class to subclasses, is discouraged. In most cases, the subclass could instead contain an instance of the base class and achieve the same result with less potential for confusion.

When overriding a virtual method in a subclass, always declare it to be virtual, so that the reader knows what's going on.

Multiple inheritance is strongly discouraged, as it can cause intolerable confusion.

11 Functions

11.1 Parameters

When defining a function, parameter order is: inputs, then outputs, at last input&output. Input parameters are usually values or const references, while output and input/output parameters will be pointers to non-const.

All parameters passed by reference must be labeled const.

```
void vFoo(const string &rs_in, string *ps_out);
```

11.2 Function Length

No hard limit is placed on function length, but prefer small and focused functions. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

12 Exceptions

Exceptions are the preferred error-mechanism as opposed to returning integer error codes. All exceptions, which can be thrown by your package/function/method, must be documented properly. Exceptions are not allowed in destructors or callbacks invoked indirectly.

When your code can be interrupted by exceptions, ensure that held resources will be deallocated when stack variables go out of scope.

13 Assertions

Use assertions to check preconditions, data structure integrity, and the return value from a memory allocator. Assertions are better than writing conditional statement that will rarely, if ever, be exercised.

Do not call assert() directly, instead use one of these functions, declared in ros/assert.h (part of rosconle package):

```
/** ROS_ASSERT asserts that the provided expression evaluates to
 * true.  If it is false, program execution will abort, with an informative
 * statement about which assertion failed, in what file.  Use ROS_ASSERT
 * instead of assert() itself.
 */
#define ROS_ASSERT(expr) ...
```

```
/** ROS_BREAK aborts program execution, with an informative
 * statement about which assertion failed, in what file. Use ROS_BREAK
 * instead of calling assert(0) or ROS_ASSERT(0).
 */
#define ROS_BREAK() ...
```

Do not do work inside an assertion; only check logical expressions. Depending on compilation settings, the assertion may not be executed.

14 Tests

See gtest.

14.1 Obtaining gtest

Gtest has been converted to a rosdep and is available in ros_comm

14.2 Google Test (gtest)

We use GoogleTest, or gtest, to write unit tests in C++. The official documentation for gtest is here:

https://github.com/google/googletest

Also refer to http://www.ibm.com/developerworks/aix/library/au-googletestingframework.html

These pages give some tips, and examples for writing and calling unit tests in ROS code. For language-independent policy and strategy for testing, see UnitTesting.

14.3 Code structure

By convention, test programs for a package go in a `test` subdirectory. For a simple package, it is usually sufficient to write a single test file, say `test/utest.cpp`.

14.4 Writing tests

The basic structure of a test looks like this:

```
 1 // Bring in my package's API, which is what I'm testing
 2 #include "foo/foo.h"
 3 // Bring in gtest
 4 #include <gtest/gtest.h>
 5
 6 // Declare a test
 7 TEST(TestSuite, testCase1)
 8 {
 9 <test things here, calling EXPECT_* and/or ASSERT_* macros as needed>
10 }
11
12 // Declare another test
13 TEST(TestSuite, testCase2)
14 {
15 <test things here, calling EXPECT_* and/or ASSERT_* macros as needed>
16 }
17
18 // Run all the tests that were declared with TEST()
19 int main(int argc, char **argv){
20   testing::InitGoogleTest(&argc, argv);
21   return RUN_ALL_TESTS();
22 }
```

14.5 Test naming conventions

Each test is a "test case," and test cases are grouped into "test suites." It's up to you to declare and use test suites appropriately. Many packages will need just one test suite, but you can use more if it makes sense.

