

Openair Coding Guidelines

Editor:	EURECOM
Deliverable nature:	Public
Due date:	July, 2015
Delivery date:	July, 2015
Version:	0.3
Total number of pages:	24
Reviewed by:	
Keywords:	

List of authors

Company	Authors
EURECOM	Christian BONNET, Lionel GAUTHIER, Rohit GUPTA, Florian KALTENBERGER, Raymond KNOPP, Navid NIKAIEN, Cedric ROUX.
OPEN CELLS	Laurent THOMAS.
OAI SA	Raphael DEFOSSEUX

History

Modified by	Date	Version	Comments
Lionel GAUTHIER	15/07/15	0.1	Initial mix of http://users.ece.cmu.edu/~eno/coding/CCodingStandard.html and ftp://ftp.estec.esa.nl/pub/wm/anonymous/wme/bssc/bssc2000(1)i10.PDF .
Laurent THOMAS	17/07/15	0.2	Cosmetics updates
Raphael DEFOSSEUX	18/10/2018	0.3	Removed all parts related to formatting It is taken care in wiki pages

Table of Contents

History	3
Table of Contents	4
1 Names	5
1.1 Make Names Fit	5
1.2 Pre-Processor Names	6
1.3 Macro Names	6
1.4 Function Names	7
1.5 Structure Names	7
1.6 Type Names	7
1.7 Enumerated types	8
2 General declaration guidelines	9
2.1 Declaration of Constants	10
2.2 Declaration of Variables	10
2.2.1 Initialize all Variables	11
2.3 Declaration of Types	11
3 Structs and unions	12
4 Guidelines for functions	13
5 Expressions	15
5.1 Conditional expressions	15
5.2 Order of evaluation	15
5.3 Use of white spaces	17
5.4 Memory allocation	17
5.5 Error handling	17
5.6 The use of the pre-processor	17
6 Portability	20
6.1 Data abstraction	20
6.2 Representation	20
6.3 Pointers	20
7 Formatting	22
7.1 To Use Enums or Not to Use Enums	22
7.2 Pointer Variables	22
7.3 Variable names on the stack	22
7.4 Use header file guards	22
8 Miscellaneous	23
8.1 Document Null Statements	23
8.2 Use #if Not #ifdef	23
8.3 No Magic Numbers	23
8.4 Error Return Check Policy	24

1 Names

1.1 Make Names Fit

Names are the heart of programming. In the past people believed knowing someone's true name gave them magical power over that person. If you can think up the true name for something, you give yourself and the people coming after power over the code. Don't laugh! A name is the result of a long deep thought process about the ecology it lives in. Only a programmer who understands the system as a whole can create a name that "fits" with the system. If the name is appropriate everything fits together naturally, relationships are clear, meaning is derivable, and reasoning from common human expectations works as expected. If you find all your names could be Thing and Do It then you should probably revisit your design.

Names shall not start with an underscore character (_).

Although C and C++ allow identifiers which begin with an underscore the programmer is strongly advised against using such names. The [emerging] C++ standard specifies that identifiers, which begin with two underscores, are reserved for compiler and library use. Traditional C libraries make use of identifiers which begin with an underscore for low level implementation details which the programmer is never supposed to use directly. Therefore, to avoid any inadvertent conflict with compiler, library or operating system identifiers the programmer is expressly forbidden to use names which begin with an underscore.

Names shall be meaningful and consistent.

Descriptive names are preferable to "random" names. However, the descriptive name must reflect the use of the entity over its lifetime. For example, "count" is better than "xyz" for the name of a variable which contains the total number of something but not if it is used to hold an error code.

Names containing abbreviations should be considered carefully to avoid ambiguity.

In the absence of a list of standard abbreviations different programmers are likely to choose different abbreviations to represent a particular word or concept. This is especially true in the multi-lingual environment found in many ESA projects. Abbreviations should be used with care. For example, does "opts" refer to "options" or the "old points" or something else?

Avoid using similar names, which may be easily confused or mistyped.

Names which differ by only one character, especially if the difference is between "1" (the digit) and "l" (the letter) or "0" (the digit) and "O" (the letter). Avoid names, which consist of similar meaning words because it is easy to confuse them. Consider the differences between "x1" and "xl" and between "countX" and "numberX".

