Project Notebook Storage Location: Drew Technologies Corporate Library

# A Coding Standard for the C Programming Language

Drew Technologies Inc.
Copyright © 1996,1999

**Contact:**

DrewTechnologies, Inc.
41 Enterprise Drive
Ann Arbor, MI 48103
(734)623-8080
support@drewtech.com

## Revision History

| DATE | COMMENTS |
|---|---|
| 04-Sep-96 | Final Draft |
| 22-Oct-99 | English corrections, address update |
| | |
| | |
| | |
| | |

# TABLE OF CONTENTS

# 1. Introduction

This document is proposed as the coding standard for all C programs written by Drew Technologies that are not required to follow an external coding standard specified by the customer.

## *1.1 Scope*

This document is a collection of rules and guidelines that include issues related not only to the C language, but also software engineering and program portability in general.

The scope of this document is such that it is most effective when applied at the earliest stages of coding. Most if not all of the concepts and practices proposed in this document can be applied to the maintenance of code that does not conform to this standard.

## *1.2 References*

Harbison, S.P.,1991, *C A Reference Manual*, Prentice Hall Inc.
ANSI,1989, *American National Standard X3.159-1989*
Oualline, S.,1992, *C Elements of Style*, M&T Books Inc.
Libes, D.1993, *Obfuscated C and Other Mysteries*, John Wiley & Sons Inc.

# 2. Process Problem

Without a coding standard in place, there is no framework through which we can:

- Enforce good structured programming practices
- Provide a reference for code walkthroughs and inspections
- Make code easier to understand and maintain

## 3. Root Causes and Effects

It is possible to write hopelessly obscure code in any language. The C programming language has some very powerful features, which can be misused to make programs hard to understand, and do things that are widely recognized as bad software engineering. Any process put into place must minimize the likelihood that this will occur.

The effects of this problem can be broadly separated into the following categories:

- Poor maintainability
- Poor portability
- Poor Reusability

### *3.1 Poor Maintainability*

Understanding is at the heart of maintainability. If a piece of code is difficult to understand, it is difficult to maintain. Factors that contribute to making code hard to understand include the following:

- No discernible naming conventions for identifiers
- No discernible indenting conventions used
- Poor use of parenthesis
- Poor use of  comments
- Overuse of conditional preprocessor directives
- Poor program structure
    - Poor module layout
    - Excessively long functions
    - Excessively optimized code
- Inclusion of dead code
- Overuse of nested includes
- Lack of module header
- Lack of function header
- Poor use of white space

### *3.2 Poor Portability / Reusability*

All software developers should be concerned with portability and reusability issues as they write code. Factors contributing to a lack of portability and reusability include the following:

- Compiler dependence
- Processor dependence
- Hardware platform dependence
- Lack of low level data abstraction
- Lack of unit test code

This lack of portability prevents us from:

- Being able to unit test code on non-target platforms (SUN or IBM-PC)
- Reacting quickly to changes in the market (New chips and compilers)
- Using CASE tools to evaluate/manage our code

### *3.3 Arguments for Efficiency*

Arguments for efficiency are often used in defense of code that is not maintainable, portable or reusable. Efficiency must be addressed in manner consistent with good engineering practices.

# 4. Recommendations

## *4.1 General Naming Conventions*

Naming conventions serve two purposes. First, they provide consistency throughout the program. Second, since identifiers are by far the bulk of any program, they should have an aesthetic quality that makes them easy to read.

Words separated by underscores satisfy both of these requirements. Underscores can easily be used in a consistent manner to separate the individual words in an identifier. The use of the underscore evolved from a desire to use a character that resembles a space, as closely as possible. People are comfortable with the aesthetics of using underscores between words because underscores resemble spaces.

The ANSI C spec requires that all compilers permit a minimum of 31 significant characters in identifiers. Most compiler vendors meet or exceed this limitation. While every attempt should be made to impart meaning to identifiers by using long names, KNOW YOUR COMPILER! Due to linker or system limitations extern or global data may be limited to something less than a full 31 characters.

### *4.1.1 Naming Variables*

The first and foremost requirement of a variable name should be that it convey at least some meaningful information about the how the variable being declared is going to be used. Examples of bad variable declarations include:

```
int i;    /* Index into array */

int x;    /* Sum of last 10 samples */

int *ts;  /* Pointer to timestamp */
```

In the preceding declarations, comments are used to help explain to the reader how these variables will be used in the program. These comments would not be required if the variables were changed to:

```
int index;

int sum_of_last_10;

int *ptr_to_timestamp;
```

The challenge is to pick variable names that help to make your code self documenting without the use of comments. Comments in the declaration do not necessarily “stay with” the reader as he works his way through the rest of the function. A good variable name will continue to impart information about the program's function to the reader each time it is used.

