Programming Guide

Programming Guide for the Robotics and Mechatronics Group at Clemson University

DOE Project Team, August 1998/October 2000

1. Introduction

This manual documents coding conventions used by the Clemson University Control and Robotics Group (CRB). Any software developed for CRB use should follow these conventions. If you have any questions please contact Markus Loffler (loffler@ces.clemson.edu).

1.1 Motivation

The software used and developed by CRB students in the course of their research has become complex enough to require the use of other students' code, in the form of source code, libraries, etc. Software projects may require multiple students to work simultaneously on the same code. Or a project may be so complex that it is passed on from one student to another. Often a student will begin with a program written by another student, and extend it to meet his needs.

A coding convention simplifies all of these tasks by providing a common set of rules for developing source code, so that students working with someone else's code are not confronted with unfamiliar, poorly structured code. A coding convention may also permit the use of automatic document generation software, such as Doxygen. While requiring some extra work initially, observing a coding convention will always save time and effort in the long run.

1.2 Doxygen

Doxygen is an automatic documention generator. It uses comments in the C++ source files and creates a very nice manual in html and pdf format. If you develop a class, only the interface of this class will be documented (the interface is: what functions and public data members does your class define). You find the interface in the <ClassName>.hpp file.

When developing software, students are to develop the manual first. Consequently a student must first write the <ClassName>.hpp file, then process this file with doxygen to create the interface portion of the manual.

1.3 Software Engineering

The code we write is complex and used and maintained by many people, often even after we leave CRB. To create good code and and facilitate working together, we use the following schemes.

1.3.1 Creation of a Class Library

1. The project manager defines the task for you. This is usually the implementation of a class or several classes (e.g. write a Matrix class)
2. You roughly sketch the interface of the class: What functions will it offer, what data will it have. (e.g. a Matrix class will have functions to multiply, add, etc.). Your project manager will correct you.
3. You create the interface (the <ClassName>.hpp file) and process it with doxygen. Your project manager will look at the manual and correct you.
4. You think about your design of the implementation and discuss it with your project manager. This is critical, as he has more experience, and can often forsee problems you can not, saving you the time and effort of having to start over.
5. You implement the class in <ClassName>.cpp.
6. You also write a test program <ClassName>.t.cpp (or multiple test programs if required) which will demonstrate and test your class. You are responsible to test your code thoroughly.
7. Your project manager does period code reviews of your code while you are writing it and comes back with corrections.

1.3.2 Creation of an Application

1. The project manager defines the task for you.
2. You create a manual of your application with Word. Your project manager will look at the manual and correct you. This manual should describe the functionality of your application. An application can be either GUI based (i.e., you include pictures [can be sketches] of windows, dialogs, etc., and describe where to click etc.) or command line based (i.e., you describe parameters, usage, etc.)
3. You think about your design of the implementation and discuss it with your project manager. This is critical, as he has more experience, and can often forsee problems you can not, saving you the time and effort of having to start over.
4. You implement the application.
5. You test your application. You are responsible to test your code thoroughly.
6. Your project manager does period code reviews of your code while you are writing it and comes back with corrections.

2. Names and Code Formatting

2.1 File Headers and Overall Format

If you start out with a new class, use the `create_class` script to create the template files

E.g., `create_class Nose`

will create:

```
Nose.hpp
Nose.cpp
examples/Nose.t.cpp
```

Then edit those files.

2.2 Indention and Brackets

Use one (1) tab for each level of indention instead of spaces.

By using a tab instead of spaces, and agreeing to use only 1 tab per indention level, we can view the code with however many spaces per indention level we want (in *vi* use the *set tabstop=x* command to set how many spaces a tab actually indents – this can be placed in your *.exrc* file in your home directory. (*.exrc* is a file executed by *vi* when you start *vi*.)

The ⇒ symbol below represents one tab character.

If there is just one command, it is indented without brackets in the next line:

```
if (dataMode == 1)
⇒dataCounter++;
```

Otherwise, the indention starts with a bracket in a new line:

```
if (dataMode == 1)
{
⇒dataCounter++;
⇒if (dataCounter > 10)
⇒{
⇒⇒cout << "Yeah" << endl;
⇒⇒dataCounter = 0;
⇒}
}
```

This format is valid for: `if`, `while`, `do`, `for`, function definitions, class definitions.