Include Units in Names

If a variable represents time, weight, or some other unit then include the unit in the name so developers can more easily spot problems. For example:

```
uint32_t rlc_timeout_msecs;  
uint32_t hw_global_rx_gain_dbm;
```

Globals

Names for system global entities shall contain a prefix, which denotes which subsystem or library contains the definition of that entity.

The subsystem or library prefix provides additional information about the origin of an entity, which can be useful to the programmer. The basic name (without the prefix) can be reused elsewhere in the source code without causing a name clash.

Example in core network, S-GW and P-GW global variables:

```
sgw_app_t sgw_app;  
pgw_app_t pgw_app;
```

Global Variables

Global variables should be avoided whenever possible.

Global Constants

Global constants should be all caps with '_' separators.

Justification

It's tradition for global constants to be named this way. You must be careful to not conflict with other global *#defines* and enum labels.

Example

```
const int A_GLOBAL_CONSTANT= 5;
```

1.2 Pre-Processor Names

Pre-processor names shall consist of upper case words separated by underscore.

Established C convention uses upper case only names for pre-processor objects such as literal constants and macros. This emphasizes the difference between true C constructs and pre-processor entities, which are not really part of the C language at all.

Pre-processor entities provide a substitution mechanism in the code itself rather than being part of the code. Pre-processor names are not bound by any of the rules of scope which apply to C and C++ names and exist from their definition until the end of compilation of the module unless explicitly redefined or even undefined.

Programmers are therefore advised to use only upper case in order to distinguish pre-processor entities from true C/C++ constructs. The use of underscore to separate words improves the readability of the name.

Pre-processor names which are defined in an interface file are not truly global in the sense that has been described previously but are local copies within any module which uses that interface file. It is still helpful to be able to determine where such names are defined.

Therefore "global" pre-processor names should also include the prefix corresponding to the subsystem or library in which they are defined, e.g.

UI_MAX_WINDOW_INDEX. "Local" pre-processor names do not need this additional prefix, e.g. SUCCESSFUL.

1.3 Macro Names

Macro names shall consist of upper case words separated by underscore.

Spacing before and after the macro name may be any whitespace, though use of TABs should be consistent through a file. If they are an inline expansion of a function, the function is defined all in lowercase, the macro has the same name all in uppercase. If the macro is an expression, wrap the expression in parenthesis. If the macro is more than a single

statement, use `do { ... } while (0)`, so that a trailing semicolon works. Right-justify the backslashes; it makes it easier to read.

Justification:

This makes it very clear that the value is not alterable and in the case of macros, makes it clear that you are using a construct that requires care.

Some subtle errors can occur when macro names and enum labels use the same name.

1.4 Function Names

Function names consist of one or more words where each word is in lower case.

Function names should probably use a combination of verb(s) and noun(s) although care must be taken to avoid ambiguity. Usually every function performs an action, so the name should make clear what it does:

`check_for_errors()` instead of `error_check()`, `dump_data_to_file()` instead of `data_file()`. This will also make functions and data objects more distinguishable.

`if (empty_thing(&thing) == TRUE)`

mean that the variable "thing" is in fact empty, or that it has been emptied successfully?

Changing the name to `is_empty_thing()` would make the meaning clear.

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

Suffixes are sometimes useful:

- *max* - to mean the maximum value something can have.
- *cnt* - the current count of a running count variable.
- *key* - key value.

For example: `retry_max` to mean the maximum number of retries, `retry_cnt` to mean the current retry count.

Prefixes are sometimes useful:

- *is* - to ask a question about something. Whenever someone sees *is* they will know it's a question.
- *get* - get a value.
- *set* - set a value.

For example: `is_hit_retry_limit`.

1.5 Structure Names

Structure names consist of one or more words where each word is in lower case. Use underbars ('_') to separate name components.

When declaring variables in structures, declare them organized by use in a manner to attempt to minimize memory wastage because of compiler alignment issues, then by size, and then by alphabetical order. E.g, don't use `int a; char *b; int c; char *d`; use `int a; int b; char *c; char *d`. Each variable gets its own type and line, although an exception can be made when declaring bit fields (to clarify that it's part of the one bit field). Note that the use of bit fields in general is discouraged (portability problems). Major structures should be declared at the top of the file in which they are used, or in separate header files, if they are used in multiple source files.