### *4.1.2 Naming Functions*

Function names should consist of a verb noun pair that when combined provide some meaningful information about what the function does. Consider the following examples:

```
PRIVATE int find_next_prime(int number);

PUBLIC int calculate_checksum( char *data, int length);

PUBLIC int queue_data(int queue_id, int data_to_queue);

PRIVATE void start_motor(int motor);
```

### *4.1.3 Naming Structures & Structure Members*

The naming of structures and structure members should follow the same basic rules that apply to variables and functions. When naming structure members, one temptation that should be avoided is the inclusion of the structure name in the member name.

### *4.1.4 Naming Constants Macros & Typedefs*

All constants, typedefs and macro names should be in upper case only. Without this distinction it becomes difficult to determine if you are:

- Invoking a macro or making a function call
- Using a constant or a variable

This distinction can be very important if your macros have side effects due to things like auto increment or auto decrement.

### *4.2 Indentation*

The purpose of indentation is to give the programmer an accurate picture of the control flow of a function at a glance. A programmer should not have to struggle or rely on comments to find the beginning and ending of a block. Curly braces should be use with every control structure even if there is only one statement to a block. This practice helps balance the amount of white space and prevents orphan statements due to indentation errors.

Tabs should be used as indentation characters. The use of spaces, while offering some benefits, is subject to errors when blocks are nested two or three levels deep. Spaces are also tedious to type.

A tab stop of four is a good trade off between adequate white space and code density. Programmers should attempt to limit there line length to 80 characters. If indentation of control structures is such that you are running out of space, then this is probably an indication of a lack of sufficient functional decomposition (See Module Layout).

#### *4.2.1 Indentation of Basic Flow Control Statements*

Give example of: if/for/while/do while

##### *4.2.1.1 Simple If Statement*

```
if ( index == END_OF_LIST )
{
    /*
     *   Do something
     */
}
else
{
    /*
     *   Do something else
     */
}
```

#### *4.2.1.2 Compound If Statement*

The compound "if" statement has been the subject of much controversy. The most common method for indenting compound "if" statements is:

```
if ( a == ONE_THING )
     do_something();
else if ( a == ANOTHER_THING )
     do_something_else();
else if ( a == YET_ANOTHER )
     do_yet_another
else
     do_last_thing();
```

This indentation style shows all of the do_xxxx functions at the same indentation level. With every other example of indentation, this implies that they are not dependent on each other. This is clearly not the case as demonstrated by the reformatted version below:

```
if ( a == ONE_THING )
{
    do_something
}
else
{
    if ( a == ANOTHER_THING )
    {
         do_something_else
    }
    else
    {
         if ( a == YET_ANOTHER )
         {
              do_yet_another()
         }
         else
         {
              do_last_thing();
         }
    }
}
```

The revised method clearly shows at a glance that the function `do_last_thing` is only called from within the else clause of the "`if ( a == ONE_THING )`" statement. This observation is not as easily made with the original example. This determination is even more difficult to make if there is a significant amount of code for each of the conditions that pushes the code across several pages.

The original example also allows the complexity of the code to grow to high level of complexity without the programmer being made aware of it. This missing check on the functions complexity can lead to compound conditional statements that are well beyond the reasoning power of any programmer. The modified example limits complexity by running programmer out of space on the line before the code becomes to complex.

#### *4.2.1.3 For Statement*

The basic “for” loop is elegant in its simplicity, it should not be corrupted by stuffing things that belong in the body of the loop.

All “for” loops should resemble the following:

```
for ( index = 0; index < NUM_ITEMS; index++)
{
     list[index] = NULL;
}
```

#### *4.2.1.4 While Statement*

Like the “for” loop, the beauty of the “while” loop lies in its simplicity. “While” loops should not be stuffed with extraneous statements except for “priming read” type assignments. The basic content of the “for” loop should be a logical compare. The following shows a correctly formatted “while” loop:

```
while ( ch = getchar() != NULL )
{
     do_something(ch);
}
```

When complex looping conditions are required, parenthesis should be used to telegraph the intent of the programmer. The follow shows the correct use of parenthesis:

```
while( (a != b) && ((c == d) || (e == f)) )
{
     do_something();
}
```

### *4.2.2 Indentation of Switch Statements*

The switch statement is the most complex control statement in C. A consistent approach to using switch statements must be achieved if code is to be reliable and maintainable.

