



# **Coding Standards for the C Programming Language**

Author: David Hildingsson

Date:2010-01-25

Rev: D

Reg. number: ARDY-CS-1010

Page 1 of 19

# Coding Standards for the C Programming Language

Updated 2010-01-25

## Contents

<b>1</b>	<b>Overview .....</b>	<b>5</b>
<b>1.1</b>	<b>Introduction.....</b>	<b>5</b>
<b>1.2</b>	<b>Scope.....</b>	<b>5</b>
<b>1.3</b>	<b>Purpose.....</b>	<b>5</b>
<b>1.4</b>	<b>Terminology.....</b>	<b>5</b>
<b>2</b>	<b>'C' Standards.....</b>	<b>6</b>
<b>2.1</b>	<b>Character Set.....</b>	<b>6</b>
2.1.1	All source code shall contain only the following ASCII characters .....	6
<b>2.2</b>	<b>File Content .....</b>	<b>6</b>
2.2.1	'C' source files (.c).....	6
2.2.2	'C' header files (.h) .....	6
<b>2.3</b>	<b>Line Length.....</b>	<b>7</b>
2.3.1	There shall be a maximum of 80 characters per line .....	7
<b>2.4</b>	<b>Indentation levels .....</b>	<b>7</b>
2.4.1	Indentation levels shall consist of three space (0x20) characters .....	7
<b>2.5</b>	<b>Braces.....</b>	<b>7</b>
2.5.1	Braces shall appear on a line by themselves, except where documented otherwise within this document .....	7
2.5.2	All text on lines between (but not including) opening and closing braces shall be indented at least one level from that of the corresponding braces.....	7
<b>2.6</b>	<b>Abbreviations/Acronyms .....</b>	<b>8</b>
2.6.1	Abbreviations/acronyms shall be mixed case with the first letter upper case and all following letters lower case .....	8
2.6.2	Abbreviations/acronyms shall be limited to those defined in an implementation-specific document, unless appearing in string literals (i.e. arrays of characters).....	8
2.6.3	Any word that has an accepted abbreviation/acronym shall not be spelled out in a source code statement .....	8
<b>2.7</b>	<b>Preprocessor directives.....</b>	<b>9</b>
2.7.1	General .....	9
2.7.2	#define.....	10
<b>2.8</b>	<b>Variables .....</b>	<b>11</b>
2.8.1	Each variable shall be defined or declared on a separate line .....	11
2.8.2	Variable names shall be mixed case and the first letter of each word shall be capitalized .....	11
2.8.3	Variables with file scope shall be declared 'static' if not referenced from outside the file in which they appear .....	11
2.8.4	Variables with file scope shall be defined before the first function.....	11
2.8.5	Local variables (those which are defined within functions) shall be defined before the first line of executable code within a function .....	11
2.8.6	Local variables (those which are defined within a block surrounded by braces) shall be defined before the first line of executable code within a block. ....	11
<b>2.9</b>	<b>Functions.....</b>	<b>12</b>
2.9.1	Function names shall be mixed case and the first letter of each word shall be capitalized.....	12
2.9.2	Function definitions shall be immediately preceded by a function header .....	12
2.9.3	Functions shall have a prototype .....	12
2.9.4	Functions shall be declared 'static' if not referenced from outside the file in which they appear ..	12
2.9.5	Functions shall specify 'void' if it takes no parameters.....	12