1.6 Type Names

User defined type and class names consist of one or more words where each word is in lower case plus an appropriate prefix or suffix.

Using an appropriate prefix or suffix as defined in the following rules distinguishes these names from both variables and functions. Note that names should first include a library prefix if necessary.

User defined type names shall end with "_t" .

Note that in C a programmer must explicitly associate a type name with "struct", "union" and "enum" constructs using "typedef".

1.7 Enumerated types

Labels All Upper Case with '_' Word Separators. This is the standard rule for enum labels. No comma on the last element.

Each enumeration within an enumerated type shall have a consistent prefix.

It is not uncommon for different enumerated types to be used within a program which have similar characteristics, such as those representing the "status" of different parts of the system. Although they are distinct, enumerations within one type may have similar roles to those of another type. To reduce the risk of name clashes for such enumerations, the name of each enumeration should include a prefix which denotes the exact enumerated type to which it belongs. This also reduces possible confusion when converting to or from integer variables.

Example in NAS layer:

```
typedef enum {
    EMM_INVALID,
    EMM_DEREGISTERED,
    EMM_REGISTERED,
    EMM_DEREGISTERED_INITIATED,
    EMM_COMMON_PROCEDURE_INITIATED,
    EMM_STATE_MAX
} emm_fsm_state_t;
```

2 General declaration guidelines

Each declaration should start on its own line and have an explanatory comment.

For some language constructs, such as the definition of pre-processor macros, this rule is implicit, and for others such as function or class definition it is unlikely that a programmer would want to declare more than one per line. However, it provides a general principle for all constructs and should be explicitly applied for members of enumeration, structures, classes, variable declarations and function parameters.

The rule means that the programmer always has the opportunity of adding or removing declaration lines without having to modify other declarations and provides space at the end of the line for an explanatory comment.

Entities should be declared to have the shortest lifetime or most limited scope that is reasonable.

The namespace within the source code is limited and therefore it makes sense not to clutter it up unnecessarily. Some entities correspond to the internal implementation of particular modules and should not be exposed elsewhere.

Global entities must be declared in the interface file for the module.

The only way to ensure consistency between different modules is to use the same information in each module. The interface file forms that single point of reference.

Rule 49. Declarations of "extern" variables and functions may only appear in interface files.

An interface file provides a means of telling modules that a variable or function has been defined somewhere but not exactly where. If the programmer uses "extern" declarations within implementation files this increases the number of places which need to be changed if the variable or function is changed.

Declarations of static or automatic variables and functions may only appear in implementation files.

This rule follows on from the previous rule but it is, in fact, poorly worded. A variable, which is declared as "extern" only, provides a placeholder, which is used to bind the variable name with the true definition of the variable.

A static or automatic variable declaration really provides the direct declaration of the variable. Therefore, each module which contains such a declaration contains a definition of its own variable. In many cases this is perfectly acceptable and allows variable names to be reused for different purposes in different modules.

If the declaration of a static or automatic variable appears in an interface file, then each module which includes the interface file will contain its own definition of its own variable. This is likely to lead to confusion and errors which are difficult to track down because the programmer will assume that the variable has global scope when in fact there are many different versions of the variable.

Order of declarations

Should appear in the order: constants and macros; types, structs and classes; variables; and functions.

Symbolic constants and macros are likely to be used within type, struct and class declarations, which in turn are likely to be used for declaring variables. Such system or file global variables are likely to be used in the functions. Grouping each category of entities together allows the programmer to find a particular declaration much more quickly than if entities were declared "as needed".

2.1 Declaration of Constants

Symbolic constants shall be used in the code

"Magic" numbers and strings are expressly forbidden.

Code, which contains literal constants, can be very hard to understand because there may be no obvious reason why a particular number, size or string appears in the code. This makes maintenance of the code difficult. Maintenance or future modification of the code is also complicated because whenever one of these literal constants needs to be changed the programmer must examine every occurrence of that literal constant in the code to determine whether it is being used for the same purpose before being able to update it.

