**Coding Guideline for the Driver Development team**

Table of Contents

[1.0 Introduction 5](#_Toc250663154)

[1.1 Layout 5](#_Toc250663155)

[1.2 Guideline Indicator 5](#_Toc250663156)

[2.0 Document Maintenance 5](#_Toc250663157)

[3.0 2.1 Revision History 6](#_Toc250663158)

[4.0 Design Guidelines 6](#_Toc250663159)

[4.1 Object-oriented 6](#_Toc250663160)

[4.2 Encapsulation 6](#_Toc250663161)

[4.3 Robustness 7](#_Toc250663162)

[4.4 Portability 7](#_Toc250663163)

[4.4.1 OS Independence 7](#_Toc250663164)

[4.4.2 Architecture Independence 7](#_Toc250663165)

[5.0 Standards 8](#_Toc250663166)

[5.1 Compiler warnings 8](#_Toc250663167)

[5.2 Virtual Destructors 8](#_Toc250663168)

[5.3 Avoid Global Constructors 8](#_Toc250663169)

[6.0 Readability and Portability Guidelines 8](#_Toc250663170)

[6.1 Line Formatting 9](#_Toc250663171)

[6.1.1 Line Lengths 9](#_Toc250663172)

[6.1.2 White Space Use 9](#_Toc250663173)

[6.1.3 Tab Character Use Disallowed 9](#_Toc250663174)

[6.1.4 Nested Operators 9](#_Toc250663175)

[6.2 Block Formatting 9](#_Toc250663176)

[6.2.1 Block Format Type 9](#_Toc250663177)

[6.2.2 Level Indent Size 9](#_Toc250663178)

[6.2.3 Examples of Block Formatting 9](#_Toc250663179)

[6.2.4 Number of Nested Blocks 11](#_Toc250663180)

[6.3 Naming Conventions 11](#_Toc250663181)

[6.3.1 Variable and Function Naming 11](#_Toc250663182)

[6.3.2 Length of Identifiers 11](#_Toc250663183)

[6.3.3 Macros and Define Identifiers 11](#_Toc250663184)

[6.3.4 Pointer Identifiers 11](#_Toc250663185)

[6.4 Use of Enumerations and Defines 11](#_Toc250663186)

[6.5 Avoid Performing Math on Enumerated Values 11](#_Toc250663187)

[6.6 Function Lengths 12](#_Toc250663188)

[6.7 File Lengths 13](#_Toc250663189)

[6.8 Explicit Integer Data Types 13](#_Toc250663190)

[6.9 Preprocessor Directives 13](#_Toc250663191)

[6.10 Macro Usage 13](#_Toc250663192)

[6.11 Class Guidelines 13](#_Toc250663193)

[7.0 Variable Storage Guidelines 14](#_Toc250663194)

[7.1 Global Variables 14](#_Toc250663195)

[7.1.1 Program-scope Variables 14](#_Toc250663196)

[7.1.2 File-scope Variables 14](#_Toc250663197)

[7.1.3 Function-scope Variables 14](#_Toc250663198)

[7.2 Passing Data as Pointers 14](#_Toc250663199)

[8.0 Control Flow Guidelines 15](#_Toc250663200)

[8.1 Use of goto 15](#_Toc250663201)

[8.2 Loop Guidelines 15](#_Toc250663202)

[8.2.1 Multiple continue Statements 15](#_Toc250663203)

[8.2.2 Multiple break Statements in loops 16](#_Toc250663204)

[8.2.3 Skipped break in switch Statements 16](#_Toc250663205)

[8.2.4 Altering the Loop Counter from within a for Loop is Discouraged 17](#_Toc250663206)

[8.3 Multiple return Statements 17](#_Toc250663207)

[9.0 General Design Guidelines 17](#_Toc250663208)

[10.0 Commenting 17](#_Toc250663209)

[10.1 dOxygen 18](#_Toc250663210)

[10.2 File Comment Blocks 18](#_Toc250663211)

[10.3 Function Comment Blocks 18](#_Toc250663212)

[10.4 Source Code Commenting 19](#_Toc250663213)

