

Software Development Guidelines for the Robotics Research Group

Version 2.0

by

**Chetan Kapoor
Salman Siddiqui**

April, 2003

Table of Contents

1	INTRODUCTION.....	3
2	PROGRAMMING GUIDELINES	3
2.1	FILES	3
2.2	SECTIONING	3
2.3	IDEMPOTENCY GUARANTEE	4
2.4	INCLUDE FILES	4
2.5	FUNCTIONAL MACROS	5
2.6	MANIFEST CONSTANTS	5
2.7	EXTERNAL REFERENCES	5
2.8	ENUMERATIONS	5
2.9	UNIONS AND SIMPLE STRUCTURES	6
2.10	CLASSES.....	6
2.11	STATIC FUNCTION DECLARATIONS	7
2.12	STATIC VARIABLE DEFINITIONS.....	7
2.13	FUNCTIONS	7
2.14	STYLISTIC ISSUES.....	7
2.15	COMMENT KEYWORDS.....	8
2.16	ABSOLUTE NO-NO'S	8
3	ONLINE REFERENCE DOCUMENTATION USING DOXYGEN.....	8
3.1	GETTING STARTED	8
3.2	CONFIGURATION FILES	9
3.3	CREATING CONFIGURATION FILES	9
3.4	FREQUENTLY USED DOCUMENTATION TAGS	10
3.5	HOW TO DOCUMENT FOR DOXYGEN	10
4	VISUAL STUDIO APPLICATION DEVELOPMENT	11
4.1	MACROS.....	11
4.2	CREATING AN OSCAR APPLICATION.....	12
	APPENDIX A: SOFTWARE DESIGN	16
	APPENDIX B: DOCUMENTATION GUIDELINES.....	20

1 Introduction

This document outlines the programming guidelines for software development at the Robotics Research Group (RRG). These guidelines are proposed to help standardize the code development, testing, and documentation process. Additionally, a software design and development philosophy for current and future software development at the Robotics Research Group is also suggested. First the programming guidelines are described, followed by documentation guidelines and steps required to develop OSCAR applications using Visual Studio. The Appendices give a synopsis of the design process and documentation examples.

2 Programming Guidelines

The programming guidelines outlined here falls into three general categories. These are:

1. Required - These are few, but very important.
2. Suggested - Generally a set of options. Your style should be one of the displayed styles or substantially similar to one of the displayed styles.
3. Should – Unless there is good reason to do otherwise, this is the way to do things.

2.1 Files

1. C source files are **required** to have the extension `.c`.
2. C – usable header files (which somebody else might use) are **required** to have the extension `.h`.
3. C++ only header files are **required** to have the extension `.hpp`.
4. C++ source files are **required** to have the extension `.cpp`.
5. C++ inline definitions are **required** to have the extension `.ipp`.
6. It is **required** that inline functions not be placed in class declaration.
7. There **should** be not more than one class definition per header file.
 - a) This guideline can be relaxed a bit in the case of very similar classes sharing a common parent.
 - b) Multiple instantiations of the same templated class being in the same header file is not only O.K, but it is generally preferred.

2.2 Sectioning

1. Each section **should** be separated by two blank lines and the name of the section should be placed in a section comment block like the following (editor macro will be made available):
 - a)
 - b) `//`
 - c) `//` Includes
 - d) `//`
2. Header – All C and C++ files are **required** to have the following header (a macro will be made available):

```

////////////////////////////////////
//
// Title       : Base.h
// Project     : OSCAR Version 2.0
// Created    :
// Author      : Chetan Kapoor
// Platforms   : All
// Copyright   : Copyright© The University of Texas at Austin, 2002. All rights reserved.
//
// This software and documentation constitute an unpublished work and contain
// valuable trade secrets and proprietary information belonging to the University. None
// of the foregoing material may be copied or disclosed without the express, written
// permission of University. THE UNIVERSITY EXPRESSLY DISCLAIMS ANY AND ALL
// WARRANTIES CONCERNING THIS SOFTWARE AND DOCUMENTATION,
// INCLUDING ANY WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
// PARTICULAR PURPOSE, AND WARRANTIES OF PERFORMANCE, AND ANY WARRANTY
// THAT MIGHT OTHERWISE ARISE FROM COURSE OF DEALING OR USAGE OF TRADE.
// NO WARRANTY IS EITHER EXPRESS OR IMPLIED WITH RESPECT TO THE USE OF
// THE SOFTWARE OR DOCUMENTATION. Under no circumstances shall the University
// be liable for incidental, special, indirect, direct or consequential damages or loss of
// profits, interruption of business, or related expenses which may arise from use of software
// or documentation, including but not limited to those resulting from defects in software
// and/or documentation, or loss or inaccuracy of data of any kind.
//
// Purpose     : Acts as the base class for all other OSCAR classes.
//
//-----
//
// $Revisions$
//
// $Log$
//
////////////////////////////////////

```

2.3 Idempotency Guarantee

In header files it is **required** that one place an idempotency guarantee immediately after the header. The idempotency format looks like the following:

```

#if !defined(xxxx)
#define xxxxx
.
.
.
#endif

```

Here xxxxx is the filename, substituting an underscore for the period. For example, for header file named “InverseKinematics.hpp”, xxxxx will be InverseKinematics_hpp.

2.4 Include Files

The include section is required to start with an include of the common header file, if one exists. For example, we might have a common header called “OSCAR.hpp” which every file might have to include.

