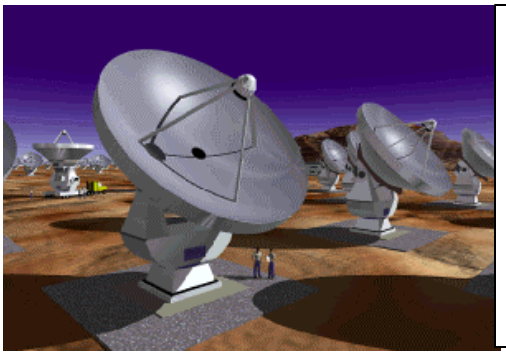

Atacama Large Millimeter Array

ALMA-SW-0009

Revision: 4

2001-08-21

Software Standard

Alan Bridger

C Coding Standards

Software Standard

Alan Bridger (ab@roe.ac.uk)
UK Astronomy Technology Centre
Mick Brooks, Jim Pisano
National Radio Astronomy Observatory

Keywords: programming standards, C, language

Author Signature:	Date:
Approved by: Gianni Raffi, Brian Glendenning Institute: ESO, NRAO	Signature: Date:
Released by: Institute:	Signature: Date:

Change Record

REVISION	DATE	AUTHOR	SECTIONS/PAGES AFFECTED
	REMARKS		
1	2000-10-04	Alan Bridger	All
	Moved to doc. Std. Removed some C++ refs. Improved structure.		
2	2001-02-28	Alan Bridger	All
	Responded to comments from review. Doc # inserted.		
3	2001-05-29	Alan Bridger	Sec. 7, Appendices A & B.
	Changes to example copyright/license following comments		
4	2001-08-21	Michele Zamparelli	Sec 8,9. Appendix A
	Added doxygen section, modified Appendix A to make it doxygen compliant		

Table of Contents

1 Preamble

1.1 Scope

This document describes the C coding guidelines agreed upon by the ALMA software engineering team. These guidelines are expected to be adhered to in all future C software development for the ALMA project. Existing code will not be required to be rewritten to adhere to these guidelines, but it is suggested that reasonable efforts are made to adhere to the new guidelines when working on existing code.

1.2 Introduction

This document was written with the following goals in mind:

1. Code should be robust and error free.
2. Code should be easy to use and understand.
3. Code should be easy to maintain.

Each section of this document contains recommended **standards** in that area, along with a further list of **guidelines**. A **standard** is required to be followed in all future development. If a programmer must violate a **standard**, then the reason for doing so must be strong and clearly documented via code comments. **Guidelines** are just that. Individual programmers are encouraged to program in a style which is consistent with the rest of the programming team, but differing code styles are acceptable.

These standards recognize that individual programmers have the right to make judgments about how best to achieve the goal of code clarity. When working on others' code, please respect and continue existing styles which, although they may differ from your own, meet these guidelines. This in turn leads to what is perhaps the most important aspect of coding style: consistency. Try, as much as possible, to use the same rules throughout the entire program or module.

Section 10 lists a number of relevant texts. [1] is essential reading for all C programmers, [2] is good C style guide and, though aimed more at C++, [3] provides a number of very useful good programming tips. [4] and [5] are programming standards from other organisations that were used as reference material for this document.

2 General

Standards

- All code should be ANSI standard and should compile without warning under at least its principal compiler. Any warnings that cannot be eliminated should be commented in the code.

3 Naming Conventions

3.1 General

Standards

- Clear and informative names are one of the best tools for creating easily understandable code. The name of any identifier should succinctly describe the purpose of that identifier.
- Avoid abstract names (in a global context) that are likely to be reused by other parts of the system.

3.2 Function Names

Standards

- Function names must identify as far as possible the action performed or the information provided by the function.
- Function names should be prefixed with the name of the module to which they belong. “Obvious” names (e.g. “util”) for modules should be avoided. Module names should be kept short.