[10.5 Header File Commenting 19](#_Toc250663214)

[11.0 Miscellaneous Guidelines 19](#_Toc250663215)

[11.1 Compilation Warnings 19](#_Toc250663216)

[11.2 Type-casting 20](#_Toc250663217)

[11.3 Pointers in Conditional Statements 20](#_Toc250663218)

[11.4 Use of the sizeof Operator 20](#_Toc250663219)

[11.5 Comparing Constants with Variables in if Statements 20](#_Toc250663220)

[11.6 Macro Parenthesizing 20](#_Toc250663221)

[11.7 Conditional Compiles 21](#_Toc250663222)

#  Introduction

The goal of this document is to help produce readable, portable and maintainable code. The guidelines mentioned in this document will help the development team produce efficient and well-documented code. Since most or all of the coding is in C++, the guidelines will assume C++ as the primary coding language.

##  Layout

The general layout of the guideline will be as follows:

* **Description** – A brief description
* **Example** – An example illustrating the guideline
* **Additional** **Information** – Motivation/background of the guideline

##  Guideline Indicator

In the guideline sections, the terms *must*, *should* and *can* have special meaning and a special indicator. A *must* requirement must be followed, a *should* is a strong recommendation and a *can* is a general guideline/rule.



Must – Mandatory, the code must follow this rule.



Should – Recommended, the code should follow the guideline, but may waive it on discretion



Can – General Rule, the guideline promotes the goals of this document.



 Correct code.



 Incorrect code.

#  Document Maintenance

# 2.1 Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Rev** | **Rev Date** | **Description** | **Editor** |
| 0.x | 01/01/2010 | Initial Publication |  |
|  |  |  |  |
|  |  |  |  |

#  Design Guidelines

The major design goals are:

* Object-oriented
	+ Inheritance (code reuse)
* Encapsulation
* Robustness
* Portability
	+ OS Independence
	+ Architecture Independence
* No memory leaks
* Performance
* Simplicity

The following sections describe each goal in a bit more detail, though they are not meant to be a complete overview of the concepts in each section.

##  Object-oriented

One of the main design goals in any software project is to abstract data associated with domain concepts into classes of objects and take advantage of commonality in those classes to achieve code reuse through inheritance. This is often easier said than done, but time spent up front in this area results in a design that is often more extensible and robust. In its simplest form, the (iterative) process is:

* Find classes to represent domain concepts
* Specify the set of operations on those classes
* Find inheritance relationships among classes
* Refactor/reorganize the class hierarchies/design as needed

##  Encapsulation

One of the advantages of C++ over C is its ability to hide details of an implementation by declaring data members of a class private or protected. Data members should not be public.  Instead, public member functions should be defined to set and return the value of those data members that need to be public.

class RaidMap {

...

 int getStripSize() { return stripSize; }

 void setStripSize(int s) { stripSize = s; }

...

};

Member function access should be declared as private or protected unless they are really needed to be part of the public interface.

##  Robustness

By taking advantage of object inheritance and encapsulation supported in C++, the code can be made more robust. Inheritance can cut down on code duplication by sharing commonality of design and encapsulation helps isolate objects as components, such that a change to fix a bug in one area of the code is less likely to break something in an area unrelated to the area of the bug fix. Adhering to the standards proposed below will also contribute to making the code more robust.

Coupling is a property of objects and systems that should be minimized. If two objects are coupled in some way for them to operate, then making changes to one without considering the other is likely to break them. Decoupled, cohesive objects are key to a robust design.

##  Portability

###  OS Independence

In order to achieve the greatest design flexibility and generality, all OS-dependent or environment-specific code should be encapsulated in an interface module with a strict, well defined API.

A good indication of how well the OS-independent code is insulated from the environment where it’s executing is to include OS-specific header files only in the interface module. The OS-independent code should be able to compile cleanly with no OS-specific header files included. This allows the implementation to be more easily ported to other OSs, since only the code in the interface module needs to be modified.