1. After the common header file comes the required #pragma directives. The rest of the header files then follow.
2. Header files should be included using one of the following forms:
 - a) Standard headers - <header.hpp>, example <iostream.h>.
 - b) All other headers must be specified relative to the root source directory - <subdir\header.hpp>.
 - c) Headers are sorted by standard headers first, followed by all other headers. A single blank line separates the standard headers from the rest. It is also acceptable to separate functional groups of headers with single blank lines. Within each group, headers are sorted alphabetically.

2.5 Functional Macros

Next comes the functional macros section, if needed. Macros can tend to be buggy, and are discouraged.

1. In C++, prefer templated functions over macros when possible.
2. Macros in all styles are all caps with underscores separating words.

2.6 Manifest Constants

Manifest constants should be used instead of hard-coded values. For example it is better to do this:

```
const unsigned int size = 10;
for (int j=0 ; j<size ; j++)...
```

rather than:

```
for (int j=0 ; j<10 ; j++)
```

2.7 External References

1. Desirability of External References
 - a. Class references are fine.
 - b. In general, external function references are bad; they should instead be included by including the appropriate header.
 - c. External variable references are real bad, since globals should not be used anyway. When globals are used they should always be referenced via a header.
2. External references are sorted alphabetically.

2.8 Enumerations

1. Typedef'ing in C. Typedef'ing is needed to avoid having to reference an enumeration with the keyword enum. In C++, the enumeration's name is of the correct scope, and this is not a problem.
2. The enum and typedef names are first caps. Names of enumerated values are first caps.

Exampe in 'C':

```
typedef enum RobotJoints {
    Revolute,
    Prismatic,
} RobotJoints;
```

Same example in 'C++':

```
enum RobotJoints {
    Revolute,
    Prismatic,
};
```

2.9 Unions and Simple Structures

1. Typedef' ing in C. Typedef' ing is needed to avoid having to reference a union or structure with a keyword. In C++ the names are of the correct scope, and this is not a problem.
2. The struct/union and typedef names are first caps. Names of struct/union members are first lower rest caps.

Exampe in 'C':

```
typedef struct ComplexInt {
    int realPart;
    int imagPart;
} ComplexInt;
```

2.10 Classes

Any structure with member functions or privates or such should be declared as a class.

1. Layout of the Class
 - a) Access sections: The public section comes first followed by the protected section, followed by the private section. The access keyword (public, private, protected) is indented one indentation level, and everything until the next access keyword is indented one more indentation level.
 - b) Usage Sections: Within each access section, things are ordered as follows:
 - i) Scoped enumerations, structures, unions and classes.
 - ii) Data Members.
 - iii) Constructors and Destructors.
 - iv) Selectors (Get Methods).
 - v) Modifiers (Set Methods).
 - c) Undefined Canonicals (like the operator= and const Reference constructor) only in the private section.
2. All class names are **required** to begin with the letters RR. This is to avoid polluting the global namespace. E.g. RRBase.

3. The class name and public member function names are all first caps (first letter of each word is capital). E.g. RRBase, GetError().
4. All protected and private member functions names are all first lower caps, and the rest are all first caps. E.g. setError(), updateDimensions().
5. The public data members of a class are also all first caps. E.g. State.
6. The protected and private data members of a class are all first lower caps, and the rest are all first caps. E.g. setError(), updateSize().

2.11 Static Function Declarations

Static functions should be declared before defined. Static function names are all first caps.

2.12 Static Variable Definitions

Static variables are grouped by type, then sorted alphabetically. Static variables follow the same rules as data member variables.

2.13 Functions

Due to the importance of functions they need a separate section separator.

1. Section Separator – See Section 3.5.
2. Function names are all first caps.
3. Function parameters are all first lower caps, and the rest are caps. This facilitates

2.14 Stylistic Issues

1. Indentation
 - a) Use 2 or 4 spaces per indentation level.
2. Spacing
 - a) One space between function parameters is allowed but not required.
 - b) In general leave one space between compound expression, operators, braces, etc.
 - c) Common sense will help.
3. Braces
 - a) Braces for functions go in the same line or the first column of the next line.
 - b) Braces for constructors with initializers go in column 1 of the next line.
 - c) Braces for all other constructors go on the same line.
4. Comments
 - a) Since all compilers we plan on using the C++ style comments, they are fine in C files.
 - b) Comments start at the current indentation column and extend to column 78.
 - c) The comment block will be made available via a macro.
 - d) Sample comment block:


```
//.....
// This is a sample comment. Notice that if the text of the comment starts to
// the end of the line, I just started another line.
//.....
```

2.15 Comment Keywords

Comment keys allow quick grepping for items which may be of importance. A comment key should be in a comment with whitespace on the left of it and whitespace or a colon on the right of it. The sequences of characters in comment keys should not be used anywhere else in a program, since that will slow down the grepping process.

1. TODO – Indicates there is code needed to be added or an issue resolved.
2. PDI – Indicates there is a platform dependency.
3. RBG – Indicates that code is real bad garbage and either could use rewriting or is an unavoidable hack.

2.16 Absolute No-No's



Never start a name with an underscore followed by a capital letter and never use a double underscore anywhere in a name. These sequences are reserved by the compiler.

2. It is **required** that all names used in your source code be meaningful, nothing like 'foobar.'

3 Online Reference Documentation Using DOXYGEN

Doxygen is a documentation system for C++, C, Java, IDL (Corba, Microsoft, and KDE-DCOP flavors) and to some extent PHP and C#.

It can help you primarily in two ways:

1. It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in \LaTeX) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.
2. You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. You can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.