# Coding Standards for the C Programming Language

Author: David Hildingsson

Date:2010-01-25

Rev: C

Reg. number: ARDY-CS-1010

Page 3 of 19

2.9.6	Functions shall explicitly define the return type.....	12
2.9.7	There shall be no space between the function name and the opening parenthesis.....	12
<b>2.10</b>	<b>Prototypes.....</b>	<b>13</b>
2.10.1	A prototype shall exist for every function.....	13
2.10.2	Prototypes shall exactly duplicate the function definition (see above).....	13
2.10.3	Static functions shall be prototyped in the source module in which they are defined.....	13
2.10.4	Non-static functions shall be prototyped in the header file associated with the source module in which they are defined.....	13
2.10.5	Prototypes shall include names and types for all parameters.....	13
<b>2.11</b>	<b>Types.....</b>	<b>13</b>
<b>2.12</b>	<b>Structures.....</b>	<b>13</b>
2.12.1	All definitions shall be defined as a new type (i.e. typedef'd).....	13
2.12.2	All structure types shall have both a name and a tag.....	13
2.12.3	The name and tag of a structure type shall be identical.....	13
<b>2.13</b>	<b>Comments.....</b>	<b>14</b>
2.13.1	The <code>/*</code> comment delimiter shall be used for all comments except as otherwise specified in this document.....	14
2.13.2	The <code>/*</code> comment delimiter shall appear directly above the line (or block) to be commented for comments describing the methodology or flow of the code.....	14
2.13.3	The <code>/*</code> comment delimiter shall begin at the same indentation as the commented portion.....	14
2.13.4	Abbreviations shall not be used in comments.....	14
2.13.5	Comments shall be written in the third person form.....	14
2.13.6	Specific code tokens shall not be referenced in comments.....	14
2.13.7	Doxygen comments.....	14
<b>2.14</b>	<b>Code structure.....</b>	<b>15</b>
2.14.1	Operator precedence in compound operations shall be explicitly denoted by surrounding parenthesis, and not dependent on C precedence rules.....	15
<b>2.15</b>	<b>Whitespace.....</b>	<b>15</b>
2.15.1	<code>*</code> shall be preceded but not followed by a space in declarations and definitions.....	15
2.15.2	A space shall be placed after keywords when followed by an opening parenthesis.....	15
2.15.3	A space shall be placed after <code>;</code> and <code>,</code> , unless at the end of a line.....	15
<b>2.16</b>	<b>Keywords.....</b>	<b>15</b>
2.16.1	<code>if</code> statements.....	15
2.16.2	<code>do</code> loops.....	16
2.16.3	<code>for</code> loops.....	17
2.16.4	<code>while</code> loops.....	17
2.16.5	<code>switch</code> statements.....	17
<b>3</b>	<b>Standards Discussions.....</b>	<b>18</b>
<b>3.1</b>	<b>Conforming to the ANSI C specification.....</b>	<b>18</b>
<b>3.2</b>	<b>Using <code>/*</code> to delimit comments.....</b>	<b>18</b>
<b>3.3</b>	<b><code>#define</code> format for macros.....</b>	<b>18</b>
<b>3.4</b>	<b>Format for types and <code>#defines</code>.....</b>	<b>19</b>



## Coding Standards for the C Programming Language

Author: David Hildingsson

Date:2010-01-25

Rev: C

Reg. number: ARDY-CS-1010

Page 4 of 19

### Revision History

<b>Date</b>	<b>Rev</b>	<b>Changes</b>	<b>By</b>
2003-06-04	A	Initial release	DH
2003-07-11	B	Minor changes	DH
2003-08-26	C	Revision history added	DH
2010-01-25	D	Minor changes	DH

## 1 Overview

### 1.1 Introduction

Arguably, programming is a hybrid of science and art. The problem is attempting to distinguish between the two. The philosophy of this document is to only specify things that are quantifiable. Things that are not quantifiable are left for specification in another document.

These standards were developed from a variety of sources including other standards documents, software examples from language references, and input from literally hundreds of programmers. This document is not meant to be a final version or formal reference, but more as a dynamic document that is updated and revised as needed to reflect new technologies, methodologies, or preferences.

### 1.2 Scope

This document addresses the format, but not the content, of C source code. There are, therefore, certain things this document specifically does not cover, including:

- Naming conventions - The naming of things in a programming language is based on the perspective of the programmer as well as the specific nature of the application, and is therefore not quantifiable.
- Comment content – The selection of specific words to indicate some specific meaning when reading the code (such as WARNING or NOTE) is implementation specific and could change based on the sensitivity or familiarity to a given set of words.

This document outlines the specific standards to be used in the creation, preparation, and maintenance of all source files for code development. It is not meant to stand alone, as there are certain mandatory things that must be specified in an implementation specific document. These items that must be documented separately are indicated in this standard.