###  Architecture Independence

In order to guarantee that code written to run on a machine with 32-bit architecture will also run on 64-bit architecture, some care must be taken. No assumptions should be made about the number of bits in built-in types other than char having 8 bits, short having at least 16 bits and long having at least 32 bits. It is best to define types that may vary with the architecture, explicitly in a header file included by all source files in the project. In addition, pointer arithmetic should be done using types that are architecture independent, such as UINT\_PTR in the Windows DDK.

#  Standards

The purpose of the following coding standards is to help reduce the occurrence of runtime errors, as well as to make the code more robust and more portable.

##  Compiler warnings

All code should be compiled by a compiler that has all warnings enabled. This is easier to do from the start than it is to do after a significant base of code has been developed.  If possible, compile with the compiler switch that treats warnings as errors.

##  Virtual Destructors

If a class has a virtual function, the destructor should be declared virtual, too. In an environment where objects are never destroyed (deleted), this is mainly to avoid the associated compiler warning.

##  Avoid Global Constructors

In C++, a global or static class variable, i.e. one whose type is not one of the built-in types like int and char, have their constructors called before main() gets control and have their destructors called after main() exits. Since the concept of ‘main()’ may not be well defined for the environment in which our code executes, it’s best to avoid the issue entirely by taking control of when static class variables get created and have their constructors called.

So instead of the following global or class static definition:

SomeClass classObj; // Automatically initialized before main()

use the following dynamic object creation

SomeClass\* classObj = NULL;

where

 classObj = new SomeClass;

appears in the system initialization function.

##

#  Readability and Portability Guidelines

This section describes the guidelines which help the code be more readable, especially during code reviews and in general. These guidelines would also make the code more portable across various text editors.

##  Line Formatting

### signal Line Lengths

The guideline recommends that there may be no more than 100 characters in each line of code. The goal is for the reader to see the end of every line when viewing on a 1024 by 768 monitor.

### signal White Space Use

White spaces should be used to make the code readable. Few examples are:

* Inserting spaces between operands and operators
* Using spaces to line up variable declarations
* Using blank lines to offset code segments
* Using spaces to align comments into blocks

### stop Tab Character Use Disallowed

There must be NO TAB characters in the code. This standard recommends use of parsing tools to find and replace any TAB characters present in the existing baseline code by 4 spaces.

The reason of having this guideline is to remove inconsistencies which occur when using ‘Diff’ and ‘Merge’ features if both tabs and spaces are present.

### stop Nested Operators

There must not be any nested operators to more than 8 levels.

##  Block Formatting

This standard recommends the use of “K&R” block format technique. The intent is to provide a consistent and predictable look over all the code.

### stop Block Format Type

The “K&R” style of block formatting must be used, where the curly brace to open a block is on the same line from the parent block, and the code of the child block is indented. The curly brace to close the block is on its own line indented to the level of the parent block. Example:

void main(void)

{

 int iVal;

 scanf(“value:”, &iVal);

 if (val < 10){

 val \*= 2;

 }

}

### stop Level Indent Size

The code editor must be set to use and indent of 4 spaces.

###  Examples of Block Formatting

For **if** statements:

if (<condition>){

 <block>

} else if (<condition>) {

 <block>

} else {

 <block>

}

For single line **if** statements:

if (<condition>) {

 <line>;

}

For **for** loops:

for (<loop conditions>) {

 <block>

}

For **while** loops:

while (<loop conditions>){

 <block>

}

For **switch** statements:

switch (<switch\_expr>) {

 case <case\_1>:

 <block>

 break;

 case <case\_2>:

 <block>

 break;

 .

 case <case\_n>:

 <block>

 break;

 default:

 break;

}

For **enum**, **struct**, and **union** declarations:

enum <enum\_name>

{

 <enum\_1>,

 <enum\_2>,

 .

 <enum\_n>

};

struct <struct\_name>

{

 <fields>

};

union <union\_name>

{

 <fields>

};

For **typedef** declarations of a previously defined type:

typedef <type> <typedef\_name>;

For **typedef** declarations of a newly constructed type:

typedef struct

{

 <fields>

} <typedef\_name>;

### signal Number of Nested Blocks

The number of nested block can be limited to 8 levels to avoid readability issues.