All the .h files in OSCAR are to be documented using doxygen **documentation commands**. Doxygen will then parse through the documentation and create the reference.

3.1 Getting Started

The executable *doxygen* is the main program that parses the sources and generates the documentation.

The executable *doxytag* is only needed if you want to generate references to external documentation (i.e. documentation that was generated by doxygen) for which you do not have the sources or to create a search index for the search engine.

The executable *doxysearch* is only needed if you want to use the search engine.

Optionally, the executable *doxywizard* is a GUI front-end for editing the configuration files that are used by doxygen.

For our purposes, we will be using **doxywizard** to create configuration files that are used to create the reference.

3.2 Configuration Files

Doxygen uses a configuration file to determine all of its settings. Each project should get its own configuration file. A project can consist of a single source file, but can also be an entire source tree that is recursively scanned.

When doxywizard is first opened, it loads a template configuration file which has all the tags set to their default values.

If the tag **EXTRACT_ALL** is set to yes, and an input source and the output location is specified, doxygen will pretend everything in your sources is documented (even if it is not) and create a `html`, `rtf`, `latex` and/or `man` directory inside the output directory. As the names suggest these directories contain the generated documentation in HTML, RTF, ~~WTeX~~ and Unix-Man page format.

The default output directory is the directory in which doxygen is started. The directory to which the output is written can be changed using the **OUTPUT_DIRECTORY**, **HTML_OUTPUT**, **RTF_OUTPUT**, **LATEX_OUTPUT**, and **MAN_OUTPUT** tags of the configuration file. If the output directory does not exist, doxygen will try to create it for you.

The generated HTML documentation can then be viewed by pointing a HTML browser to the `index.html` file in the `html` directory. For the best results a browser that supports cascading style sheets (CSS) should be used.

3.3 Creating Configuration Files

A configuration file is a free-form ASCII text file with a structure that is similar to that of a Makefile, default name `Doxyfile`. It is parsed by doxygen. The file essentially consists of a list of assignment statements. Each statement consists of a `TAG_NAME` written in capitals, followed by the = character and one or more values.

When you open up doxywizard, you will see a number of tabs at the top. Each tab deals with a specific area of creating the reference. So basically, you can fine-tune doxygen by specifying values for each tag so that doxygen generates the documentation you want. For instance, the **Input** tab deals with the specification of the files that are to be used for creating the reference. Therefore, doxygen will only parse the specified files and provide an output.

For further information about the different options available, see the Appendix.

3.4 Frequently Used Documentation Tags

- `\brief` – This is a brief description of the class/enum/function.
- `\author` – This is the name of the author.
- `\class` – This is the name of the class.
- `\param` – This is the argument taken by a function.
- `\return` – This is the return value of a function.
- `\exception` – This is an exception or error thrown by the function.
- `\sa` – See also; This allows the user to look at other similar functions as well.
- `\enum` – This is the name of the enumeration.

For further information, see the Appendix.

3.5 How to Document for DOXYGEN

All classes and member functions are preceded by a documentation block. This documentation block contains documentation commands and their corresponding values. For instance:

- Classes:

```
/**
\class RRBase
\author Chetan Kapoor
\brief This is the Base Class. (A brief description should be one line)

RRBase acts as the Base Class for OSCAR. (A more detailed description)
*/
```

- Functions:

```
/**
\brief Get the sum of two vectors. (A brief description. Should be one line)

This method is used to calculate the sum of two vectors of the same type,
and return the resulting vector
\param x X is the first Vector.
\param y Y is the second Vector.
\return The resulting vector.
\exception vectorTypeMismatch #vectorTypeMismatch
*/
```

- Enumerations:

```
/**
```

```
\enum RRCoordinateStatus
```

```
\brief Enumeration for Active/Inactive coordinate status.
```

Enumeration contains two boolean values. Active status is denoted as True. Inactive status is denoted as False.

```
*/
```

Always follow up a brief description by a blank line, and then a detailed description of the class or function.

For further information, see the Appendix.

4 Visual Studio Application Development

4.1 Macros

Macros are used to add documentation blocks to the source and header files in OSCAR. More specifically, they are used to add documentation blocks to a file for the file itself, and for the classes and functions declared in that file. For instance:

- All Files in OSCAR are required to have a documentation block at the very top that contains information such as the name of the file, author of the file, date it was created, copyright notice, purpose, etc.
- All Classes in OSCAR are required to have a documentation block preceding the declaration of the class that contains information such as the name of the class, author of the class, and description of the class.
- All Functions in OSCAR are required to have a documentation block preceding the declaration of the class that contains information such as the name of the function, parameters of the function, return value of the function, and exceptions.

OSCAR Macros are used to add the documentation blocks automatically. Once the documentation blocks have been loaded, the user must fill out the blocks with the appropriate information. The OSCAR Macros are available in the file **OSCAR MACROS.DSM**. This file is located in the directory **O:\Utilities**.