It is far better to define a symbolic constant once for a particular meaning of a literal constant and then use this symbolic constant. This improves readability of the code in the areas where the symbolic constant is used, and distinguishes between different symbolic constants, which may have the same literal constant value. Modification is also easier because there is only one place, which needs to be changed.

For C: all symbolic constants shall be provided using an "enum" or the #define mechanism.

The use of an "enum" is preferred over the use of #define for small groups of related integer constants because the compiler is able to perform additional type checking.

2.2 Declaration of Variables

Global variables shall be declared as "extern" towards the top of the interface file, and defined at the top of the implementation file, above any function declarations. The definition in the implementation file may also include initialization of the variable.

Each variable shall have its own personal declaration, on its own line

Having a complete variable declaration on each line means that the type and name and possibly a brief descriptive comment about use or properties of that variable are always next to each other.

One problem with variable declarations is that each variable has a type and may also have a "type-qualifier". Some people believe that the "type-qualifier" belongs with the type, and others believe that it belongs with the variable. Where variables with different "type-qualifiers" are declared in lists it is not always possible to position the "type-qualifiers" in a consistent manner. With one variable declaration per line there can be no confusion.

Whichever style is chosen should be used consistently throughout the code!

In the absence of any logical grouping of individual variables, the variable declarations should be grouped together by type and alphabetically within each type. This allows declarations to be found more easily.

```
/*
 * these declarations are unclear
 */
int x, y, *z; /* x,y are ints but z is pointer to int */
char* reply, c; /* reply is pointer to char, but c is char! */

/*
 * these declarations are better
 */
char c; /* current character during string handling */
char *reply; /* user's response to prompt */
int x; /* x coordinate within frame */
```

```
int y; /* y coordinate within frame */
int *z; /* pointer to z coordinate */
```

2.2.1 Initialize all Variables

- You shall always initialize variables. Always. Every time. gcc with the flag -W may catch operations on uninitialized variables, but it may also not.

Justification

- More problems than you can believe are eventually traced back to a pointer or variable left uninitialized.

2.3 Declaration of Types

Programmers should be encouraged to use "typedef" to create type names which are meaningful in the problem domain. This promotes readability because it is immediately obvious what the "dimensions" of a variable should be.

3 Structs and unions

A self-referential struct must have a tag name or forward declaration.

The use of a tag name may be clearer in that the complete declaration appears in one place, but means that two names are used for each struct. Note that the tag name is only ever used in the main declaration of the struct.

```
typedef struct binary_tree_s
{
int nValue; /* useful info */
struct binary_tree_s *pLeftSubTree; /* values < nValue */
struct binary_tree_s *pRightSubTree; /* values >= nValue */
} binary_tree_t;
```

The use of a forward declaration is less clear, but only uses one name. The main declaration of the struct resembles the C++ approach even when using C so this can provide an upgrade path for the code without major modification. If the internal details of the struct can be hidden within the implementation file then only the forward declaration will be needed in the interface file.

```
typedef struct binary_tree_s binary_tree_t;
struct binary_tree_s
{
int nValue; /* useful info */
binary_tree_t *pLeftSubTree; /* values < nValue */
binary_tree_t *pRightSubTree; /* values >= nValue */
};
```

4 Guidelines for functions

A full function prototype shall be declared in the interface file(s) for each globally available function.

The ANSI C and C++ standard provide function prototypes as a means of "forward-declarations" of function, which are defined elsewhere. Function prototypes are a big improvement on the "old K & R style" because the compiler is able to provide consistency checking of the return type and the number and type of arguments given by the prototype compared to those used in the function call or the actual declaration of the function.

Each function shall have an explanatory header comment.

Each function (or prototype) shall be preceded by a brief comment block, which describes the purpose of the function and its parameters. Any assumptions made about the parameters should also be mentioned.

Each function shall have an explicit return type.

A function, which returns no value, shall be declared as returning "void".

A function, which does not have an explicit return type, is assumed to return an integer. This can lead to problems if there is a mismatch between what the caller of the function expects and what the function actually returns. If the return type is not specified explicitly the compiler cannot issue appropriate warnings if the function is called incorrectly or if the return value is invalid.

A function, which takes no parameters, shall be declared with a "void" argument.