Use spaces to align comments ("-" represents a space):

```
if (dataMode == 1)
{
⇒dataCounter++;------// As the data counter is 32 bit, we
⇒--------------------// don't need to be worried about an overrun

}
```

2.3 Symbol Names

> Try to use self-explaining names, even if they are longer. Remember, your code will be written once, but read MANY TIMES!

e.g. instead of

```
crTempObj(…)
```

use

```
createTemperatureObject(…)
```

2.3.1 Standard Variable Names

There are names or parts of variable names that occur over and over again for different projects. Use the following standards for those names:

Description	**Name**	**Example**
Number of…	`num…`	`numListElements`
Size [number of bytes]	`…Size`	`bufferSize`
Filename	`…FileName`	`DataFileName`
Filename (including the path)	`…FileNameAndPath`	`DataFileNameAndPath`

2.3.2 Standard Function Names

There function names that occur over and over again for different projects. Use the following standards for those names:

Description	**Name**	**Example**

Setting/resetting/getting a Boolean state	`set…On`
	`set…Off`
	`is…On`
		`setDataLogginOn();`
		`setDataLoggingOff();`
		`isDataLoggingOn();`

Setting/getting a value	`set…`
	`get…`
		`setSize(int x, int y);`
		`getWidth();`
		`getHeight();`

Callback	`on…`	`onButtonClick();`

2.3.3 Typenames

(neither data nor functions - Class, struct, union, typedef, enum, templates)

First letter is capital, each letter of a new word is capital also:

```
class TemperatureType
```

2.3.4 Variable Names

Each letter of a new word is capital, but not the first letter:

```
double myTemperatureInDegrees;
```

2.3.5 Data Member Prefix in Classes

Member Type	Prefix	Example
data member	`d_`	`d_myTemperature`
static data member	`s_`	`s_myTemperature`
constant data member	`c_`	`c_myTemperature`
static constant data member	`sc_`	`sc_myTemperature`
enum	`e_`	`e_myTemperature`

There should be no identifiers that use all capitals. This will avoid conflicts with preprocessor #defines. TRY TO AVOID THE USE of #defines, use constant variables and `enums` instead (see Programming Guide).

2.4 Variable Declaration/Definition

Variables are declared one per line, with an inline comment describing the variable (if necessary).

```
int    motorSpeed;      // Speed of the primary motor
double motorVelocity;   // Velocity of the primary motor
```

Notice that both inline comments could (and should in this case) be eliminated by using more descriptive variable names…

```
int    primaryMotorSpeed;
double primaryMotorVelocity;
```

There is another advantage to using more descriptive names – the inline comments appear only where you declared the variables. Where the variable is used in the code, you will not see the comment, only the variable name, so a descriptive name helps there.

2.5 Spaces

Put spaces around operators, and after parameters in functions

instead of:

```
a=1;
a+=1;
if(a==1)
for (i=1;i<5;i++)
copyFile(“test.dat”,filename,1000);
void copyFile(char *source,char *target,int len);
```

do this:

```
a = 1;
a += 1;
if (a == 1)
for (i = 1; i < 5; i++)
copyFile(“test.dat”, filename, 1000);
void copyFile(char *source, char *target, int len);
```

2.6 Blank Lines

Use blank lines to make your code more readable

Don’t have all lines of code sticking together. Use blank lines to group logical blocks of your declaration and inside functions.

2.7 Comments

Comment the code as much as possible!

2.7.1 Function Headers

A header is preceding every function:

```
//=========================================================
// MyClass::myMethod: Short Description of myMethod
// ---------------------------------------------------------
// input  : firstParameter - The label on the x-axis
//          secondParameter - Something else
// output : destination – Where to store the result
// return : -1 error, 0 success
//
// Sets the label that appears at the x-axis. If the
// string is set to zero or “”, no label will appear
//=========================================================
```

Note: A function definition does not need to be fully documented, if it is already documented in the headerfile. The header doesn’t have to include all entries, but should at least contain the === and the function name.

Do NOT use tabs; use only spaces in function header comments.

The simplest way to create function header comments is to copy an existing header block and modify it.