Following Are the Steps on How to Use these Macros:

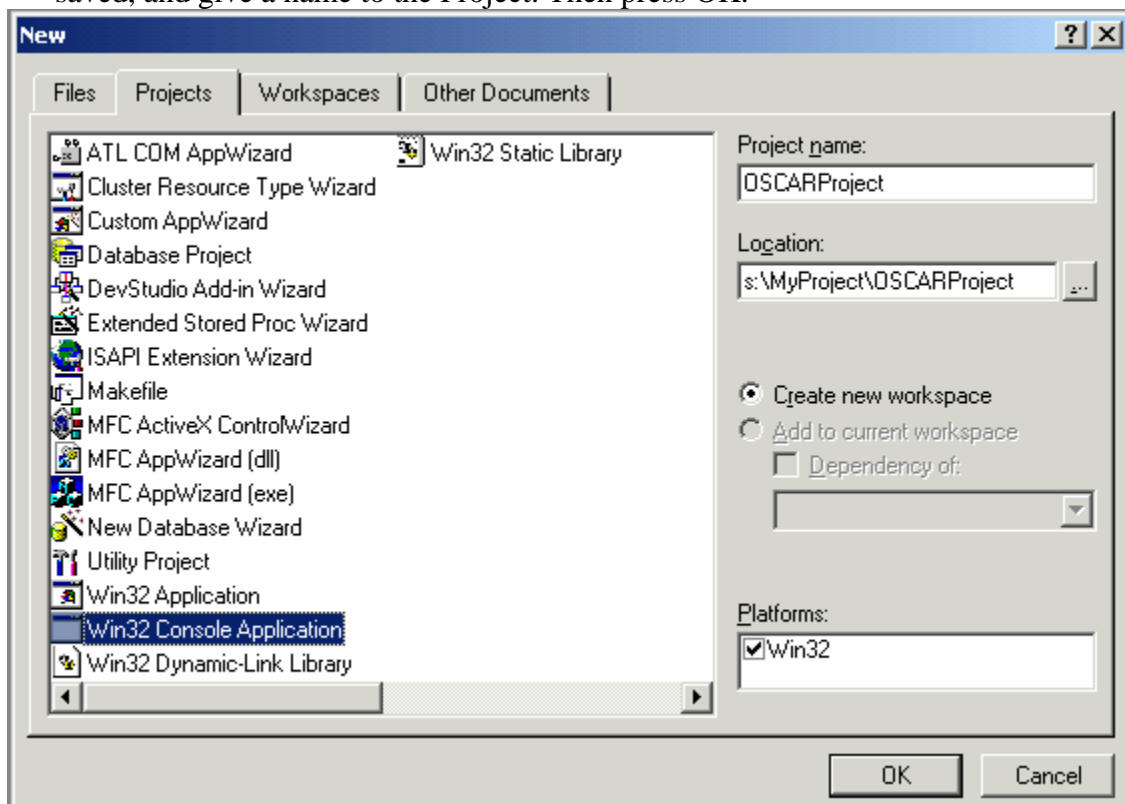
1. Open an **OSCAR header or source file** in Microsoft Visual C++.
2. Go To Tools>Macro. (A window will open showing a listing of Macros/Options)
3. Select the Macro that you want to run from the listing of Macros on the left.
4. Click on **run**.

Now go ahead and fill out the documentation block with the appropriate information.

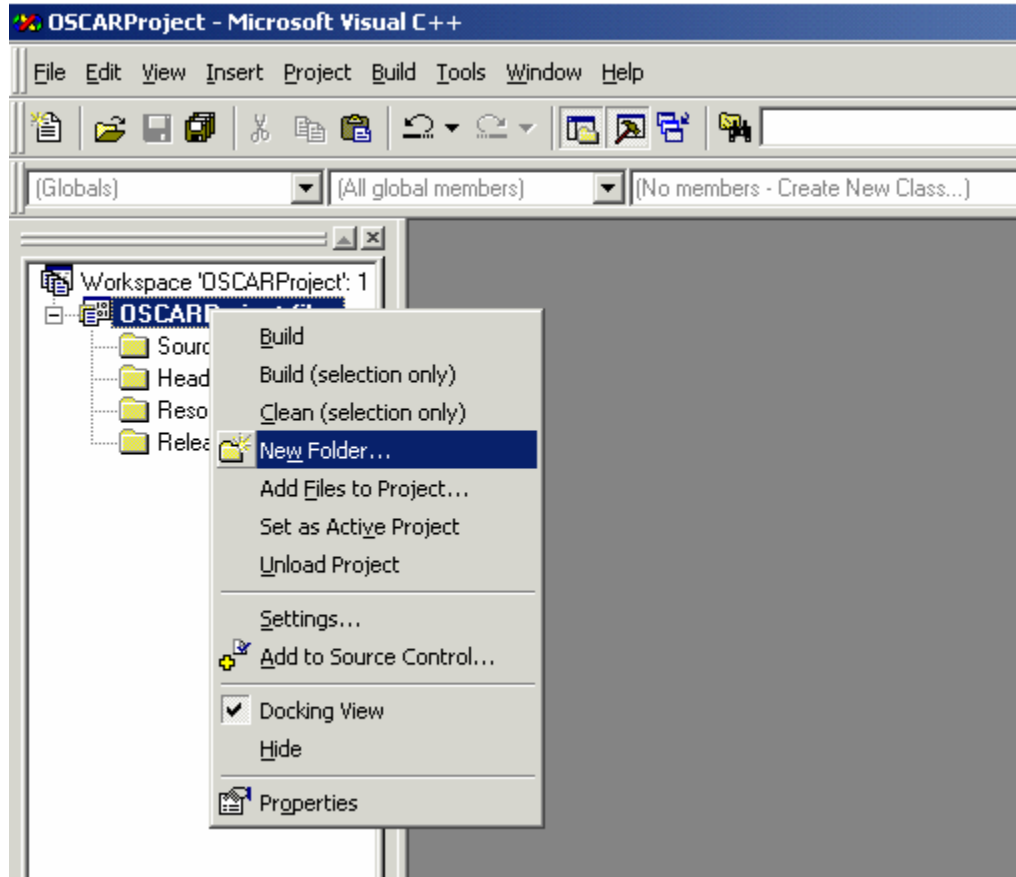
4.2 Creating an OSCAR Application

Following Are the Steps to Create an OSCAR Application:

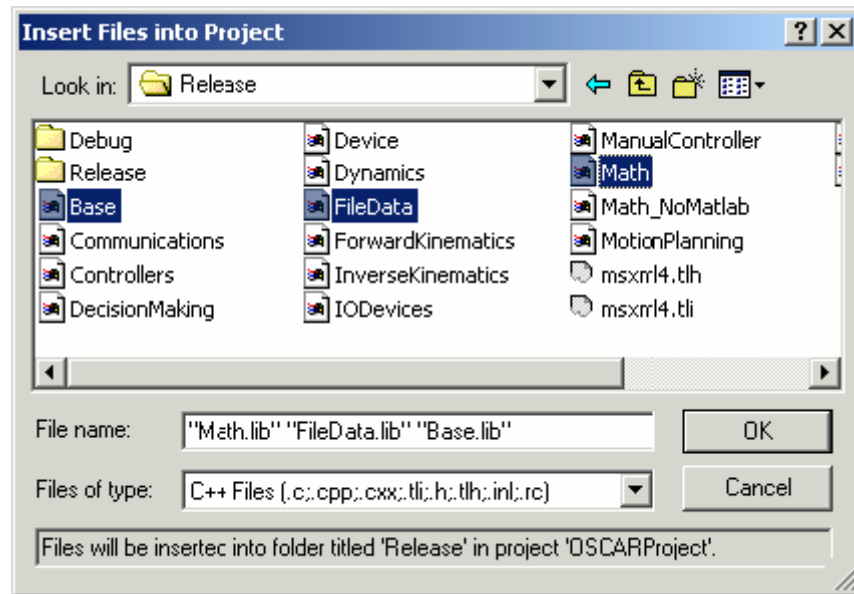
1. First, use SmartCVS to download the library project files from the CVS repository to your local machine and rebuild them. Then, use the .bat file found on the server in z:\ to copy the necessary files from the project folders into c:\oscar\includes and c:\oscar\libraries. The rest of these instructions will assume that all header files are located in c:\oscar\includes\[library_name] and that all libraries are location in c:\oscar\libraries\debug and c:\oscar\libraries\release.
2. Open Microsoft Visual C++.
3. Go To File>New.
4. Select **Win32 Console Application**. Specify the Location for the Project to be saved, and give a name to the Project. Then press OK.



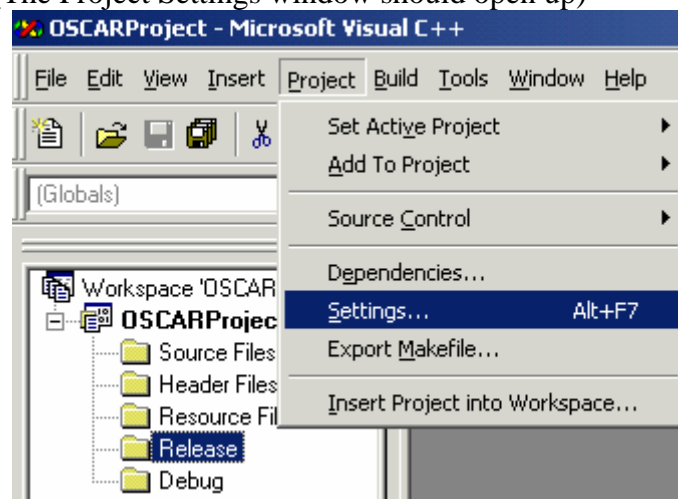
5. **Create an empty project.** Press OK. (An empty Workspace is created)
6. Now click on the **File View** Tab located at the **bottom left hand corner** of the MS Visual screen. (The directory structure and file listing becomes visible in the workspace window)
7. Now we need to **create two folders: Debug and Release**. Right click on the name of the Project in the workspace window, and **add** the two **new folders**.



8. Now we need to insert the appropriate OSCAR library file to be used for your application into the Debug Folder.
 - **Right-click** on the **Debug** folder,
 - Select **Add Files To Project**,
 - Traverse to the directory: **c:\oscar\libraries\debug**
 - **Select** the appropriate OSCAR Library files for your application.
 - Press OK.
9. Now we need to insert the appropriate OSCAR library file to be used for your application into the Release Folder.
 - **Right-click** on the **Release** folder,
 - Select **Add Files To Project**,
 - Traverse to the directory: **c:\oscar\libraries\release**
 - **Select** the appropriate OSCAR Library files for your application.
 - Press OK.