If a function does not take any argument, then the programmer should make this explicit by specifying "void" in the argument list. This should also be reflected in the function prototype. This avoids any confusion between the function prototype and the "old K & R style" of declaration, which contained no information about the function's arguments.

A parameter, which is not changed by the function, should be declared "const".

The compiler should be used to trap as many problem areas as possible. If a parameter should remain unchanged by a function, then the programmer should inform the compiler so that it can issue a warning if the parameter is changed inadvertently.

The "assert()" macro, or similar, should be used to validate function parameters during program development.

The ANSI C and [emerging] C++ standard provide an "assert()" macro. The "assert()" macro causes the program to abort with some diagnostic information if the integer expression which is given as the argument evaluates to zero, i.e. false. This facility is enabled during development and disabled prior to release by defining a particular preprocessor variable before compilation.

During development it is common for a parameter to contain a value which is incorrect. There may be several reasons for this: there is an error in the code or the value was not foreseen or the code, which handles this particular value, is not yet available. The use of assertions to validate the parameters means that such values are trapped the first time that they are encountered during development.

A function may not return a reference or pointer to one of its own local automatic variables.

With the notable exception of static data variables within a function, all local variables are created on entry into the function and returned to the system on exit from the function. Therefore, the caller has no guarantee that the memory corresponding to what was once the local data variable still contains anything useful or whether it has already been overwritten by something else.

The programmer should be careful not to introduce other potential memory problems in an attempt to work around the one described here. Although it is unlikely to occur in an ordinary function which return simple types, please see Rule 83 for further details.

5 Expressions

5.1 Conditional expressions

Conditional expressions must always compare against an explicit value.

The traditional C idiom has always favored brevity in code and this is extended to the implicit comparison of values against zero, which is also used to denote false. In some cases this can be counter-intuitive and the programmer reading the code must expend some effort to understand what is actually happening.

The other area where this can be problematic is when dealing with the return codes from functions. In many cases there is a range of return codes which may indicate different levels of success or failure of the function and not just zero (false) and non-zero (true).

Different routines are not consistent in their use of return codes to indicate success and failure. Many routines which deal with pointers return NULL (i.e. zero) to indicate failure, while many others use a negative number to indicate failure. The common string comparison routine `strcmp()` uses zero to indicate equality!

Routines, which use enumeration as the return value, can suffer when a new member is added to the enum, which may cause the renumbering of the existing values, and invalidate any implicit comparisons.

The programmer is therefore encouraged to use explicit comparisons in the code so that the meaning is clear, and to safeguard against changes in the underlying implementation of functions and their return types.

```
if (Function(parameter) == SUCCESSFUL)
```

is guaranteed to continue working as expected even if the actual value associated with the return code `SUCCESSFUL` is ever changed. It doesn't matter whether `Function()` returns a negative number, zero, or a positive number as long as `SUCCESSFUL` has the appropriate value. The symbolic name also helps to clarify the code. On the other hand, the following fragment is unclear and will not continue to work correctly if the return value is changed to or from zero.

```
if (Function(parameter))
```

Note also that the language standards define zero as meaning false. They do not define a particular non-zero number to mean true. It is theoretically possible to fall foul of a compiler if a symbolic constant for true is defined to be a particular value. For example if `MyFalse` is declared to be 0 and `MyTrue` to be 1 and if value is 2 then both `value!=MyFalse` and `value!=MyTrue` are true at the same time. This does not lead to intuitive code!

5.2 Order of evaluation

The programmer shall make sure that the order of evaluation of the expression is defined by typing in the appropriate syntax

The order of evaluation of individual expressions within a statement is undefined, except when using a limited number of operators.

The comma operator in "expression1, expression2" guarantees that expression1 will be evaluated before expression2. Note that the comma operator is not the same as the comma used as a separator between function arguments.

The logical-AND operator in "expression1 && expression2" guarantees that expression1 will be evaluated first. expression2 will only be evaluated if expression1 is true.

The logical-OR operator in "expression1 || expression2" guarantees that expression1 will be evaluated first. expression2 will only be evaluated if expression1 is false.

The conditional operator in "expression1 ? expression2 : expression3" will evaluate expression1. If it is true expression2 is evaluated, otherwise expression3 is evaluated.

All expressions, which appear as function arguments, are evaluated before the function call actually takes place. Note that the order of evaluation of the arguments is not specified.