The first rule of switch statements is brevity. Switch statements that go on for page after page with individual cases that span pages is a sure sign that the function needs further decomposition. While there is no effective way to limit the number of cases in a switch, the number of statements in each case should be limited to seven lines or less counting the break and a blank line after the break. In other words, if the function of the case can not be handled in five lines, the case should probably be a function.

The following is an example of a correctly coded switch statement:

```
switch(action)
{
     case INVALID_START:
          state = IDLE;
          break;

     case VALID_START:
          state = RX_MSG;
          rx_buffer.depth = 0;
          break;

     case VALID_DATA:
          state = RX_MSG;
          enqueue_rx(rx_char);
          break;

     case ESC_FOUND:
          state = ESC;
          break;

     case INVALID_ESCAPE:
          state = IDLE;
          rx_buffer.depth = 0;
          enqueue_loader(BAD_MESSAGE);
          break;

     case ESCAPE_SYNC:
          state = RX_MSG;
          enqueue_rx(SYNC);
          break;

     case ESCAPE_ESCAPE:
          state = RX_MSG;
          enqueue_rx(ESCAPE);
          break;

     default:
          enqueue_fault(INVALID_INPUT);
```

```
            break;
    }
```

### *4.2.3 Indentation of Structures and Unions*

In order to keep pointer casts as simple as possible, structures and unions should always be declared with a typedef. The following indentation standard should be applied to structures and unions:

```
typedef struct
{
     U32 element_a;
     U32 element_b;
     U16 element_c;
} DEMO_STRUCTURE;
```

Notice that the structure tag is not used in the example above. The only time a structure tag should be used is when a structure or union will include a reference to itself. For example, if the structure above was going to be used on a doubly linked list, the structure would look like:

```
typedef struct demo_struct
{
     U32 element_a;
     U32 element_b;
     U16 element_c;
     struct demo_struct *previous;
     struct demo_struct *next;
} DEMO_STRUCTURE;
```

Notice that the structure tag should always be a lower case version of the typedef name.

Compound structures or structures that contain unions should be indented in the following manner:

```
typedef enum { DEMO_NAME, DEMO_VALUE, DEMO_COUNT } DEMO_TAG;

typedef struct
{
     DEMO_TAG tag;
     union
     {
          U32  value;
          U32  count;
          U8   name[DEMO_NAME_SIZE];
     }
} DEMO_UNION;
```

The use of enumerated types for union tags provides a measure of protection, at no additional cost, when using tagged unions. Only enumerated types should be used as union tags.

### *4.3 Parenthesis*

The C language provides the programmer with a rich set of operators. It also burdens him with the task of remembering 15 precedence rules. Consider the following:

```
result = x << 1 + 5;
```

Which operator is performed first? Better yet, which operator did the programmer intend to be performed first? The answer to the first question is easily determined by finding that binary operator precedence chart in K&R. The second question is harder to determine without asking the programmer who wrote the code, or completely reverse engineering the function that this line appears in to fully understand the task that it is trying to accomplish.

In order to save time and prevent frustration, programmers should use parenthesis to help show intent. The previous example could be transformed into:

```
result = x << (1 +5);
```

or

```
result = (x << 1) + 5;
```

These rules also apply to Boolean operations such as:

```
while ( (a != b) && (( c!= d)  || ( e != f)) )
{
     do_something();
}
```

### *4.4 Comments*

Comments are a great tool. Unfortunately they are often used to compensate for bad programming practices that otherwise would be unacceptable. Comments should be reserved to explain the programmer's intent. Not to attempt to make poorly written code easier to understand.

Comments should convey useful information about the program and not attempt to teach C to the programmer reading the code. The following are examples of comments that are essentially a waste of keystrokes.

```
x = x * x;  /* Square x */
y = y -5;  /* subtract 5 from y */
```

These comments are explaining the mechanics of what is being done not the intent. Put another way, they are explaining the "how" and not the "why".

The following example shows the proper use of comments:

```
/*
 * Find the first data item in the list that is greater than the
 * item we have so that we can insert it into the list in order.
 */
cursor = head_of_list;
while( (cursor->data < item_to_insert) && (cursor != NULL) )
{
     cursor = cursor->next;
}
```

### *4.5 Preprocessor Directives*

A large number of conditional preprocessor directives (#if, #ifdef, #else, #ifndef, etc.) can make code that is not readable or maintainable.