2.7.2 Classes

Use a class header that contains a description:

```
//=========================================================
// class Nose
// ---------------------------------------------------------
// \brief
// This class is a sophisticated Nose management class.
//
// The class Nose provides all generic functionality for
// Nose treatment. For your specific Nose, derive a class
// from Nose and specify the Nose geometry in your derived
// class, overriding getNoseGeometry()
//
// <b>Example Program</b>
//
// \include example/Nose.t.cpp
//=========================================================
```

doxygen notes:

The description in this header will be converted by our doxygen preprocessor to:

`/*! \class Nose … description */`

to include the class description in the manual.

- Use the \brief command to specify a brief description, which is important for the class list. Note: You must have a blank line between the brief description and the detailed description.

- Use the \include command to include the source code of the example

Comment data members and methods, indenting one level under the member or method being commented, in the .hpp file.

```
class Nose
{
 public:
  Nose ();
  ~Nose ();

  int pickNose (int whichHand, double howMuchForce);
    // This function allows one to pick one's nose. Call this
    // function only when alone or in the car.
    //  whichHand      Number of hand used to pick nose.
    //  howMuchForce   Force exerted on nose in newtons. Be careful with this
    //                 parameter. Too much force can damage the nose!
    //  return         -1 for error, 0 success.
};
```

doxygen notes:

The comment below functions and data members will be converted by our doxygen preprocessor to:

`/*!< comment */`

to be included in the manual.

- Use exactly two spaces after // to indicate parameters (the preprocessor will add \param)

- Use more then two spaces after // for continued parameter descriptions

- The preprocessor replaces "return" by "\return"

2.7.3 Logic Blocks within a Function

Separate logic blocks with blank lines and precede each block with a comment that explains what the block does.

```
int readNumberFromFile()
{
   // Open the file (this is a useless comment, I know what fopen
   // does)

   // Open the list of variables that will be logged (this is a
   // meaningful comment because it tells me what the program is
   // really doing).
   fp = fopen(“test.txt”, “r”);
   if (fp == NULL)
      return –1;

   // Read the data all at once into a val structure
   int val;
   if (fread(&val, sizeof(val), 1, fp) != 1)
   {
      fclose(fp)
      return –1
   };

   // Close the file (again, unnecessary comment)
   fclose(fp);

   return val;
}
```

2.8 Include Guard Convention

Include guards prevent a header file from being included multiple times (or in a cyclic fashion).

```
// For C include files use .h suffix.
#ifndef INCLUDED_MyFile_h
#define INLCUDED_MyFile_h

// Include file body here

#endif


// For C++ include files, use .hpp file suffix.
#ifndef INCLUDED_MyClass_hpp
#define INLCUDED_MyClass_hpp

// Include file body here

#endif
```

3. Programming Style Guides

3.1 Which Types to Use

Where possible, use only `int` and `double` variables.

In the interface methods of your classes, use only `int` and `doubles`. The space saving or extra precision achieved by using other types is not worth the possible problems caused by type conversion.

You can assume that `int` is at least 32 bit long.

3.2 #defines

TRY TO AVOID THE USE of #defines, use constant variables and enums instead.

The reason for this is:

a) Type checking of the compiler is used (it is not if you use `#define`)

b) The namespace is not polluted

`#defines` are sometimes ok in the cpp file (a file that is not included by other files), but still should be avoided.

[Example] Instead of:

```
#define NUM_POINTS  100
#define POINT_SIZE  4
#define PI  3.14159

class Stars
{
   int d_x[NUM_POINTS];
   int d_y[NUM_POINTS];
};
```

do this:

```
class Stars
{
   Stars ();  // Constructor
   enum
   {
      e_numPoints = 100,
      e_pointSize = 4
   };

   const double d_pi;

   int d_x[e_numPoints];
   int d_y[e_numPoints];
};

Stars::Stars() : d_pi = 3.14159
{
   …
}
```

4. File Organization Style Guides

4.1 Files

Each class *MyClass* should be implemented in two files: *MyClass.hpp* and *MyClass.cpp*

`MyClass.hpp` is the header file, also called "interface"

`MyClass.cpp` is the implementation

As a general rule, the headerfile only contains the minimum, which is: class definition and function declarations. All the actual code goes into the `MyClass.cpp` file.