Guidelines

- Function names should normally be formed from two parts: an action (verb) and an object (noun) of the action. Exceptions are query functions where the second part is not a noun, but the name should form a “question”. Each word forming the function name is capitalised, except for the first which should be lowercase (this will be the module name). Examples of acceptable function names are `chatCloseSession`, `gpsSetTime` and `driveIsActive`.

3.3 Variable Names

Standards

- The type and purpose of each variable should be evident within the code in which it is used. E.g. the reader will expect `counter` to be an `int`, `motorSet` might be a `BOOLEAN` or an array representing a set – context will usually clarify this. And while the type of `sessionId` might not be obvious it will be expected to be an identifier or handle that labels some sort of session.
- Names should be formed from composite words, delimited by upper case letters, with no underscores allowed (except for appending pointer identifiers etc.) Underscores should be used as delimiters in macro names.

Guidelines

- Variable names should be short, but meaningful.

- Local variables should be named with the content of the variable. Each word except the first one is capitalised. For example, `counter` and `sessionId` are acceptable variable names.

- Global variables should be similarly named except that each word is capitalised. For example, `ExposureTime` is acceptable.

- Pointer variables should be identified, preferably by appending "`_p`", for example:

```
struct s * meaningfulName_p
```

3.4 Other Names

Standards

- Macros, enumeration constants and global constant and `typedef` names should be stylistically distinguished from function, and variable names.

- Types (struct types, etc.) should be given meaningful names and begin with an uppercase letter in order to distinguish them from variables.

Guidelines

- Constant names should be in uppercase, with multiple words separated by an underscore. Valid examples are `MAX_COLUMN` or `THOUSAND`.

- Macros, with or without parameters, should also be in all upper case.

- Enumeration constants and global `typedef` names should be in all uppercase with individual words separated by underscores, e.g. `DATA_VALID`.

3.5 File Names

Standards

- File names should represent the content or role of the file.

- Header file names should have the extension "**.h**".

- C Implementation (source) file names should have the extension "**.c**".

- File names should contain only alphanumeric characters, "_" (underscore), "+" (plus sign), "-" (minus sign), or "**.**" (period). Meta-characters and spaces must be avoided. Also file names that only vary by case are not permitted.

- When used with the ESO CMM filenames should contain the module name as a prefix.

4 Variables, Operators and Expressions

Standards

- Global variables should be avoided unless they are really necessary, and good reasons should be given for their use.
- Let the compiler work out the length of objects: use `sizeof` rather than declaring structure lengths explicitly.
- Use a cast explicitly when converting unusual types, otherwise in general allow the compiler to do the casting.
- Avoid machine dependent fill methods. Do not apply right shift (>>) or left shift (<<) operators to signed operands, and do not use shifts to perform division or multiplication. Clearly noted exceptions are allowed.
- Do not write code that depends on the byte order of a particular architecture, nor on particular word alignment boundaries.
- Un-initialised automatic variables must never be used. This is detected by lint.
- Pointer assignment shall be between pointers of the same type (`void *` is an exception , though should be used only with good reason). Avoid treating pointers as integers or vice versa.
- To check the validity of a pointer, compare it to a typecast `NULL`. Do not use the integer 0. Note that `stdio.h` must be included to define `NULL`.

Guidelines

- Beware of side effects in macro parameters. Don't use increment or decrement operators on macro arguments; if the macro references the argument more than once the increment or decrement will be repeated.
- To improve clarity use parentheses even when not required. Using parentheses also helps avoid errors caused by misunderstood operator precedence.
- Provide an in line comment for each variable declaration (except for obvious loop counters).

5 Functions

Standards

- ANSI-C style should be used for function declaration, with parameter types in the function declaration and a return type declared. For example use:

```
int power (int base, int n)
    {
    .......
```

```
    }
```

instead of:

```
power (base, n)
int base, n;
    {
    .......
    }
```

- Assumptions should not be made about the order of parameter evaluation. Always explicitly code the desired order of evaluation of sub-expressions. For example:

```
func(x * i, y / i, i++)
```

may be evaluated from right to left or left to right with different compilers and different CPUs.