In addition, a "full expression" is the enclosing expression that is not a subexpression. The compiler will evaluate each full expression before going further.

The operators, etc. which are explained above are known as "sequence points" and can be used to determine the order of evaluation of the expressions in some statements. However the order of evaluation of expressions between each sequence point is undetermined.

This means that different compilers may produce different answers for particular statements. This is most notable when using expressions which contain side-effects. For example the outcome of the following two statements is undefined.

```
firstArray[i] = secondArray[i++];
```

The programmer must use parentheses to make intentions clear.

As well as considering the order of evaluation of a statement, the programmer must also consider the precedence of the operators used in the statement because these affect the way the compiler breaks down the statement into expressions. For example, the relative precedence of the addition and multiplication operators mean that the expression

```
a + b * c
```

is really treated as

```
a + ( b * c )
```

rather than

```
( a + b ) * c
```

The programmer is advised to make explicit use of parentheses to reduce any possible confusion about what may have been intended and what the compiler will assume by applying its precedence rules.

The programmer must always use parentheses around bitwise operators.

The use of the bitwise operators is not always intuitive because they appear to offer two different types of behavior. On the one hand they behave like arithmetic operators while on the other hand they behave like logical operators. The precedence rules may mean that a bitwise operator behaves as expected when used in one context, but not when used in another context. The safest course is to use parentheses with the bitwise operators so that the code is clear.

```
if (statusWord & PARTICULAR_STATUS_BIT)
```

lacks an explicit comparison so the programmer modifies it to what would initially appear to be an equivalent form:

```
if (statusWord & PARTICULAR_STATUS_BIT != 0)
```

Unfortunately, the programmer has just introduced an error into the code! The precedence rules mean that the bitwise-AND operator has a lower precedence than the != comparison operator and as a result PARTICULAR_STATUS_BIT is tested against zero and the outcome is then combined with the statusWord using the bitwise-AND. The programmer must use parentheses to restore the code to working order:

```
if ((statusWord & PARTICULAR_STATUS_BIT) != 0)
```

5.3 Use of white spaces

Do not use spaces around the "." and "->" operators or between a unary operators and their operands.

These operators are tightly coupled with their operands and the precedence rules mean that they are evaluated before the other operators. Adding spaces between the operator and the operand makes them appear to be more loosely coupled than they really are and this can lead to confusion.

Other operators should be surrounded by white space.

In contrast with the previous rule, other operators should be surrounded by white space in order to give additional visual separation of the operands in order to show that they are more loosely coupled. Note that parentheses may be used to group operands for evaluation purposes and white space may be omitted for "tightly-coupled" operands.

```
y = (a * x * x) + (b * x) + c; // or y = a*x*x + b*x + c;
```

However, when in doubt about operator precedence, use parentheses!

5.4 Memory allocation

To be done.

5.5 Error handling

To be done, Add something about Debug versus Release versus RelWithDebInfo.

5.6 The use of the pre-processor

The programmer should use #include system header files using <name.h> and user header files using "name.h"

There are two different reasons for using different #include semantics for system header files and user header files. The first is that the programmer can see from the use of angle brackets or double quotes whether the header file is a system header file or a user header file as this might not be obvious from the name alone.

The second reason is that on some systems the different `#include` semantics imply slightly different behavior when it comes to searching for the include file in the file system.

The `#include` line may not contain the full path to the header file

If the programmer specifies the full path to a header file then the source file will need to be modified when the header file is moved. Different operating systems also use different ways of specifying a full path and this will therefore need to be changed when porting the software from one system to another.

Use conditional compilation to select diagnostic or non-portable code.

Diagnostic code should be enclosed within `#ifdef/#endif` pairs so that it can be removed from the release version of the program while leaving it available for development purposes.

Non-portable or system-specific code should also be enclosed within suitable `#ifdef/#else/#endif` pairs. This allows different versions of code to be compiled without the overhead of a separate file for each system, and the configuration problem, which this may cause. Such conditional compilation can allow for an alternative version of the code, or for a series of different versions.

```
#if INTEGER_ARITHMETIC
/*
 * Faster, but less accurate version using integer
 arithmetic.
 */
#else
/*
 * Standard floating point version.
 */
#endif /*INTEGER_ARITHMETIC*/
#if UNIX
/* Unix specific code */
#elif DOS
/* DOS specific code */
#else
ERROR: This code only works on Unix and DOS
#endif
```

Pre-processor macros, which contain parameters or expressions, must be enclosed in parentheses.