Formats required by the associated development tools (editors, compilers, assemblers, etc.) shall override any conflicting standard contained herein.

### 1.3 Purpose

The purpose of these coding standards is to facilitate the maintenance, portability and reuse of C source code. The documented conventions will create an environment where multiple programmers may work within and maintain the same source files with minimal differences and distractions. This will allow the evaluation of the programming techniques and algorithms without the introduction of style or etiquette discussions. Ultimately, this will minimize programming resources and create a more efficient work environment.

This document is a specification for the creation of source files. Conformance with the specification is required, but is not an item for review discussions. Conformance is to be obtained via inter-office communication and informal discussions with authors of out-of-spec code.

### 1.4 Terminology

The word 'shall' is used to indicate items that are required. The word 'should' is used to indicate recommendations and guidelines.

An indentation level, as referred to in this document, is the minimum amount of space that must be put at the beginning of a line.

## 2 'C' Standards

### 2.1 Character Set

#### 2.1.1 All source code shall contain only the following ASCII characters

1. Tab (0x09) is prohibited. See the discussion under Indentation below.
2. Space (0x20) through '~' (0x7E)
3. Linefeed (0x0A)
4. Carriage Return (0x0D)
5. Exceptions: Special Swedish characters may be used in comments.

### 2.2 File Content

#### 2.2.1 'C' source files (.c)

All 'C' source files shall have the following sections appearing in the order presented here, though any number of these sections may be empty in a given file.

##### 2.2.1.1 File header

The format of this section is specified in an implementation specific appendix to this document due to the fact that some of the information required in this section can be retrieved from Configuration Management Software if used. This section shall contain at least a reference to the name of the file, whether in the form of the archive name or the file name itself, as well as a short description as to the type of functionality the file provides. There shall not be function or variable lists in this section of the file. This header shall be the same as the header for 'c' header files (.h).

##### 2.2.1.2 C Source File Content

This includes anything except for function definitions and the following items:

- 'extern' declarations
- prototypes for functions not defined as 'static'

##### 2.2.1.3 Function Definitions

#### 2.2.2 'C' header files (.h)

All 'C' header files shall have the following sections appearing in the order presented here, though any number of these sections may be empty in a given file.

##### 2.2.2.1 File header

The format of this section is specified in an implementation specific appendix to this document due to the fact that some of the information required in this section can be retrieved from Configuration Management Software if used. This section shall contain at least a reference to the name of the file, whether in the form of the archive name or the file name itself, as well as a short description as to the type of functionality the file provides. There shall not be function or variable lists in this section of the file. This header shall be the same as the header for 'c' source files (.c).

## 2.2.2.2 *Start of inclusion redundancy protection*

This section provides protection against redundant inclusion of the header file in a given source file. This can happen if a source file includes two header files, with the second header file also explicitly including the first. The suggested form of this protection is to use some form of the filename in combination with the '#ifndef' and '#define' preprocessor statements. The actual specification is implementation specific to allow for a specification that uniquely protects every file in a given project.

Example:

```
ifndef MODULE_H
#define MODULE_H

/* body of module.h */

#endif /* not MODULE_H */
```

## 2.2.2.3 *C Header File Content*

Everything except for anything used only in a single C source file.

## 2.2.2.4 *End of inclusion redundancy protection*

This section signals the end of the information protected from redundant inclusion.

## 2.3 **Line Length**

### 2.3.1 There shall be a maximum of 80 characters per line

Limits the number of characters that must fit in a page width for printing.

Tools usually have line length limits, so compatibility with the majority of tools is insured.

When accessing code without standard development tools, the majority of display tools support 80 characters.

## 2.4 **Indentation levels**

### 2.4.1 Indentation levels shall consist of three space (0x20) characters

Some tools (debuggers or ICEs, for instance) may not allow the customization of tab widths. Due to the fact that there may not be alternative tools, using space characters for indentation ensures that the code appearance is still consistent in these tools.