Exceptions (having code in the headerfile, also called *inline*):

1. If the implementation of a function is only one line long, you can put it in the headerfile for simplicity
2. For time critical functions
3. For template functions/classes

4.2 Examples

Each class *MyClass* should have a example program that tests the class and demonstrates how to use it. Call it *MyClass.t.cpp* and put it in the *examples* directory

4.3 Typical Directory Structure

The typical directory structure of a one-class project is (example of a Complex class):

```
Complex
|
|-- Complex.hpp
|-- Complex.cpp
|-- makefile
|-- examples
    |
    |- Complex.t.cpp
    |- makefile
```

5. Doxygen

5.1 How to use doxygen

First step is to download doxygen (http://www.stack.nl/~dimitri/doxygen). Then use the –g option to generate a configuration file, then edit this configuration file and run doxygen with its name as the parameter.

5.2 Useful doxygen commands

`\a <word>`	Highlights <word>, for example to highlight a parameter in a function description
`\c <word>`	Displays <word> in code-style
`\code … \endcode`	Displays <…> in code-style
`\htmlonly … \endhtmlonly`	Uses <…> directly as html code
`\image html <name>`	Includes the image name into the html code
`\image latex <name.eps>`	Includes the eps image into the latex code
`\include`	Includes a file

6. Example Class

6.1 Source Code

This class demonstrates the coding convention. If you have a question about how to do something, do it like this class does (in terms of indention, variable naming, commenting, etc.)

```
//=============================================================================
// Project: QRTS Utility Libraries
// ---------------------------------------------------------------------------
// Package: Client/Server
// Authors: Markus Loffler
// Start Date: Wed Oct 04 14:17:52 utc 2000
// Compiler: Watcom/GNU
// Operating System: QNX4/Neutrino
// ---------------------------------------------------------------------------
// File: Client.hpp
// Headerfile for the class Client.
//=============================================================================


#ifndef INCLUDED_Client_hpp
#define INCLUDED_Client_hpp

#include "ClientServer.hpp"

#ifdef NTO
   #include <sys/neutrino.h>
#else
   #include <sys/kernel.h>
#endif

#include <errno.h>
#include <string.h>


//=============================================================================
// class Client
// ---------------------------------------------------------------------------
// \brief
// The class Client connects to a server and sends messages to the server
//
// The class Client is responsible for the client side of a client/server
// relationship. In a client server relationship, a program is designated
// the server. The server registers itself under a unique server name and
// waits for messages. The client finds the server by its name, connects
// to it, and sends messages to the server. The server receives messages,
// processes them and sends a reply back to the client.
//
// The class Client provides a platform independent wrapper for client
// functionality. It provides functionality to find the server by its
// name, and send messages in various formats to the server.
//
// <b>Example Programs:</b>
//
// \include example/Client.t.cpp
//=============================================================================

class Client : public ClientServer
{
 public:
   // ----- Constructors and Destructor -----

   Client ();
      // Creates the Client object, but does not connect to the server.
      // You must use the connectToServer() function before using any of
      // the sendMessage() functions.

   Client (char *serverName, char *serverFilename = 0,
           double startServerTimeout = 3);
      // Creates the Client object and connects to a server. See
      // connectToServer() for a detailed description of the connection process.
```

```
   virtual ~Client();
      // Disconnects from the server.


   // ----- Manipulators -----
   int connectToServer (char *serverName, char *serverFilename = 0,
                        double startServerTimeout = 3);
      // Tries to connect to the server. If the server can't be found,
      // this function tries to start the specified program
      // and tries to connect again.
      //  serverName          The name under which the server is registered
      //  serverFileName      The filename of the server program
      //  startServerTimeout  If the server couldn't be located after this
      //                      timeout (in seconds), the status is set to error
      //                      and the function returns

   // --- The following functions send a message to a process

   void sendMessage (void *msg, void *reply, int msgSize, int replySize);
      // Sends a message to the server and receives a reply. This function
      // will block until the server sends the reply.
      //  msg        Start address of the message
      //  reply      Start address of the reply buffer
      //  msgSize    Size of the message (in bytes)
      //  replySize  Size of the reply buffer (in bytes)

   int sendMessage (void *msg, int msgSize);