Code that is unreadable due to complex (often nested) conditional preprocessor directives is usually the result of code that has been ported several times. When #ifdefs and #ifs are nested, it can be very difficult to determine whether a statement is used or not unless you examine many preceding lines and simulate the execution of the preprocessor.

### *4.6 Program Structure*

A module should be structured in a way that makes it easy to read and understand. The sections that follow detail a framework that makes code easier to understand and maintain.

### *4.6.1 Module Layout*

Before the issue of module layout is addressed, a review of the fundamentals of modular design are in order to help clarify the requirements that need to be satisfied in order to achieve effective module layout.

The fundamental concept that drives the division of a program into multiple modules is that most problems can be decomposed into a series of simple tasks. Modular code is the end result of applying the "Divide and Conquer" paradigm until the problem can be solved by implementing a number of individual functions.

The coding standard helps programmers make the determination of when this process of "Divide and Conquer" is complete by establishing a limit on the maximum length that a function should have. Functions should be limited to one to three pages. Large control blocks should not span more than one page.

Functional decomposition progresses from the general to the specific. Functions within modules should also progress from the general to the specific. A module should contain only one major function and any subordinate or worker functions that are local to that module. Any worker or subordinate function that is common to several modules should be placed into its own module.[1]

All worker or subordinate functions should be declared static or PRIVATE. The major function within a module should be declared PUBLIC. All data local to the module should be declared static or PRIVATE. Data that is shared between multiple modules should be declared in the module that initializes it. There should be a header file for each module in the system. This header file should contain the function prototype of the major function as well as any external declarations for data that will be shared between modules.

The module should also contain unit test functions that can be conditionally compiled when a module is undergoing unit test. These unit test functions should be placed at the end of the module.

Appendix A contains examples of both modules and module headers.

[1] This is actually a defect in the design that is detected while coding. A function common to several modules should have been identified as a separate module during the functional decomposition.

### *4.7 Dead Code*

Dead code often results from programmers leaving old code around in functions as a sort of ad-hoc CM or archive of old functions that have been rewritten or otherwise replaced. This sort of historical archiving is a CM function and should be addressed with CM tools like SCCS/RCS.

Nothing is more frustrating than spending time looking at and trying to understand a function only to find out latter that this function is no longer called. Don’t leave dead code in modules.

### *4.8 Include Files*

If a system has been correctly decomposed into subsystems and subsystems have been correctly decomposed further into modules and if header files have been produced that contain interfaces to each module then only the subsystem header files should include header files. Consider the following example:

> Assume that we have a system composed of the following:
>
> - buffer management subsystem
>   - heap module
>   - partition module
> - serial device subsystem
>   - serial device driver module
> - user interface subsystem
>   - user interface module
>
> The following header files would be required:
>
> - buffer_user.h
> - serial_user.h
>
> Since the only users of the "user interface subsystem" are humans, there is no need for a public header file for the user interface.
>
> The header file for the buffer management subsystem (buffer_user.h) would include the following:
>
> - heap_user.h
> - partition_user.h
>
> The header file for the serial device subsystem (serial_user.h) would not include any header files. Since the serial subsystem is composed of only one module.
>
> Since the user interface is a user of both the serial device subsystem and the serial device subsystem, the user interface code modules would include both buffer_user.h and serial_user.h.
>
> As the serial device driver is a user of the buffer management subsystem, it would include buffer_user.h.

The tradeoff being made is one that potentially could lead to large numbers of headers being included by modules that belong to subsystems that interface to many different subsystems. This tradeoff is not unreasonable given that during the design phase the subsystem headers are probably the best reference to consult prior to writing any code.

Due to the fact that the inclusion of a header for a specific subsystem implies an interface to the subsystem, an attempt should be made to prevent unused/unrelated header files from being included.