## 2.5 **Braces**

### 2.5.1 Braces shall appear on a line by themselves, except where documented otherwise within this document

Enhances readability by forcing vertical white space

### 2.5.2 All text on lines between (but not including) opening and closing braces shall be indented at least one level from that of the corresponding braces

## 2.6 Abbreviations/Acronyms

### 2.6.1 Abbreviations/acronyms shall be mixed case with the first letter upper case and all following letters lower case

For example:

```
int Asap;
```

An alternate suggestion was to have these all upper case with the underscore ('\_') character separating words. By specifying the rule for mixed case, it eliminates the confusion for commonly used abbreviations/acronyms such as Radar, as well as allowing the reader to distinguish between the end of an acronym/abbreviation and the start of the next word or acronym/abbreviation.

### 2.6.2 Abbreviations/acronyms shall be limited to those defined in an implementation-specific document, unless appearing in string literals (i.e. arrays of characters)

Allows a standard vocabulary to be established and eliminates the confusion as to whether 'dis' stands for dispel, disable, discontinue, and so on.

For the purpose of minimizing the amount of textual output (to conform to length limitations, for example), or time the processor spends in performing output, it is often desirable to abbreviate commonly used words. Thus, abbreviations in strings are allowed, but should be documented in the source code and/or in the appropriate user document.

### 2.6.3 Any word that has an accepted abbreviation/acronym shall not be spelled out in a source code statement

Assures consistency in variable naming and syntax

## 2.7 Preprocessor directives

### 2.7.1 General

#### 2.7.1.1 *Arguments to preprocessor directives shall be in all upper case, with underscores ('\_') separating words, unless specified otherwise in this document.*

This is a common convention for the majority of, if not all, programmers.

For example:

```
#ifdef COMPILER_FOR_NEW_HARDWARE
#define TOTAL_COUNT 10
```

Allows easy identification of macros versus functions. This is useful if the following line in the code causes the indicated compiler error:

```
if( PROCESS( ) ) ...
Missing ';' at line xxxxx
```

where xxxxx is the stated line of code. If there was no way to distinguish between macros and functions, the debugging of this error is not immediately apparent. Knowing that 'PROCESS' is a macro will allow the user to quickly identify that the error is probably caused by something in the macro definition

#### 2.7.1.2 *The '##' preprocessor operator should not be used*

e.g. the following code should only be used after careful deliberation:

```
#define TYPE_DEF( id, name ) typedef unsigned id UNSIGNED_##name
TYPE_DEF( long, DWORD );
```

The code creates a type 'UNSIGNED\_DWORD' which is an unsigned version of the 'long' type. This style of programming is restricted because it is necessary to search for the string 'DWORD' in order to find the definition of the 'UNSIGNED\_DWORD' type (instead of searching for 'UNSIGNED\_DWORD', which won't be found).

#### 2.7.1.3 *Preprocessor directives shall match the indentation of the block before them*

#### 2.7.1.4 *Lines between conditional preprocessor statements (#if, #ifdef, #ifndef, etc.) shall be indented at least one level from that of the corresponding preprocessor lines*

```
#ifdef xxx
    ...statement(s) ...
#endif
```

## 2.7.2 #define

### 2.7.2.1 *Values of definitions shall be surrounded by opening and closing parenthesis ,‘( and ’*

For example:

```
#define ALPHA (BETA)           // Correct
#define ALPHA BETA             // Incorrect
#define ALPHA (BETA + GAMMA)   // Correct
#define ALPHA BETA + GAMMA     // Incorrect
```

The reason is that if the second definition labeled as ‘Incorrect’ was used in the source code as:

```
NewVariable = ALPHA * 4;
```

The result would be:

```
NewVariable = BETA + (GAMMA * 4);
```

instead of the desired:

```
NewVariable = (BETA + GAMMA) * 4;
```

### 2.7.2.2 *Macro parameters shall be surrounded by opening and closing parenthesis*

Allows expressions to be evaluated correctly when passed as a macro parameter. For example:

```
#define MULTIPLY_BY_FOUR( Num ) (Num * 4)
```

invoked as:

```
MULTIPLY_BY_FOUR( 3 + 2 );
```

will produce the result 11, instead of the desired result of 20. The correct macro definition is:

```
#define MULTIPLY_BY_FOUR( Num ) ((Num) * 4)
```

### **2.8 Variables**

2.8.1 Each variable shall be defined or declared on a separate line

2.8.2 Variable names shall be mixed case and the first letter of each word shall be capitalized

2.8.3 Variables with file scope shall be declared 'static' if not referenced from outside the file in which they appear

2.8.4 Variables with file scope shall be defined before the first function

This (combined with the standard below) allows programmers to easily locate variable definitions instead of having to scan through lines of code to find where variables are declared.