- When unusual data types are mixed across function boundaries use explicit type casts to make it clear what is happening.
- If no return value is required for a function, it should be declared as a `void` function.
- If a function has an empty argument list this should be clearly indicated using `(void)`, even though many compilers accept `()`.

6 Statements and Control Flow

Standards

- Always provide a default for switch statements.
- Always use braces to delimit the block in an if statement, even if there is no else block. A complete `if (condition) statement` that fits on one line is an exception.
- Always use braces to delimit the block in while and do … while statements.
- Avoid the use of `goto` statements. If their use is necessary, clearly document the reasons.

Guidelines

- For loops should never have an additional increment or decrement of the loop counter inside the loop.
- Break each case of a switch statement to avoid falling through to the next case. The only exception is multiple case labels.

- In a while loop, the increment or decrement should be at the top or bottom of the block. Do not increment or decrement loop counter variables in the middle of a block. This makes it harder to find the loop counter.

7 Code Commenting

Standards

- All files, both header and source, must begin with the standard header information, which contains the file identification information, a one line description of the module, and the copyright notice. Different copyright statements may be required for different sites.
- The header may also contain a longer description of the purpose of the module and any other pertinent information about the module.
- A change log for each module must be maintained in a manner appropriate to the development environment. Exactly how this is done is still to be determined by the development environment chosen.
- Comments intended for documentation should use the doxygen style. A template for the ALMA project may be found in the Appendix A. This template includes a simple function example.
- Block style and in-line comments are both acceptable.
- Block style comments should be preceded by an empty line and have the same indentation as the section of code to which they refer. Block style comments should appear at the beginning of the relevant segment of code.

Block Style:

```
/*
 * ..............
 * ....................
 * ........................
 */
```

- Brief comments on the same line as the statement that they describe are appropriate for the in-line commenting style. There should be at least 4 spaces between the code and the start of the comment.

In-Line Comments:

```
..........              /* ......... */
..........              /* ......... */
..........              /* ......... */
```

- Do not break in-line comments into multiple lines, thus:

```
..............    /* ..........      AVOID THIS!
                   .............. */
```

- Use in-line comments to document variable usage and other small comments. Block style comments are preferable for describing computation processes and program flow.

Guidelines

- Use the **Id** RCS identifier for all RCS-based source-code control tools. The **Id** identifier expands to **$Id: filename revision date time author state $.** Note that when used with ESO's CMM system the identifier must be preceded by **@(#).**
- Use a "ToDo" section in the header comments to indicate those items that remain to be done.

7.1 Header Files

Standards

- Header files should be used as a means of interface specification for a source module. For example, for a module `motion_control.c`, the external interface containing data declarations, function prototypes and all information necessary to use the module should be declared in `motion_control.h`.
- For each function prototype in the header file, the following information should be given in comments:

  What the parameters are, how they are used, and any preconditions that must be established for the parameters.

  What the function is going to do to the parameters.

  What the results/return values are in all the different cases possible for the function.

- To avoid nested includes, use #define wrappers as follows:

```
#ifndef MOTION_CONTROL_H
#define MOTION_CONTROL_H

......
......

#endif /* MOTION_CONTROL_H */
```

Guideline

– If the header might be used by C++ code then the following #ifdef wrapper will allow the same file to be used by both:

```
#ifdef _cplusplus
extern “C” {
#endif
......
......

#ifdef _cplusplus
}
#endif
```

A sample header file example is included in Appendix A.

7.2 Source Files

Standards

- All comments in source code files must be up-to-date at all times during that code's lifetime.
- Code comments should be used to give an English language synopsis of a section of code, to outline steps of an algorithm, or to clarify a piece of code when it is not immediately obvious what was done or why it was done. In no case should the code comments just parrot the code.
- Use consistent nested indentation throughout your code. When working in existing code, respect the indentation style currently in use. At a minimum, indentation should be consistent within a function.
- A sample implementation file template is included in Appendix B.