##  Naming Conventions

### signal Variable and Function Naming

This standard recommends adopting the “camel-hump” method. In this method each new word of the identifier is capital, except the first word. This promotes readability and consistency of code. Please note that this format is case-sensitive. A few examples are: srbIoControlSize, keyLength, tempBuffer etc.

One character variable names should be avoided, except for temporary “throw-away” variables.

The standard recommends that the data members of class should have a “m\_” prefix before the prefix specifying the type of the data.

### signal Length of Identifiers

The recommendation in this standard is to be prudent and strike a compromise between making the name meaningful and at the same time not making it too long to remember or confuse.

### stop Macros Typedefs and and Define Identifiers

When defining macros to specify constant values, the identifier must always be uppercase and appropriate underscore should be used to separate words. This also applies to typedefs. Example:

#define RAID\_CFG\_TYPE 5001

###  Pointer Identifiers

A lower case ‘p’ prefix must be used to identify any variable being used as a pointer. This is necessary to signal readers about the purpose of the variable throughout its scope boundary. Example:

PSCSI\_REQUEST\_BLOCK pSrb;

## stop Use of Enumerations and Defines

Arbitrary constants aka magic numbers must not be used in the code. These magic numbers are only obvious to the developer and that also for a short while. The use of these numbers makes the code difficult to maintain and increases chances of introducing bugs if the number needs to be modified in the future. Appropriate macros and or enumerations should be used to define these arbitrary constants.

## stop Avoid Performing Math on Enumerated Values

Mathematical operations must not be performed on enumerated values. This is bug prone and may cause a value to go out of range, which introduces ‘hard-to-find’ bugs in the code.

Example of correct and incorrect:

int advance\_state(int cur\_state)

{

 int ret\_state;

 if (cur\_stat <= 2){

 ret\_state = cur\_stat + 1;

 }

 else{

 ret\_state = 0;

 }

 return ret\_state;

}

enum state\_enum

{

 INITIAL\_STATE,

 MIDDLE\_STATE,

 ENDING\_STATE,

 NUM\_STATES /\* note: this enum tag must come last \*/

};

enum state\_enum advance\_state(enum state\_enum cur\_state)

{

 enum state\_enum ret\_state;

 if (INITIAL\_STATE == cur\_state){

 ret\_state = MIDDLE\_STATE;

 }

 else if (MIDDLE\_STATE == cur\_state){

 ret\_state = ENDING\_STATE;

 }

 else{

 ret\_state = INITIAL\_STATE;

 }

 return ret\_state;

}

enum state\_enum advance\_state(enum state\_enum cur\_state)

{

 enum state\_enum ret\_state;

switch (cur\_state){

 case INITIAL\_STATE:

 ret\_state = MIDDLE\_STATE;

 break;

 case MIDDLE\_STATE:

 ret\_state = ENDING\_STATE;

 break;

 case ENDING\_STATE:

 default:

 ret\_state = INITIAL\_STATE;

 break;

 }

 return ret\_state;

}

## signal Function Lengths

This standard recommends keeping the line count in any one function to a minimum – 60-90 lines is a good goal. The intent is to allow the reader to be able to understand everything a routine is doing without having to flip from page to page or scroll up and down.

## signal File Lengths

This standard recommends that the total number of lines within a file should be limited to 3000. Any file longer than this merits review.

##  Explicit Integer Data Types

This standard recommends establishing symbols that define the sign and size of variables, as opposed to taking the implicit compiler value, e.g., “int”. These standard definitions should be placed in a global header file accessible project-wide. Here are examples of this type of definition.

typedef signed char INT8;

typedef unsigned char UINT8;

typedef signed short INT16;

typedef unsigned short UINT16;

typedef signed int INT32;

typedef unsigned int UINT32;

## signal Preprocessor Directives

This standard recommends minimizing the use of preprocessor directives to enhance the understanding of the program flow and improve readability.