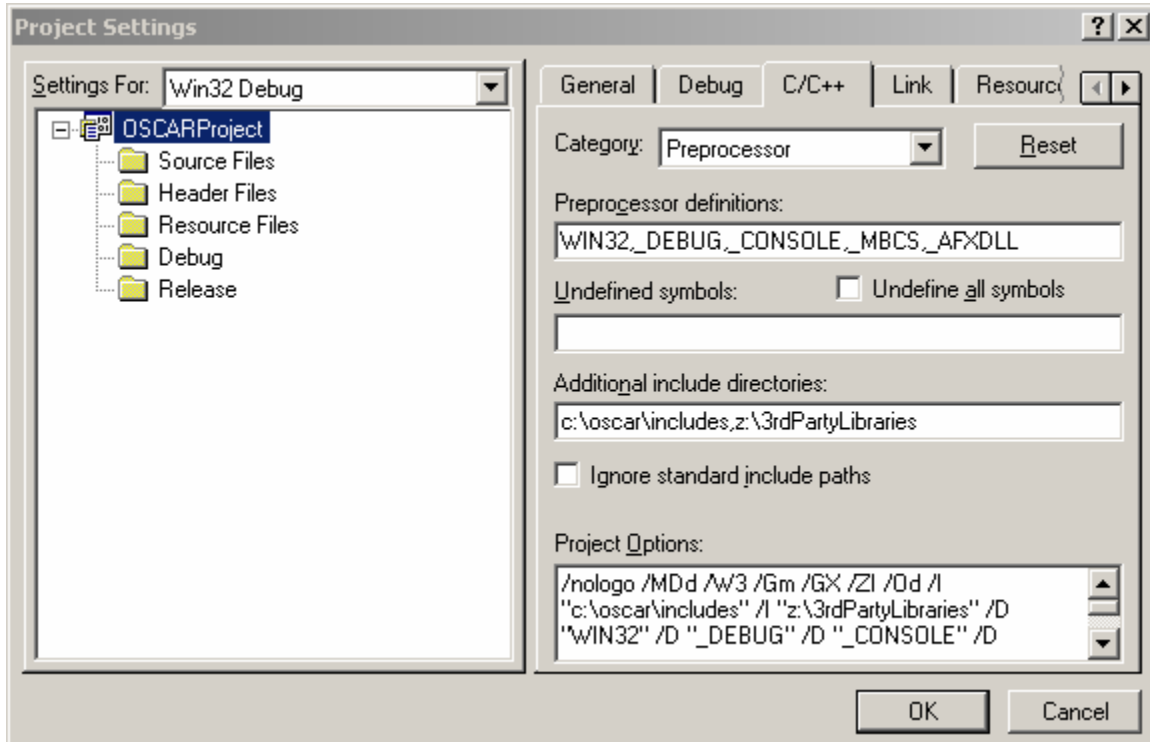
10. Now Right-click on the Project name in the workspace window, and Select **settings**. (The Project Settings window should open up)



10. Select **All Configurations** from the Settings For Menu.

- Click on the **General** tab.
 1. Select **Use MFC in a Shared DLL** from the Microsoft Foundation Classes Menu.
- Click on the **C/C++** tab.
 1. Select **Code Generation** from the category menu.
 - Select **Pentium** from the Processor menu.
 2. Select **Listing Files** from the category menu.
 - Check the **generate browse info** checkbox.
 3. Select **Pre processor** from the category menu.
 - In the **Additional include directories** text-box, specify the path **c:\oscar\includes,z:\3rdPartyLibraries** (your z:\ should be mapped to [\\mrburns\oscar](http://mrburns.oscar))
- Click on the **Browse Info** tab.

1. Check the **Build browse info file** checkbox

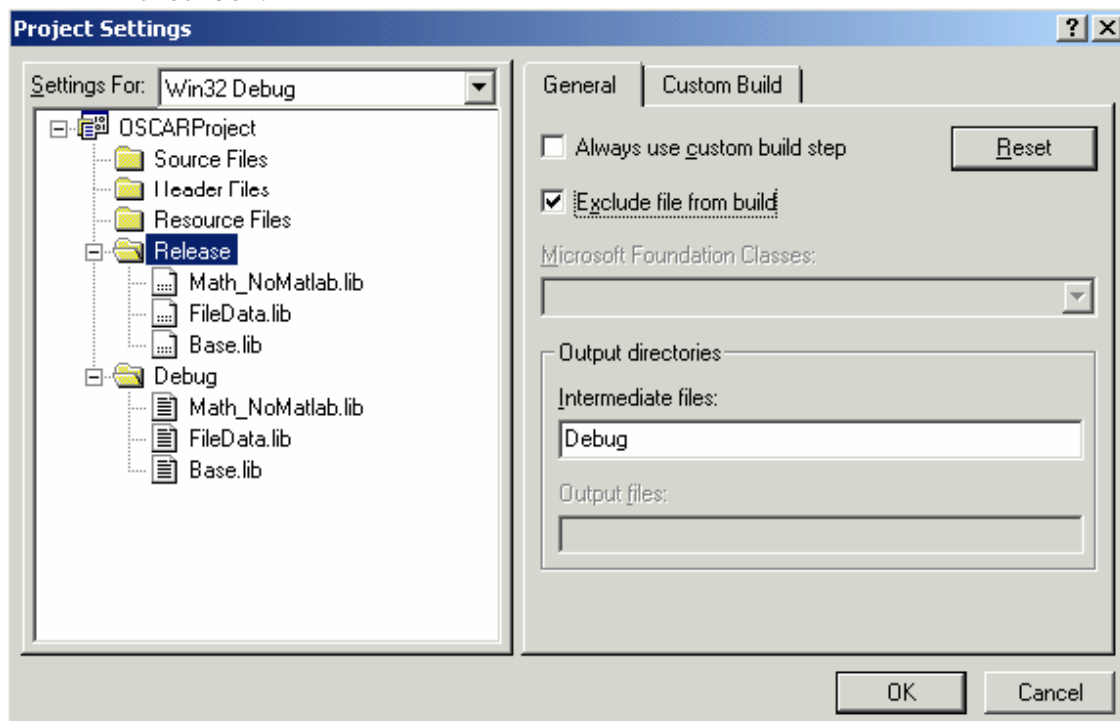


11. Select **All Win32 Debug** from the Settings For Menu.

1. **Highlight the Release folder**, and Check the **Exclude file from build** checkbox.