### *4.9 Module Header*

The following header is for use with systems that use RCS. It uses various RCS keywords to convey information to about the files current version and modification history. If RCS is not available or if the project dictates the use of another source code control system the following items should be replaced with equivalents. If no equivalents exist, the information should be updated by hand to resemble the RCS equivalents.

```
/*   $Id$ */
/*************************************/ /*************************************/
/*     CONFIDENTIAL PROPRIETARY      */ /*    CONFIDENTIAL PROPRIETARY       */
/*      UNPUBLISHED SOURCE CODE      */ /*     UNPUBLISHED SOURCE CODE       */
/* Copyright 1996, Drew Technologies */ /* Copyright 1996, Drew Technologies */
/*       ALL RIGHTS RESERVED         */ /*       ALL RIGHTS RESERVED         */
/*************************************/ /*************************************/
/*
**
** MODULE AUTHOR(S):
**   Michael Drew
**
** MODULE TITLE:
**
** MODULE FUNCTION:
**   This is module functions as a demonstration header.
**   Each module should begin with one.
**
** History
 * $Log$
*****************************************************************************
*/
```

The following are descriptions of the RCS keywords used in the header:

- $Id$
  A standard header containing the filename of the RCS file, the revision number, the date and time, the author, the state, and the locker (if locked).

- $Log$
  The log message supplied during checkin, preceded by a header containing the RCS filename, the revision number, the author, and the date and time. Existing log messages are not replaced. Instead, the new log message is inserted after $Log:...$. This is useful for accumulating a complete change log in a source file.

### 4.10 Function Header

The function header need not contain revision information. Revision information should kept at the top of the file in the module header where it is immediately visible to someone inspecting the code for changes. The following function header should be found before each function.

```
/*
****************************************************************************
**
** FUNCTION AUTHOR(S):
**    Michael Drew
**
** FUNCTION NAME:
**
** FUNCTION INPUTS:
**    Variables that contain useful information
**
** FUNCTION DESCRIPTION
**    This is a function header.
**    Each function should have one
**
** FUNCTION OUTPUTS:
**    Returns a value that has meaning
**
****************************************************************************
*/
```

### 4.11 White Space

English uses white space (blank lines) to help separate one idea from another when ideas are expressed in paragraphs. Likewise code should be broken up using white space to help show the breakdown of tasks within a function.

### 4.12 Low Level Data Abstraction

In order to help insure code portability the use of standard (but ambiguous) data types should be avoided. The following should be included in an include file called ctypes.h:

```
     typedef unsigned char   U8;
     typedef unsigned short  U16;
     typedef unsigned int    U32;
     typedef signed char          S8;
     typedef signed short    S16
     typedef signed int      S32
```

These values are for example only. The size of scalar data types varies greatly from machine to machine and from compiler to compiler with the same target. The best defense against this sort of thing is to KNOW YOUR COMPILER. Once you know your compiler, codify this knowledge into the ctypes.h file so that it is documented.

### *4.13 Compiler and Processor Dependence*

In order to achieve reasonably portable code, all processor and compiler dependencies need to be codified and isolated in a uniform manner.

Just as every system should contain a ctypes.h header file that provides abstraction of low level data types, every system should contain a cpu.h file that abstracts processor and compiler dependencies. For example, compiler vendors that cater to the 68XX family of processors often provide non-ANSI extensions that allow variables to be placed in page zero ( fast access) memory. This extension takes the following form:

```
@dir int x;
```

This valuable feature can be used and the code still remain portable by placing the following line in cpu.h:

```
#define    FAST  @dir       /* Place variable in page zero for fast access */
```

This technique has the added benefit of giving an arguably better name to this extension as well. Now, whenever as programmer wants to use a fast variable he simply declares it as:

```
FAST U8 index;
```

When and if the code moves to a compiler or platform that does not support this feature, the cpu.h header file is the only file that needs to change.

All other non-ANSI features that a programmer wishes to use must be codified in the cpu.h file. Extreme care should be taken when using any non-ANSI feature. Thought should be given to how this feature can be implemented on a strictly ANSI compiler. If a feature can not be easily duplicated or removed, both **portability and non-target unit test capability WILL BE LOST**.

### *4.14 Hardware Platform Dependence*

Abstraction of the hardware platform is best achieved through the use of a header file, hardware.h, that isolates and codifies the hardware platform. This file should contain constants that define the following:

- Peripherals
    - Addresses
    - Interrupt vectors or levels
    - Special Access macros (upper byte only etc)
- Memory
    - Type (FLASH, RAM, EPROM)
    - Start Address

- End Address
- Size (Byte, Word, LONG)

### *4.15 Unit Test Code*

The inclusion of unit test code within a module can be an invaluable aide to programmers involved with the maintenance or enhancement of software modules. Unit test code can be used to regression test a module once a modification has been made to it. Without unit test code, the software developer must rely on integration or system testing to uncover defects.
It is notoriously difficult to exercise all critical paths and edge conditions during integration or systems test.