At no time should code within a preprocessor conditional block be dependent on code outside the preprocessor conditional block as in the following:

#ifdef DEBUG

 log\_debug(“Warning: Bad thing happening, condition = %d”,

#else

 printf(“Current condition = %d”,

#endif

Condition);

##  Macro Usage

This standard recommends limited use of macro, only on a need-to basis. Please note that the macros must not:

1. include other macros
2. Masquerade as functions.
3. Access local variables and use parameters instead

##  Class Guidelines

The standard recommends that each of the defined C++ classes should have their own .h file which contains the class declaration and a .cpp file which contains the class definition of the member functions.
Small classes which are more of helper and utility classes can be included in the same .h file.

The data members of the class should be declared public, protected and private , in that order.

The class member functions should not be defined within the class unless the functions are inline member functions and are atmost 1 line.

If the derived class has a member function which overrides a virtual member function in the base class, put the “virtual” keyword before the declaration of the function in the derived class, even though it is not required by C++ syntax. Having the “virtual” keyword before the declaration improves the readability of the code by reminding that the function is virtual.

#  Variable Storage Guidelines

##  Global Variables

### yield Program-scope Variables

This standard discourages the use of program-scope global variables to avoid convoluted and un-maintainable code. It is up to the discretion of the team to have a global variable but a lot of thought should be given before allowing a program-scope global variable.

###  File-scope Variables

This standard encourages a file-scope global variable as a much better alternative to program-scope global variable. These variables should be declared at the top of the file; use the **static** keyword to force them as file-scope. There should be appropriate comments to indicate the scope of the variable.

###  Function-scope Variables

The use of local variables is encouraged. When a reader of a program sees a local variable, he immediately knows the scope of that variable is limited to the function.

## yield Passing Data as Pointers

This standard recommends avoiding the practice of passing a structure by value. Instead, the structures should be passed as pointer or references. Example:

BOOL validate\_struct(OneStruct newStruct )

{

 BOOL valid = TRUE;

 if (newStruct.var1 < 7){

 if (newStruct.var2 < 11){

 valid = FALSE;

 break;

 }

 }

 return(valid);

}

BOOL validate\_struct(OneStruct \*newStruct )

{

 BOOL valid = TRUE;

 if (newStruct != NULL)

 {

 if (newStruct->var1 < 7){

 if (newStruct->var2 < 11) {

 valid = FALSE;

 break;

 }

 }

 }

 return(valid);

}

#  Control Flow Guidelines

This section explains how the execution of a program should be done in order to make the code easy to follow, debug and maintain.

## stop Use of goto

This standard recommends that the **goto** statement must not be used. Other alternative coding methods should be devised instead of using the **goto** keyword. Here is an example:

if (<condition1>)

{

 <block>

}

else

{

 goto label1;

}

if (<condition2>)

{

 <block>

}

label1:

if (<condition1>)

{

 <block>

 if (<condition2>)

 {

 <block>

 }

}

##  Loop Guidelines

### signal Multiple continue Statements

This standard discourages the use of multiple **continue** statements. Instead of using a **continue** statement, a conditional statement can be used. Here is an example using a **for** loop:

/\* loop with continue \*/

for (index = 0; index < 10; index++)

{

 int i = 5;

 if (i == 5){

 continue;

 }

 <code>

}

/\* loop with continue removed \*/

for (index = 0; index < 10; index++)

{

 int i = 5;

 if (i != 5){

 <code>

 }

}

### yield Multiple break Statements in loops

This standard discourages the use of multiple **break** statements in loops. A **break** statement prematurely terminates a loop. The abuse of this statement, like the **continue** statement, produces loops that are difficult to follow. An example is shown below:

/\* code with multiple break \*/

for (index = 0; index < imax; index++)

{

 if (doFSA)

 {

 FSAToStateA();

 if (cond1)

 {

 break;

 }

 FSAToStateB();

 if (cond2)

 {

 break;

 }

 FSAToStateC();

 break;

 }

}

/\* code with multiple breaks removed \*/

for (index = 0; index < imax; index++)

{

 if (doFSA)

 {

 FSAToStateA();

 if (!cond1)

 {

 FSAToStateB();

 if (!cond2)

 {

 FSAToStateC();

 }

 }

 break;

 }

}

Note the title of this section says “in loops”. The **break** statement must be used in **switch** statements.