      // Sends a message to the server and receives an integer reply.
      // This function will block until the server sends the reply.
      //  msg        Start address of the message
      //  msgSize    Size of the message (in bytes)
      //  return     The integer reply received from the server

   void sendMessage (int msg, void *reply, int replySize);
      // Sends a message to the server that only consists of an integer value
      // and receives a reply.
      // This function will block until the server sends the reply.
      //  msg        Start address of the message
      //  reply      Start address of the reply buffer
      //  replySize  Size of the reply buffer (in bytes)

   int sendMessage (int msg);
      // Sends a message to the server that only consists of an integer
      // value and receives an integer reply
      // This function will block until the server sends the reply.
      //  msg        The integer value to send to the server
      //  return     The integer reply received from the server

   void sendMessage (void *msg, int msgSize, Status &status);
      // Sends a message to the server and receives a status reply.
      // If the status is error, then the status message is added to \a status.
      // This function will block until the server sends the reply.
      //  msg        Start address of the message
      //  msgSize    Size of the message (in bytes)
      //  status     A status object. The received error message  will be
      //             added to this status object (only in case of error)

   void sendMessage (int msg, Status &status);
      // Sends a integer message to the server and receives a status reply.
      // If the status is error, then the status message is added to \a status.
      // This function will block until the server sends the reply.
      //  msg        The integer value to send to the server
      //  status     A status object. The received error message will be
      //             added to this status object (only in case of error)

   // ========== END OF INTERFACE ==========

   // ----- Protected Data Members -----
 protected:
   char *d_serverName;
      // The name of the server

   // --- An ID that identifies the server (used to send messages to)
   #ifdef NTO
      int d_connectionId;    // NTO: Connection ID
   #else
      pid_t d_theirPid;      // QNX4: Pid of server
   #endif
};
```

```
//=========================================================================
// Client::sendMessage
// -------------------------------------------------------
// Note: These functions is defined inline to increase speed
//=========================================================================

inline void Client::sendMessage(void *msg, void *reply, int msgSize, int replySize)
{
   #ifdef NTO
      if (MsgSend (d_connectionId, msg, msgSize, reply, replySize) == -1)
      {
         d_status.setStatusError()
            << "[Client::sendMessage()]  "
            << "Error sending message - " << strerror(errno) << endl;
      }
   #else
      if (Send (d_theirPid, msg, reply, msgSize, replySize) == -1)
      {
         d_status.setStatusError()
            << "[Client::sendMessage()]  "
            << "Error sending message - " << strerror(errno) << endl;
      }
   #endif
}

inline int Client::sendMessage(void *msg, int msgSize)
{
   int reply = 0;      // Reply of message

   sendMessage(msg, &reply, msgSize, sizeof(reply));
   return reply;
}

inline int Client::sendMessage(int msg)
{
   int reply = 0;      // Reply of message

   sendMessage(&msg, &reply, sizeof(msg), sizeof(reply));
   return reply;
}

inline void Client::sendMessage(int msg, Status &status)
{
   sendMessage(&msg, sizeof(msg), status);
}

inline void Client::sendMessage(int msg, void *reply, int replySize)
{
   sendMessage(&msg, reply, sizeof(msg), replySize);
}


#endif
```

6.2 Manual created by doxygen

This is the RTF output generated by doxygen from the example class.

The class Client connects to a server and sends messages to the server.

`#include <Client.hpp>`

6.2.1 Public Methods

- **Client** ()
- **Client** (char *serverName, char *serverFilename = 0, double startServerTimeout = 3)
- virtual **~Client** ()
- int **connectToServer** (char *serverName, char *serverFilename = 0, double startServerTimeout = 3)
- void **sendMessage** (void *msg, void *reply, int msgSize, int replySize)
- int **sendMessage** (void *msg, int msgSize)
- void **sendMessage** (int msg, void *reply, int replySize)
- int **sendMessage** (int msg)
- void **sendMessage** (void *msg, int msgSize, Status &status)
- void **sendMessage** (int msg, Status &status)

6.2.2 Detailed Description

The class Client connects to a server and sends messages to the server.

The class Client is responsible for the client side of a client/server relationship. In a client server relationship, a program is designated the server. The server registers itself under a unique server name and waits for messages. The client finds the server by its name, connects to it, and sends messages to the server. The server receives messages, processes them and sends a reply back to the client.