Test suites are CamelCased, like C++ types

Test cases are camelCased, like C++ functions

14.6 Building and running tests

Add your test to your package's `CMakeLists.txt` like so:

```
catkin_add_gtest(utest test/utest.cpp)
```

These calls will cause the `utest` executable to be built during the main build (a simple `make`), and will put it in TBD. Note that unlike rosbuild, specifying directory hierarchy in the target declaration is not allowed. You can run the test with `make test`. You can also just run the test executable directly, e.g.:

```
./bin/test/utest
```

14.7 Test Out

The console output from a test will look something like this:

```
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from MapServer
[ RUN      ] MapServer.loadValidPNG
[       OK ] MapServer.loadValidPNG
[ RUN      ] MapServer.loadValidBMP
[       OK ] MapServer.loadValidBMP
[ RUN      ] MapServer.loadInvalidFile
[       OK ] MapServer.loadInvalidFile
[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran.
[  PASSED  ] 3 tests.
```

When run as `make test`, each test will also generate an XML file, in:

- $ROS_ROOT/test/test_results/<package-name> if you're running ROS cturtle
- ~/.ros/test_results if you're running ROS Diamondback.
- To make sure where they are generated on your system, you can run: rosrun rosunit test_results_dir.py

The generated XML looks something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite tests="3" failures="1" disabled="0" errors="0" time="25" name="AllTests">
  <testsuite name="MapServer" tests="3" failures="1" disabled="0" errors="0" time="25">
    <testcase name="loadValidPNG" status="run" time="24" classname="MapServer">
      <failure message="/Users/gerkey/code/ros-pkg/world_mod-
els/map_server/test/utest.cpp:56&#x0A;Failed&#x0A;Uncaught exception : This is OK on OS X"
type=""/>
    </testcase>
    <testcase name="loadValidBMP" status="run" time="0" classname="MapServer" />
    <testcase name="loadInvalidFile" status="run" time="1" classname="MapServer" />
  </testsuite>
</testsuite>
```

We will eventually have a way of parsing and rendering these results into a web-based dashboard.

15 Deprecation

To deprecate an entire header file within a package, you may include an appropriate warning:

```
#warning mypkg/my_header.h has been deprecated
```

To deprecate a function, add the deprecated attribute:

```
ROS_DEPRECATED int myFunc();
```

To deprecate a class, deprecate its constructor and any static functions:

```
class MyClass
{
public:
  ROS_DEPRECATED MyClass();

  ROS_DEPRECATED static int myStaticFunc();
};
```

16 Program Structure

16.1 module

The entire program is expediently implemented in various modules. Each module consists of a source file and a header file. For each source (.cpp), there should be a corresponding file (.h) with the same module name. An abbreviation should be assigned to each module and shown in the block comment of the file. To allocate the function `main()` efficiently, the module including `main()` must have the key word `main_module`.

Example:

```
Module  -  Battery charge control

source file: charge_control.cpp
header file: charge_control.h

module abbreviation: CHA
```

The header file descripts normally the module interface, it is user-oriented. On the opposite, the source file implements the module and is developer-oriented. For this reason, the comments about module interface should be left in .h file and module intern function should be explained in .cpp file. There must be only one comment for each function to reduce maintenance effort and error potential.

16.2 Structure of a Header File

All of a project's header files should be listed as descendants of the project's source directory without use of UNIX directory shortcuts . (the current directory) or .. (the parent directory).

Use standard order for readability and to avoid hidden dependencies: Related header, C library, C++ library, other libraries' .h, your project's .h.

In dir/foo.cpp, whose main purpose is to implement the stuff in dir/foo.h, order your includes as follows:

1. dir/foo.h.
2. C system files.
3. C++ system files.
4. Other libraries' .h files.
5. Your project's .h files.

Example:

File: foo/server/fooserver.cpp

```
#include "foo/server/fooserver.h"

#include <sys/types.h>
#include <unistd.h>

#include <hash_map>
#include <vector>

#include "base/basictypes.h"
#include "base/commandlineflags.h"
#include "foo/server/bar.h"
```

file: charge_control.h

```
************************************
block comment w.r.t. Doxygen-guideline
************************************
#ifndef _CHARGE_CONROL_H_
#define _CHARGE_CONROL_H_
//=====================================================================
// includes
#include
//=====================================================================
// data types
//=====================================================================
// interface macros and variables
// ====================================================================
// module functions
#endif // _CHARGE_CONROL_H_
/*********************************************************************/
```

16.3 Structure of a Source File

file: charge_control.c

```
************************************
block comment w.r.t. Doxygen-guideline
************************************
// ====================================================================
// includes
#include "charge_contol.h"

// ====================================================================
// external module variables

// ====================================================================
// internal module variables

// ====================================================================
// module functions

// = eof: charge_control.c
/*********************************************************************/
```