2.8.5 Local variables (those which are defined within functions) shall be defined before the first line of executable code within a function

This (combined with the standard above) allows programmers to easily locate variable definitions instead of having to scan through lines of code to find where variables are declared.

2.8.6 Local variables (those which are defined within a block surrounded by braces) shall be defined before the first line of executable code within a block.

This (combined with the standard above) allows programmers to easily locate variable definitions instead of having to scan through lines of code to find where variables are declared.

Local variables defined within a block should be avoided.

## 2.9 Functions

### 2.9.1 Function names shall be mixed case and the first letter of each word shall be capitalized

This causes there to be no difference between variable and function names, which is the way the compiler works. This standard eliminates the ambiguity of naming variables that are pointers to functions.

### 2.9.2 Function definitions shall be immediately preceded by a function header

The format of the function header is specified in an implementation specific appendix to this document to allow modification for specific documentation requirements. There shall not be modification lists in the function header. The function header must contain at least the function name, a description of the function, and documentation of any return values. The specification of the function header is implementation specific so that it may be designed to meet any additional development requirements.

### 2.9.3 Functions shall have a prototype

### 2.9.4 Functions shall be declared 'static' if not referenced from outside the file in which they appear

### 2.9.5 Functions shall specify 'void' if it takes no parameters

The 'C' language allows for an empty parameter list if no parameters are required.

### 2.9.6 Functions shall explicitly define the return type

The 'C' language standard specifies that if no return type is specified, the function is assumed to return an int.

### 2.9.7 There shall be no space between the function name and the opening parenthesis

Example: `void near MyFunction(...)`

### **2.10 Prototypes**

2.10.1 A prototype shall exist for every function

2.10.2 Prototypes shall exactly duplicate the function definition (see above)

2.10.3 Static functions shall be prototyped in the source module in which they are defined

2.10.4 Non-static functions shall be prototyped in the header file associated with the source module in which they are defined

2.10.5 Prototypes shall include names and types for all parameters

Advanced editors can present the prototype when the function is called. If the names of the parameters are present, the prototype gives the programmer an idea of what a given parameter is for.

### **2.11 Types**

Types shall end with ‘\_T’ or start with ‘T’. If simple types as byte, word, dword and similar are defined these types doesn’t need the type suffix or prefix.

### **2.12 Structures**

2.12.1 All definitions shall be defined as a new type (i.e. typedef’d)

Allows reference to the structure with just the type name

2.12.2 All structure types shall have both a name and a tag

Some tools understand structure tags and not type names for structures while others are the other way around. In order to avoid possible conflicts, providing both a name and a tag allows all tools to operate in the desired fashion

2.12.3 The name and tag of a structure type shall be identical

## 2.13 Comments

2.13.1 The `/**` comment delimiter shall be used for all comments except as otherwise specified in this document

Allows the use of the `/**`, `*/` comment pair to comment out blocks of code which may also include comments.  
See [Standards Discussions](#)

2.13.2 The `/**` comment delimiter shall appear directly above the line (or block) to be commented for comments describing the methodology or flow of the code.

Allows for a PDL (Pseudo-Design Language) parser. The parser will use the comment starting indentation as an indication of where to put the comment in PDL. This alleviates the need to maintain a separate design document. The use of advanced text editors with the capability to display comment lines differently than other lines of text allows for easy differentiation between comments and source code.

2.13.3 The `/**` comment delimiter shall begin at the same indentation as the commented portion

2.13.4 Abbreviations shall not be used in comments

Comments shall be used to describe the code at a high level

2.13.5 Comments shall be written in the third person form

The code is not a person, and the comments shall not reflect ownership by a specific group of people. For example, the use of the term 'we' shall be replaced by the term 'the software'. Note that this does not indicate that functional groups shall not be referenced. For instance, if there is one part of the code that handles the user interface, it is appropriate to indicate that the 'User Interface Software' does something, instead of just indicating 'the software'.