The class Client provides a platform independent wrapper for client functionality. It provides functionality to find the server by its name, and send messages in various formats to the server.

Example Programs:

```
//=============================================================================
// Project: QRTS Utility Libraries
// ----------------------------------------------------------------------------
// Package: Client/Server
// Authors: Markus Loffler
// ----------------------------------------------------------------------------
// File: Client.t.cpp
// Example for the Client class.
//=============================================================================
```

```
#include <iostream.h>
#include "Client.hpp"


int main()
{
  cout << "Client..." << endl;

  Client client("qrts/testserver", "Server.t");
  if (client.d_status.isStatusError())
  {
    cout << client.d_status.getMessageText() << endl;
    return -1;
  }

  client.sendMessage(10);
  char *text = "This is text as a message";
  cout << client.sendMessage(text, strlen(text)) << endl;

  Status status;
  client.sendMessage(1, status);
  cout << "MESSAGE 1" << endl;
  cout << "Message Text: " << status.getMessageText() << endl;
  cout << "Error Code:   " << status.getStatusErrorCode() << endl << endl;

  status.setStatusOk();
  client.sendMessage(2, status);
  cout << "MESSAGE 2" << endl;
  cout << "Message Text: " << status.getMessageText() << endl;
  cout << "Error Code:   " << status.getStatusErrorCode() << endl << endl;

  client.sendMessage(0);

  return 0;
}
```

Definition at line 38 of file Client.hpp.

6.2.3 Constructor & Destructor Documentation

`Client::Client ()`

Creates the Client object, but does not connect to the server. You must use the **connectToServer**() (*p.19*) function before using any of the **sendMessage**() (*p.19*) functions.

`Client::Client (char * serverName, char * serverFilename = 0, double startServerTimeout = 3)`

Creates the Client object and connects to a server. See **connectToServer**() (*p.19*) for a detailed description of the connection process.

`Client::~Client () [virtual]`

Disconnects from the server.

6.2.4 Member Function Documentation

```
int Client::connectToServer (char * serverName, char * serverFilename =
0, double startServerTimeout = 3)
```

Tries to connect to the server. If the server can't be found, this function tries to start the specified program and tries to connect again.

Parameters:

serverName The name under which the server is registered
serverFileName The filename of the server program
startServerTimeout If the server couldn't be located after this timeout (in seconds), the status is set to error and the function returns

```
void Client::sendMessage (void * msg, void * reply, int msgSize, int
replySize)
```

Sends a message to the server and receives a reply. This function will block until the server sends the reply.

Parameters:

msg Start address of the message
reply Start address of the reply buffer
msgSize Size of the message (in bytes)
replySize Size of the reply buffer (in bytes)

```
int Client::sendMessage (void * msg, int msgSize)
```

Sends a message to the server and receives an integer reply. This function will block until the server sends the reply.

Parameters:

msg Start address of the message
msgSize Size of the message (in bytes)

Returns:

The integer reply received from the server

```
void Client::sendMessage (int msg, void * reply, int replySize)
```

Sends a message to the server that only consists of an integer value and receives a reply. This function will block until the server sends the reply.

Parameters:

msg Start address of the message
reply Start address of the reply buffer
replySize Size of the reply buffer (in bytes)

```
int Client::sendMessage (int msg)
```

Sends a message to the server that only consists of an integer value and receives an integer reply This function will block until the server sends the reply.

Parameters:

msg The integer value to send to the server

Returns:

The integer reply received from the server

`void Client::sendMessage (void * msg, int msgSize, Status & status)`

Sends a message to the server and receives a status reply. If the status is error, then the status message is added to *status*. This function will block until the server sends the reply.

Parameters:

msg Start address of the message
msgSize Size of the message (in bytes)
status A status object. The received error message will be added to this status object (only in case of error)

`void Client::sendMessage (int msg, Status & status)`

Sends a integer message to the server and receives a status reply. If the status is error, then the status message is added to *status*. This function will block until the server sends the reply.

Parameters:

msg The integer value to send to the server
status A status object. The received error message will be added to this status object (only in case of error)

The documentation for this class was generated from the following file:

Client.hpp