### yield Skipped break in switch Statements

This standard recommends that for each case statement in a switch case, there should be a corresponding break. In certain situations, it is desirable to have the code flow from one case to the following case. In those situations, there must be a comment indicating the intention of omitting the break.

### yield Altering the Loop Counter from within a for Loop is Discouraged

The standard recommends that the loop counter should not be altered inside the loop. The loop counter should only be modified by the “advance section” of the **for** loop. If altering the loop counter is the only way to accomplish what is desired, this is an indication that a **while** or **do-while** loop should be employed.

##  Multiple return Statements

The standard discourages the use of multiple **return** statements in a function as it decreases the readability of code and makes it more difficult to modify.

Multiple return points can be avoided without producing a large number of nested conditionals by employing the “fall through” method. With the “fall through” method, a single level of nesting is used. A status variable is maintained and sections of a function are only entered while the status variable is valid. In addition, resources allocated by the function are initially set to some sort of unallocated status. Then at the end of the function, if an error has occurred, any successfully allocated resources before the error occurred are freed. Here is an example:

#  General Design Guidelines

The standard recommends the following Design guidelines to be followed at appropriate design scenarios.

* A member function, which does not affect the state of the object, should be declared as a const.
A public function must never return a reference or a pointer to a local variable.
* Every variable that is declared is to be given a value before it is used.
* Avoid usage of preprocessor macros in a place where function calls could be used. Use macros only where function calls can not be used.
* Always provide empty brackets ("[]") for delete when deallocating arrays. Otherwise, you'll get memory leak.
* Always provide the return type of a function explicitly.
* Use constant references (const &) instead of call-by-value, unless using a pre-defined data type or a pointer.
* Avoid redefinition of an inherited default parameter value.

#  Commenting

It is also one of the most difficult skills in which to develop a habit. The commenting of code, may seem unimportant but is perhaps the single most useful mechanism in making code readable. For precisely this reason, the programmer must provide detailed and accurate comments to the extent that the program is readable by persons other than the one who programmed it. It should also be noted that a complex piece of code should be refactored or rewritten rather than commented extensively.

Use either the // or /\* \*/ syntax, as long as you are consistent.

##  dOxygen

The commenting should also follow the dOxygen documentation standards established for the project, to allow extracting the comments automatically from the source code.

## stop File Comment Blocks

Every C- and H-file must begin with a comment block identifying the code as INTEL confidential and proprietary.

 /\*
 \* INTEL Confidential

 \* Copyright 2000-2010 Intel Corporation All Rights Reserved
 \*
 \* The source code contained or described herein and all documents related to \*\*

\*\* the source code ("Material") are owned by Intel Corporation or its \*\*

\*\* suppliers or licensors. Title to the Material remains with Intel \*\*

\*\* Corporation or its suppliers and licensors. The Material contains trade \*\*

\*\* secrets and proprietary and confidential information of Intel or its \*\*

\*\* suppliers and licensors. The Material is protected by worldwide copyright \*\*

\*\* and trade secret laws and treaty provisions. No part of the Material may \*\*

\*\* be used, copied,reproduced, modified, published, uploaded, posted, \*\*

\*\* transmitted, distributed, or disclosed in any way without Intel’s prior \*\*

\*\* express written permission. \*\*

\*\* \*\*

\*\* No license under any patent, copyright, trade secret or other intellectual \*\*

\*\* property right is granted to or conferred upon you by disclosure or \*\*

\*\* delivery of the Materials, either expressly, by implication, inducement, \*\*

\*\* estoppel or otherwise. Any license under such intellectual property \*\*

\*\* rights must be express and approved by Intel in writing. \*\*

\*\* \*\* \*/

## stop Function Comment Blocks

Every subroutine or function must begin with a function header comment block. At a minimum, the block should contain the function’s name, a description of what it does, a list of the parameters, and a description of the return value, if any.