2.13.6 Specific code tokens shall not be referenced in comments

Comments shall not be dependent on code specifics, but instead shall reflect the processing or procedures used  
Example:

For the line of code:

```
NumBytesTransferred = 0;
```

Incorrect:

```
// Set NumBytesTransferred to 0
```

Correct:

```
// Set the count of the number of bytes transferred to 0
```

2.13.7 Doxygen comments

Comments shall be formatted so documentation (HTML or PDF format) can be generated with the doxygen and graphviz tools. See <http://www.doxygen.org> and <http://www.graphviz.org/>.

## 2.14 Code structure

2.14.1 Operator precedence in compound operations shall be explicitly denoted by surrounding parenthesis, and not dependent on C precedence rules.

```
NewValue = (OldValue + (5 * X)) - 4;
```

Eliminates any confusion regarding algorithms based on a lack of understanding of the standard C precedence rules

## 2.15 Whitespace

2.15.1 '\*' shall be preceded but not followed by a space in declarations and definitions

2.15.2 A space shall be placed after keywords when followed by an opening parenthesis

```
for ( ... )  
if ( ... )  
return ( ... );
```

2.15.3 A space shall be placed after ';' and ',', unless at the end of a line.

```
for ( x = 1; y == 2, z == 3; )
```

## 2.16 Keywords

2.16.1 'if' statements

```
if ( xxx )  
{  
    ...statement(s) ...  
}  
else if ( yyy )  
{  
    ...statement(s) ...  
}  
else  
{  
    ...statement(s) ...  
}
```

## 2.16.1.1 *Shall have braces even if there exists none or only one statement*

Allows addition of debug or additional statements with minimal work

Handles particular case if statement is a macro, as described below:

```
if( xxx )
    Process( x );
else
    ...statement(s)...
```

Suppose **Process** is a macro as follows:

```
#define Process( x )    Handle( x )
```

Now, suppose **Process** must be modified so that it creates a variable, puts the parameter in it, and calls **Handle**, as follows:

```
#define Process( x ) \
{ \
    int Arg[2]; \
    Arg[0] = x; \
    Arg[1] = 0; \
    Handle( Arg ); \
}
```

This macro will fail to compile because when the macro is substituted, the ';' which follows the **Process( x )** statement will be appended after the '}' in the macro, terminating the 'if' statement. This will cause the compiler to choke because it came across an 'else' statement with no corresponding 'if' statement.

## 2.16.1.2 *Final 'else' (if it exists) shall be on a line by itself*

### 2.16.2 'do' loops

```
do
{
    ...statement(s) ...
}
while ( xxx );
```

#### 2.16.2.1 *Shall have braces even if there exists none or only one statement*

Allows addition of debug or additional statements with minimal work

Suffers from same weakness as 'if' statements

#### 2.16.2.2 *'while' shall be on the same line as the closing brace*

Allows differentiation between an empty 'while' loop and 'do-while' loops

#### 2.16.2.3 *do' shall be on a line by itself*

## 2.16.3 'for' loops

```
for ( xxx; yyy; zzz )
{
    ...statement (s) ...
}
```

### 2.16.3.1 *Shall have braces even if there exists none or only one statement*

Allows addition of debug or additional statements with minimal work

Matches standards for 'if' and 'do-while' statements.

## 2.16.4 'while' loops

```
while ( xxx )
{
    ...statement (s) ...
}
```

### 2.16.4.1 *.Shall have braces even if there exists none or only one statement*

Allows addition of debug or additional statements with minimal work

Matches standards for 'if' and 'do-while' statements.

## 2.16.5 'switch' statements

```
switch ( zzz )
{
    case #1:
        ...statement (s) ...
        break;
    ...
    default:
        ...statement (s) ...
        break;
}
```

### 2.16.5.1 *Shall have braces even if there exists none or only one statement*

Allows addition of debug or additional statements with minimal work

Matches standards for 'if' and 'do-while' statements..