12. Select **All Win32 Release** from the Settings For Menu.

2. **Highlight the Debug folder**, and Check the **Exclude file from build** checkbox.



13. Press OK.
14. Now, Go to **Tools**, Select **Options**.
 - Click on the **Tabs** tab.
 1. In the **Tab size** text-box, put down 2.
 2. In the **Indent size** text-box, put down 2.
 3. Check the **Insert spaces** checkbox.
 4. Press OK.
15. Now, Go to File>New.
16. Select **C/C++ source file**, or **C/C++ Header File**, and give it a name.
17. Press OK.

Now you are all ready to go!

Appendix A: Software Design

1. Software Design Philosophy

All mature industries, like automobiles, are characterized by the presence of a components industry. For example, an automobile is made up of components which are supplied from different vendors. Currently there is a revolution taking place in the commercial software industry. This is brought on by the popularity of object-oriented design. A similar revolution is desired in the intelligent machines software industry. This can be achieved by the stressing the development of software modules (henceforth called components) which have the following characteristics:

- Standardized Interfaces,
- Completeness and abstractness (should have a black box appearance),
- Reusability, and
- Portability.

For a long time, software development was directed towards fulfilling project goals. This often led to the development of shoddy software which had little reuse potential. This made future releases harder and more expensive. Emerging software design philosophies stress the development of robust components which can be modified and assembled to achieve project goals. Keeping this paradigm shift in mind, the software design philosophy of the Robotics Research Group can be summed up in one line:

“From Projects to Components.”

2. How do we achieve our design goals?

One means of achieving our design goal of building reusable components is to develop software using object-oriented programming. Object-oriented programming has four key components. These are:

- Information hiding,
- Data abstraction,
- Dynamic binding, and
- Inheritance.

Classes, a key component of object-oriented programming (in C++) provide all the information necessary to construct and use objects of a particular type. A class here is analogous to a template from which instances (objects) can be created. Each instance has a state associated with it.

Information hiding is achieved by hiding the inner working details of a class. This helps in avoiding inter-dependencies between components and also makes modification easier.

Date-abstraction is used to abstract the functionality of a class and give it a black-box look and feel.

Dynamic binding is used for dynamic polymorphism and to avoid use of *switch* statements to determine the type of an object (i.e. which class it was instantiated from).

Inheritance provides us a means of sub-classing. It allows for creation of new classes by specifying the differences between the new class and an existing class. This promotes reusability.

Another issue to consider in the design and the specification of components is the types of reuse we desire from our components. Components can offer reuse in two forms: generality and extensibility. An example of generality is the first generation of the Mac's. These Mac's were general enough to run different software but were not extensible. That is, you could not make hardware modifications like adding RAM. **Our goal is to exploit reuse via generality and extensibility.** To achieve this kind of reuse, a solid set of methods and techniques is desired.

3. The Design Process

This section gives a brief outline of traditional engineering software design and object-oriented software design as specified by Booch. This information is compiled from books on engineering design and software design. An interesting thing to note is that none of the software design books reference the work done in the area of engineering design in the earlier part of this century. I think that is a mistake, as these books contain a wealth of design information which is applicable to all systems.

3.1. Traditional Engineering Design

3.1.1. What is Design?

As defined by Paul and Beitz, *Designing is an intellectual attempt to meet demands in the best possible way.* A systematic design alone can produce rational solutions.

3.1.2. What are the qualities of a good designer?

- Initiative,
- Resolution,
- Economic insight,
- Tenacity,
- Optimism,

- Sociability, and
- Teamwork.

3.1.3. Summary of the Design Process

1. Task clarification,
2. Conceptual Design,
3. Embodiment Design, and
 - a) Layout design
 - b) Form design
4. Detail design.

All throughout the design process one should have:

- ⇒ the required motivation,
- ⇒ clarify the boundary conditions,
- ⇒ dispel prejudice,
- ⇒ look for variants,
- ⇒ make decisions, and
- ⇒ balance between an intuitive approach and a systematic approach.

3.1.4. Critical Path Analysis

A *critical path analysis* allows designers to manage time for

- ⇒ feasibility study,
- ⇒ search for solutions, and
- ⇒ evaluation of results,

3.2. Software Design Process (Booch)

There are three steps in object-oriented software development as specified by Booch. These are outlined in the following section:

3.2.1. Requirement Analysis

The **objectives** of requirement analysis are:

- To find out what customers want the system to do, and
- to form a contract between the customer and the developer.

The **deliverables** of the requirement analysis are:

- System Charter: This outlines the responsibilities of the system.
- System Function Statement: This outlines the key cases of the system.

3.2.2. Domain Analysis

The **objectives** of domain analysis are:

- Add appropriate level of abstraction to a system.
- Identify all major objects in the domain including all data and major operations.

The **deliverables** of domain analysis are:

- Class diagrams which identify the key classes, or types, of the domain.

- Class specification, which contain all semantic definitions of the classes, their relationships, their attributes, and their key operations.
- Object-scenario diagrams, which illustrate how the objects will interact to carry out key system functions.
- Data dictionary, which lists all domain entities including classes, relationships, and attributes.

The **steps** involved in domain analysis are:

- define Classes,
- define relationships,
- define operations,
- find attributes (properties that describe that class),
- define inheritance, and
- validate and iterate.

3.2.3. *System Design*

The **objectives** of system design are:

- Determine effective, efficient, and cost-effective implementation to carry out the function.
- To map ideal or logical analysis developed during domain analysis to objects and classes which can be coded.