The constants and macros provided by the pre-processor are not really part of the language and are simply substituted directly into the source code. Therefore, care should be taken when using expressions (even in constants!). The body of the macro should be enclosed in parentheses.

```
#define TWO 1+1 /* should be (1+1) */
six = 3 * TWO; /* six becomes 3*1+1 i.e. 4 */
```

If a macro contains any parameters, each instance of a parameter in both the macro declaration and in the macro body should be enclosed in parentheses. The parentheses around the parameter in the declaration protect the macro in the event that it is called with an expression which uses the comma operator which would cause confusion about the number of parameters. The parentheses around the parameters in the macro body protect the macro from conflicts of precedence if the parameter is an expression.

```
#define RECIPROCAL(x) (1/x) /* should be (1/(x)) */
```

```
half = RECIPROCAL(1+1); /* half becomes 1/1+1 i.e. 2 */
```

Macros should not be used with expressions containing side effects.

The macro hides the underlying expression.

The programmer may only see one expression (with possible side effect) when calling the macro, but underneath the macro body may use this expression several times. The order of evaluation of arguments before calling a function does not apply to a macro because it is not a function but merely a pre-processor convenience.

```
#define SQUARE(x) ((x)*(x)) /* nothing unusual */
z = SQUARE(y++); /* z becomes y++*y++ i.e. what? */
```

The pre-processor may not be used to redefine the language.

Programmers who are more comfortable using another programming language may be tempted to provide pre-processor macros, which map C and C++ constructs into some semblance of this other language.

```
#define IF "if"
#define THEN "{"
#define ELSE "} else {"
#define FI "}"
IF (x == 1)
THEN
statement(s);
ELSE
statement(s);
FI
```

This is expressly forbidden because the programmer is supposed to be using C or C++ and not some other language with which a future maintenance programmer may not be familiar. Certain constructs may not map directly from C into the other language and may therefore have restrictions on their use. Various support tools, such as syntax aware editors, will be unable to work with the macros.

6 Portability

6.1 Data abstraction

The programmer should use "problem domain" types rather than implementation types.

The programmer is encouraged to add a level of data abstraction to the code so that variables are expressed in terms which relate directly to the problem domain rather than to the underlying "computer science" or hardware implementation.

The use of "typedef" can also improve maintainability because only one place in the code needs to be changed if the range of values for that type must be changed. For example, the range of values may be changed by converting a variable, or set of variables, from "short" to "unsigned short".

Portability is improved because only one place in the code needs to be changed to make use of different hardware characteristics on different platforms.

```
typedef unsigned short AgeInYears_t;  
typedef float KmPerHour_t;
```

6.2 Representation

The programmer may not assume knowledge of the representation of data types in memory.

Different systems may make use of different representations of data types in memory. Such differences may include differences in word ordering, byte ordering within a word and even the ordering of bits within a byte. The programmer should therefore avoid any specific knowledge of an underlying representation when manipulating the data because what may be valid on one system may not hold true on another system.

The programmer may not assume that different data types have equivalent representations in memory.

This rule follows from the previous section and the rule above. If the different types specify different ranges of values, it is likely that they are represented differently in memory. The programmer may not assume that different types share a common representation in memory.

6.3 Pointers

The programmer may not assume knowledge of how different data types are aligned in memory.

Different hardware platforms impose different restrictions on the alignment of data types within memory.

The programmer may not assume that pointers to different data types are equivalent.

Some hardware platforms actually use different pointer representations for different data types. Therefore, assigning one pointer value for one type to a pointer variable for another type is not always guaranteed.

The only exception is the equivalence of void* pointers to pointers of other types.

The programmer may not mix pointer and integer arithmetic.