Guidelines

- Code comments which apply to a block of code (either a loop or branch construct, or a grouping of statements), should be immediately above the block and indented to the same level as the code.
- Code comments which apply to a single statement may be immediately above or to the right of the statement.
- Use a line of dashes ‘—‘ to visually block off function definitions in the source file. This allows easy identification of where the functions start.

8 Readability

Standards

- Write only one statement per line.

Guidelines

- Use blanks around all binary operators except "." and "->".

- In compound statements, put braces ({}) in a separate line and aligned with indented statements. For example:

```
{
......;
......;
}
```

- If a compound statement is longer than 20 lines, the closing brace should have an inline comment to indicate which block it delimits.

```
  .
  .
}    /* ...... */
```

- Use a blank after commas (arguments list, values, etc), colon, semicolon and control flow keywords:

```
procedure(arg1, arg2);
for (i = 0, j = strlen(s) – 1; i < j; i++, j--)
if (...)
while (...)
```

- Do not use blanks between an identifier and any of "(", ")", "[" or "]":

```
procedure(arg1, arg2)
a[I] = 1;
z[I] = a + (sin(x) + cos(b[i]))
printf("%d", (a + b))
```

- Line up continuation lines with the part of the preceding line they continue:

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

if ((a == b) &&
    (c == d))
     ....
```

```
printf("%d %s",
       (a + b),
       "abcd")
```

- Indents should be 4 characters. If this causes difficulty with code which is deeply nested, the number of spaces may be 2 or 3; the indentation should be kept constant within each file, however.

- Source code lines should be kept to less than 80 characters in length.

- Object declarations should be visually aligned as it is easier to scan a list of objects when names and types appear in the same column. However, this guideline should be used with some common sense. Examples:

```
struct FL_ENTRY * fl_p;
struct FLDSYM   * sym_p;
char            * wsbuf;
int               length;
```

- The following templates provide examples of how to indent the most commonly used code structures. Note that braces on a new line are the style here, but braces on the same line as the opening statement are also acceptable, but remember that consistency is always a goal.

```
typedef struct
    {
        ..... ......;
        ..... ......;
        ..... ......;
    } ......;

if (....)
    {
    ......;
    ......;
    }
else
    {
    ......;
    ......;
    }

if (....)
    {
    ......;
    ......;
    }
else if (....)
    {
    ......;
    ......;
    }
else if (....)
```

```
        {
        ......;
        ......;
        }

    switch (....)
        {
        case ...:
            ......;
            ......;
            break;

        case ...:
            ......;
            ......;
            break;

       .
       .