The **deliverables** of system design are:

- Architectural description, i.e. hardware, CPU, OS, language, etc.
- Executable release descriptions.
- Class-category diagrams, which model the partitioning of the system into high-level groups.
- Design class diagrams, which show detail data types and structures, etc.
- Design object-scenario diagrams, which show detail operational logic.
- New specifications supporting these diagrams.
- Amended class specifications showing full operational specifications for those operations with complex algorithms (very valid in robotic software).

The **steps** involved in system design are:

- determine initial architecture,
- plan executable releases,
- develop executable releases,
- refine the design.

This section gave a brief overview of the iterative process of object-oriented analysis and design. As a class is a key component of object-oriented software, meaningful names should be used for them. Ill defined names simply imply ill-defined abstractions. Also, a good class has loose or weak coupling and does not have a lot of relationships to other classes outside of its super-types or subtypes.

Appendix B: Documentation Guidelines

Following are the Configuration File Settings for preparing the Configuration File to be used by Doxygen to generate the online reference for OSCAR. All these options have to be selected in order to generate and maintain a consistently documented online reference.

General configuration options

```
PROJECT_NAME = "OSCAR VERSION 2.0 ONLINE REFERENCE"  
PROJECT_NUMBER = 2.0  
EXTRACT_ALL  
EXTRACT_STATIC  
EXTRACT_LOCAL_CLASSES  
HIDE_IN_BODY_DOCS  
BRIEF_MEMBER_DESC  
CASE_SENSE_NAMES  
VERBATIM_HEADERS  
SHOW_INCLUDE_FILES  
JAVADOC_AUTOBRIEF  
INLINE_INFO  
TAB_SIZE = 5  
MAX_INITIALIZER_LINES = 30  
OPTIMIZE_OUTPUT_FOR_C  
SHOW_USED_FILES
```

Configuration options related to warning and progress messages

```
WARNINGS  
WARN_IF_UNDOCUMENTED  
WARN_IF_DOC_ERROR  
WARN_FORMAT = "$file:$line: $text"  
WARN_LOGFILE = warnings
```

Configuration options related to the input files

```
INPUT = "o:/Version 2.0/Source/FileData/" \  
        "o:/Version 2.0/Source/Base/" \  
        "o:/Version 2.0/Source/Math/" \  
        "o:/Version 2.0/Source/Communications/" \  
        "o:/Version 2.0/Source/Control/" \  
        "o:/Version 2.0/Source/Controllers/" \  
        "o:/Version 2.0/Source/DecisionMaking/" \  
        "o:/Version 2.0/Source/Device/" \  
        "o:/Version 2.0/Source/Dynamics/" \  
        "o:/Version 2.0/Source/ForwardKinematics/" \  
        "o:/Version 2.0/Source/InfoBasedCriteria/"
```

```
"o:/Version 2.0/Source/InverseKinematics/" \  
"o:/Version 2.0/Source/IODEVICES/" \  
"o:/Version 2.0/Source/ManualController/" \  
"o:/Version 2.0/Source/MotionPlanning/" \  
"o:/Version 2.0/Source/ObstacleAvoidance/"  
FILE_PATTERNS = *.h  
RECURSIVE = YES  
EXCLUDE = "o:/Version 2.0/Source/Math/LSGRG/" \  
          "o:/Version 2.0/Source/Math/Debug/"  
EXCLUDE_PATTERNS = *.C \*.ipp
```

Configuration options related to the alphabetical class index

```
ALPHABETICAL_INDEX = YES  
COLS_IN_ALPHA_INDEX = 1
```

Configuration options related to the HTML output

```
GENERATE_HTML  
HTML_FILE_EXTENSION = .html  
HTML_ALIGN_MEMBERS  
CHM_FILE = "OSCARv2.0 Help"  
ENUM_VALUES_PER_LINE = 4  
GENERATE_TREEVIEW  
TREEVIEW_WIDTH = 280
```

Configuration options related to the preprocessor

```
ENABLE_PREPROCESSING  
SEARCH_INCLUDES  
SKIP_FUNCTION_MACROS
```

Configuration options related to the dot tool

```
CLASS_DIAGRAMS  
HIDE_UNDOC_RELATIONS  
HAVE_DOT  
CLASS_GRAPH  
COLLABORATION_GRAPH  
TEMPLATE_RELATIONS  
INCLUDE_GRAPH  
INCLUDED_BY_GRAPH  
GRAPHICAL_HIERARCHY  
DOT_IMAGE_FORMAT = gif  
DOT_PATH = "c:/Program Files/ATT/Graphviz/bin/"  
MAX_DOT_GRAPH_WIDTH = 1024  
MAX_DOT_GRAPH_HEIGHT = 1024
```

GENERATE_LEGEND
DOT_CLEANUP

Additional Required Software:

Graphviz is used by Doxygen to generate Graphical representations of existing relationships between all objects. Graphviz contains the **dot** tool and this allows us to represent all object properties and relationships in a graphical manner.

Installation of Graphviz:

Graphviz is an open source software. It can be downloaded from the web from the following address:

<http://www.research.att.com/sw/tools/graphviz/download.html>

Install Graphviz in a directory of its own.

Then while configuring the Doxygen Configuration File for the online reference, specify the absolute path to the bin subdirectory in the Graphviz directory, in the section for **configuration options related to the dot tool**. This is necessary since Doxygen needs to know where this tool is in order to generate graphical representations.

For further information on Configuration options, see:

<http://www.stack.nl/~dimitri/doxygen/config.html>