Pointers are not integers. The programmer must not treat pointers as integers. The programmer is prohibited from assigning integers (or integer values) to pointers and vice versa. If we assume the following declarations:

```
int intValue;
Thing_t thingArray[10];
Thing_t *thingPointer = thingArray; // i.e. &thingArray[0];
intValue = thingPointer; // DO NOT DO THIS IN REAL CODE!
intValue += 1; // increase integer value by 1
thingPointer += 1; // adjust pointer value to point to
// next thing, i.e. &thingArray[1]
```

At this point it is unlikely that `intValue` and `thingPointer` contain the same value because `intValue` has been incremented by 1 whereas `thingPointer` has been adjusted by the size of a `Thing_t` object (including possible adjustments for the alignment of `Thing_t` objects in memory).

7 Formatting

7.1 To Use Enums or Not to Use Enums

C allows constant variables, which should deprecate the use of enums as constants. Unfortunately, in most compilers constants take space. Some compilers will remove constants, but not all. Constants taking space precludes them from being used in tight memory environments like embedded systems. Workstation users should use constants and ignore the rest of this discussion.

In general, enums are preferred to *#define* as enums are understood by the debugger.

Be aware enums are not of a guaranteed size. So if you have a type that can take a known range of values and it is transported in a message you can't use an enum as the type. Use the correct integer size and use constants or *#define*. Casting between integers and enums is very error prone as you could cast a value not in the enum.

7.2 Pointer Variables

- place the * close to the variable name not pointer type

Example

```
char *name= NULL;
```

```
char *name, address;
```

7.3 Variable names on the stack

- use all lower case letters
- use '_' as the word separator.

Justification

- With this approach the scope of the variable is clear in the code.
- Now all variables look different and are identifiable in the code.

7.4 Use header file guards

Include files should protect against multiple inclusion through the use of macros that "guard" the files. Note that for C++ compatibility and interoperability reasons, do **not** use underscores '_' as the first or last character of a header guard (see below)

```
#ifndef sys_socket_h
#define sys_socket_h /* NOT _sys_socket_h_ */
#endif
```

8 Miscellaneous

8.1 Document Null Statements

Always document a null body for a for or while statement so that it is clear that the null body is intentional and not missing code.

```
while (*dest++ = *src++)
{
    ;
}
```

8.2 Use #if Not #ifdef

Use #if MACRO not #ifdef MACRO. Someone might write code like:

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

Someone else might compile the code with turned-of debug info like:
cc -c lurker.cc -DDEBUG=0

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

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

If you really need to test whether a symbol is defined or not, test it with the defined() construct, which allows you to add more things later to the conditional without editing text that's already in the program:

```
#if !defined(USER_NAME)
#define USER_NAME "john smith"
#endif
```

8.3 No Magic Numbers

A magic number is a bare naked number used in source code. It's magic because no-one has a clue what it means including the author inside 3 months. For example:

```
if (22 == foo) { start_thermo_nuclear_war(); }
else if (19 == foo) { refund_lotso_money(); }
else if (16 == foo) { infinite_loop(); }
else { cry_cause_im_lost(); }
```

In the above example what do 22 and 19 mean? If there was a number change or the numbers were just plain wrong how would you know? Instead of magic numbers use a real name that means something. You can use #define or constants or enums as names. Which one is a design choice. For example:

```
#define    PRESIDENT_WENT_CRAZY    (22)
const int WE_GOOFED= 19;
enum {
    THEY_DIDNT_PAY= 16
};

if        (PRESIDENT_WENT_CRAZY == foo) { start_thermo_nuclear_war();
}
else if (WE_GOOFED                == foo) { refund_lotso_money(); }
else if (THEY_DIDNT_PAY           == foo) { infinite_loop(); }
else                                     {
happy_days_i_know_why_im_here();}
```

Now isn't that better? The const and enum options are preferable because when debugging the debugger has enough information to display both the value and the label. The #define option just shows up as a number in the debugger which is very inconvenient. The const option has the downside of allocating memory. Only you know if this matters for your application.

8.4 Error Return Check Policy

- Check every system call for an error return, unless you know you wish to ignore errors. For example, *printf* returns an error code but rarely would you check for its return code. In which case you can cast the return to **(void)** if you really care.
- Include the system error text for every system error message.
- Check every call to malloc or realloc unless you know your versions of these calls do the right thing. You might want to have your own wrapper for these calls, including new, so you can do the right thing always and developers don't have to make memory checks everywhere.