        default:
            ......;
            ......;
            break;
        }

    while (....)
        {
        ......;
        ......;
        }

    for (...; ...; ...)
        {
        ......;
        ......;
        }

    do
        {
        ......;
        ......;
        }
    while (....);
```

9 Doxygen Use

The ALMA standard code documentation tool is doxygen. In this section a suggested standard doxygen usage is presented. It is not intended that this be a doxygen primer - the documentation for doxygen is easily available (see [6]) and is much better for that. Note that the latest version of doxygen is 1.2.8.1

In very brief terms doxygen uses special comment delimiters (/**…*/) to allow the programmer to add his or her own documentation comments. In addition doxygen allows

developers to add own comment tags, for instance for binding together logically grouped functions.

In addition special tokens, similar to Javadoc ones, allow the specification of particular attributes such as author, version, etc.

A suggested example ALMA usage of doxygen may be found in the appendices, including examples of some doxygen features.

Notice that for doxygen to use the documentation, the latter must begin with /**, not simply /*. The normal C-style comment (/* …. */) may be used for comments related to technical details of the implementation, which do not necessarily have to be included in the published documentation. A few points worth mentioning are:

- Use @see (where appropriate) for cross-reference. Author and version are stored in the code repository.
- Use @return, @pre, @post, and @param (where appropriate) for C functions.
- Use a standard configuration file to store doxygen options.

A preliminary doxygen configuration file for ALMA has been added to the doxygen module in the ESO CMM. It should be noted that this is a suggested doxygen usage and that no doubt it will solicit comments.

10 References

1. The C Programming Language (second edition), Kernighan, B. and Ritchie, D., Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1988.
2. The Elements of C Programming Style, Ranade, J. and Nash A., , McGraw-Hill, 1993.
3. Writing Solid Code, Maguire, S., Microsoft Press, 1993.
4. SEL-94-003, 1994, The C Style Guide, NASA Software Engineering Laboratory.
5. SGP/4.2, 1994, Starlink C Programming Standard, Starlink Project, Rutherford Appleton Laboratory, Charles, A. and Murray, J.
6. Doxygen, van Heesh, D., http://www.doxygen.org, 2000.

Appendix A. Header File Example

```
#ifndef FOO_H
#define FOO_H

/*  @(#) $Id$
 *
 * [Insert copyright statement appropriate to author(s).]
```

```
 *
 * Produced for the ALMA project
 *
 * This library is free software; you can redistribute it
 * and/or modify it it under the terms of the GNU Library
 * General Public License as published by the Free Software
 * Foundation; either version 2 of the License, or (at your
 * option) any later version.
 *
 * This library is distributed in the hope that it will be
 * useful but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.
 * See the GNU Library General Public License for more
 * details.
 * You should have received a copy of the GNU Library
 * General Public License along with this library; if not,
 * write to the Free Software Foundation, Inc., 675
 * Massachusetts Ave, Cambridge, MA, 02139, USA.
 * Correspondence concerning ALMA should be addressed as
 * follows:
 *        Internet email: alma-sw-admin@nrao.edu
 */

/* Summary
 * A sample foo module, serves also as doxygen
 * documentation example
 */

/* Synopsis
 * The foo module doesn't do much here, but hold a count to
 * get & set.
 * It is used by ??? to blah, blah.
 */

/* Usage Example
 *    int myCount;
 *    fooInit();
 *    fooSetCount(100);
 *    myCount = fooGetCount();
 */

/* Todo
 *   a) Not much to do
 *   b) find a better example for the coding standards doc
 *   c) start involving Bar and not only Foo
 */

/*
 * Prototypes
 */

/** @name Counter functions  */
/*@{ */

/** initialise the foo  */
void fooInit();
/** Set the counter */
void fooSetCount(int newCount);
/** Get current value of counter */
int  fooGetCount();
```

```
/*@} */

/** @name Transmogriphier  functions */

/*@{ */
/** Calvin's transmogriphier must be loaded
 * @param profile: 0 for Calvin, 1 for Hobbes
*/
void transmogriphierLoad(int profile);

/** the transmogriphier must be unloaded after usage  */
int transmogriphierUnload(void);

/** this activates the transmogriphier
 @return code indicating success (0) or failure ( non zero) */
void transmogriphierStart(void);
/*@} */

#endif /* FOO_H */
```

Appendix B. Implementation File Template

```
/* @(#) $Id$
 *
 * [Insert copyright and license statement as presented in
Appendix A here]
 *
 */

#include “foo.h” /* Prototype declarations and interface */

/*
 * Module level variables
 */
static int FooCount;    /* The current count */

/*------------------------------------------------------------
 */
void fooInit()
{
    FooCount = 0;    /* Reset the counter */
}

/*------------------------------------------------------------
 */
void fooSetCount(int newCount)
{
    FooCount = newCount;    /* Set new counter value */
}

/*------------------------------------------------------------
```

```
 */
int  fooGetCount()
{
    return FooCount;
}
```