 /\*\*
 \* @brief
 \*
 \* Description:.
 \*
 \* @param[in]
 \* @param[in]
 \* @param[in]
 \* @return TRUE if successful, FALSE if Not successful
 \*
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Here are the key points in the header format:

* /\*\*, at the top line, to identify this header to dOxygen
* @brief, to identify the routine by name
* @param[], either [in] or [out], to describe the formal parameters
* @return, to describe the function return value

The remainder of the function header should be devoted to the description of the purpose of the routine and the algorithms used to accomplish that purpose. You should target an audience of a peer programmer familiar with the project and with C coding.

##  Source Code Commenting

Source code commenting should be done at the level that a proficient C-programmer, other than the original author, can read the code and understand quickly what the code does. Granted, there is a lot of subjectivity in the preceding sentence. Mainly, the writer of the source code must keep in mind what might be difficult for others to understand. Code which optimizes all of the issues covered in this document will already have reached a certain level of readability. However, there will certainly be many places where even the most readable code still needs further commenting to be truly understandable by other programmers.

## yield Header File Commenting

This standard recommends against putting function headers to decorate the prototype definitions in a header file. The reasoning is that when debugging, the source file is shown (not the header file). If the comments are in the header file, the debugger must now open two files to read about the code.

#  Miscellaneous Guidelines

Here is the recommended way to organize source and header files.

* dOxygen Header, in project format
* Standard Intel file header, with copyright date and disclaimer
* For Header files: opening guard lines 🡪 #ifndef DEFINED #define DEFINED
	+ Note that the #define should be the same name as the filename.
* Include Files
* Macro and Typedef Definitions
* Prototypes
* For Source files: Global Variables
* For Source files: Code
* For Header files: closing guard lines 🡪 #endif /\* DEFINED \*/
* dOxygen Footer

##  Compilation Warnings

All the code in a project should be written such that no compilation warnings are generated. If the project team deems a warning acceptable, the programmer should devise a method to defeat the warning, for example by casting to void an unused formal parameter. The project team must accompany such methods by explicit comments to explain what and why of it.

##  Type-casting

This standard recommends against the use of casting in order to defeat compiler warnings. Avoid the use of arbitrary casting.

## stop Pointers in Conditional Statements

Pointers being checked for validity in conditional statements must be explicitly compared to the NULL constant. Implicit Boolean comparisons are not guaranteed to work on all machines, especially for those that implement ”real” and “protected” modes similar to those modes on the Pentium 3 chip On that chip, real mode NULL was 0, which protected mode NULL was pageX:0 (not 0). Most current CPUS treat all pointers the same, rendering this re``````

if (ptr) /\* bad – NULL may not be 0 on all machines \*/

-or-

if (!ptr) /\* bad \*/

if (NULL != ptr) /\* good \*/

-or-

if (NULL == ptr) /\* good \*/

## yield Use of the sizeof Operator

The **sizeof** operator should be used whenever the size of a data structure is required. Constants and **define’**s must not be used, since these are unreadable. In addition, when using the **sizeof** operator, it is more proper to use a data type than a variable itself.

## stop Comparing Constants with Variables in if Statements

Use the construct “if (CONST == var)”, so the compiler will detect the attempt to assign a value to a constant, detecting the “=/==” mistake described above.

if (x = NEW\_VALUE)

.

.

if (NEW\_VALUE == x)

.

.

## stop Macro Parenthesizing

Macros that accept parameters and macros that contain mathematical expressions must be fully parenthesized. In the absence of full parentheses, hard-to-detect errors can occur. Here are some examples:



#define MSG\_LEN HEADER\_LEN + BODY\_LEN

if (MSG\_LEN / 8 == 5) …

This is incorrect, because the order of operations is not explicit. The division by “8” would only be done on BODY\_LEN and not on the whole sum. The following is the correct definition:



#define MSG\_LEN (HEADER\_LEN + BODY\_LEN)

## signal Conditional Compiles

This standard recommends that conditional compiles should be kept to a minimum. Conditional compiles make code difficult to read, and often, it is difficult to determine which conditional compile defines are in effect. Instead of using many conditional compiles, conditional changes should be compartmentalized into grouped areas.