### 2.16.5.2 *'case' labels shall be indented once from the braces*

### 2.16.5.3 *Statements and 'break's shall be indented twice from the braces*

### 2.16.5.4 *If a 'break' is left out of one of the case statements, a comment shall be written to indicate that the fall-through behavior is desired*

### 2.16.5.5 *The 'default' label shall always appear.*

This ensures that any case no explicitly handled is caught.

## 3 Standards Discussions

During the course of development of this standard, there have been numerous discussions on each of the items above. Some of these discussions warrant documentation so that the history behind decisions made with respect to this document is understood.

### 3.1 *Conforming to the ANSI C specification*

The ANSI standard for C is flawed in a number of ways that make it more difficult to implement well-engineered software. For instance, the ANSI standard allows two (or more) separate modules to declare a variable X, which, when linked, are resolved to the same memory location. This causes problems if separate programmers author the two files and don't realize the variable name already exists. Another problem is the ANSI specification that only the first 32 characters of identifiers are unique. This means that programmers must explicitly ensure that two similar variable names differ within the first 32 characters, or they will be considered the same variable. Combine this issue with the first and there is a potential problem that is very hard to solve.

### 3.2 *Using `/**` to delimit comments*

Although the `/**` is not ANSI standard, most (if not all) compilers now support this mechanism. In addition, it allows the nicety of using the older comment delimiters `/*` and `*/` to comment out entire blocks of code. Removing blocks of code can also be done by surrounding the code with a `#if 0...#endif` block. Although both methods provide the result desired, if the `/*` and `*/` combination is used only for this purpose (i.e. these sequences don't occur anywhere else in the code), it is arguably easier for a chroma-coding parser to color these blocks (since there are no imbedded comments within the block). Since there is no strong technical argument either way, this item in the standard could be done either way.

### 3.3 *#define format for macros*

Initially, the formatting of (`#define`) macros followed the same naming convention as functions, except with a pre-pended `M_` to indicate that the program referred to a macro as opposed to a function (see the current specification for macro format for an explanation of why this differentiation is important). This formatting was chosen to reflect the fact that macros are used in a way similar to functions. However, it required the artifice of adding the `M_` to the name in order to distinguish between the two. By having the macros the way they are currently specified, there is an inherent differentiation between macros and functions.

## 3.4 *Format for types and #defines*

Many standards use all upper case to indicate a type name. The same standards indicate that '#defines' are also in all upper case. This results in a preprocessor directive (#define) having the same format as compiler syntax. This can cause name conflicts, especially when working with large code bases or third-party software, when the name of a #define is identical to the name of a type. This is also referred to as conflicting declarations between namespaces<sup>1</sup>. If the preprocessor and compiler are separate processes, the compiler will generate an error that is not clear as to the origin. Debug requires searching all included information, including third-party software, to locate the name conflict

For a discussion on name spaces (or overloading classes in C), see<sup>1</sup>.

One of the suggested solutions to this problem is to have a naming convention to indicate the difference between types and the #define constants, for instance, appending an \_E for enums, \_S for structures, and an \_T for typedefs. As specified above, the approach for this document is to not restrict the names for things, but simply the format of the names. But there are other reasons to not implement this solution:

1. Changing from one type (i.e. a typedef) to another (i.e. a structure) requires a change to every line of code using this type. This is prohibitive in a large code base.
2. The \_T, \_E, ... can still collide with other #define names. For example, providing a #define for all letters on the keyboard could be 'KEYBOARD\_A', ...'KEYBOARD\_T', etc. In this case the '\_T' does not indicate the item is a type.
3. End users of a type should not need to know the specific implementation. This reason also goes toward object oriented programming in C. Nothing but the implementation of the type should 'know' how the type is implemented. Everything outside the implementation of the type should interact with the type through interface functions/macros.

The solution presented in this document puts types and #define constants in separate namespaces, thereby eliminating the aforementioned problems.

There is also precedence for this standard: ANSI standards have all types in all lower case letters.

---

<sup>1</sup> C, a reference manual / Samuel P. Harbison, Guy L. Steele, Jr. -- 3rd ed., Copyright 1991 Prentice-Hall, Inc. pp59-60.