Unit test code should be located at the end of the module. It should be "set off" from the rest of the code by some attention grabbing mechanism like the following:

```
/***********************************************************************/
/*                                                                     */
/*             WARNING!  WARNING! WARNING! WARNING!                    */
/*                    UNIT TEST CODE FOLLOWS                           */
/*                                                                     */
/***********************************************************************/
```

The unit test code and any functions that it employs to exercise the main function contained in this module should be contained within conditional compile directives such as:

```
#ifdef UNIT_TEST_FOR_THIS_MODULE

#endif     /* UNIT_TEST_FOR_THIS_MODULE */
```

Where "THIS_MODULE" is replaced with the module's name in upper case.

The unit test code should contain any stub functions that may be required to simulate external interfaces required to test this module along with a main function that will allow the unit test code to be compiled and linked to a stand alone executable image.

### *4.16 Efficiency*

All software developers should strive to achieve the simplest solution to the problem at hand.

**"More computing sins have been committed in the name of efficiency (without necessarily achieving it) than any other reason -- including blind stupidity."**
(W.A. Wulf, Proceedings of the 25th National ACM Conference)

The following philosophy of optimization should be kept in mind when designing or coding a system:

- It is easier to make a working system efficient than it is to get an efficient system working.
- Optimize a system only if it fails a performance test. It is futile to optimize a system, if that optimization brings no practical benefit.
- Simplicity is a virtue that bears its own rewards.
- Optimize only those parts of the system that need optimizing.

Efficiency is more likely to be achieved by selecting the appropriate algorithm than by trying to squeeze out microseconds by tricky coding. Tricky coding is much harder to understand and more difficult to get right than simple, straight forward code.

## 5. Conclusions

A coding standard can not magically convert bad programmers into good programmers. A coding standard can not make up for a bad design or a lack of design review. A coding standard can help a programmer identify when a design has gone astray only if it is used. A coding standard without a rigorous code walk-through or inspection process will yield only marginal improvements in the quality of software that an organization produces.

## 6. Appendix A

Example of header file ctypes.h

```
#define   PRIVATE   static
#define   PUBLIC

#define   TRUE      (1)
#define   FALSE     (0)

/*
 *   The following are processor/Compiler dependent
 */
typedef unsigned char   U8;
typedef unsigned short  U16;
typedef unsigned int    U32;
typedef signed char          S8;
typedef signed short    S16;
typedef signed int      S32;
```

## 7. Appendix B

Example of typical module layout

```
/*   $Id$ */
/*************************************/ /*************************************/
/*     CONFIDENTIAL PROPRIETARY      */ /*    CONFIDENTIAL PROPRIETARY       */
/*      UNPUBLISHED SOURCE CODE      */ /*     UNPUBLISHED SOURCE CODE       */
/* Copyright 1996, Drew Technologies */ /* Copyright 1996, Drew Technologies */
/*       ALL RIGHTS RESERVED         */ /*       ALL RIGHTS RESERVED         */
/*************************************/ /*************************************/
/*
**
** MODULE AUTHOR(S):
**   Michael Drew
**
** MODULE TITLE:
**   appendix_a.c
**
** MODULE FUNCTION:
**   This is module functions as a complete example of a module for
**   appendix_a.
**
** History
 * $Log$
*****************************************************************************
*/
#include <ctypes.h>
#include <cpu.h>
/*
 *   Include any xxx_user.h type files that this module needs here.
 */
#include <appendix_a_user.h>

/*
 *   Declare PUBLIC variables here
 */
PUBLIC U32 appendix_a_public_data;

/*
 *   Place PRIVATE data structures local to this module here
 */
typedef struct
{
     U32  data;
}TYPE_APPENDIX_A;

/*
 *   Declare PRIVATE variables here
 */
PRIVATE U32 appendix_a_private_data;

/*
 *   Place PRIVATE Function prototypes here
 */
PRIVATE U32 appendix_a_function_2(U32 parameter);
```

```
/*
*****************************************************************************
**
** FUNCTION AUTHOR(S):
**    Michael Drew
**
** FUNCTION NAME:
**    appendix_a_function_1
**
** FUNCTION INPUTS:
**    None
**
** FUNCTION DESCRIPTION
**    This is one of the main functions for this module.
**    This function is known to the outside world through the
**    appendix_a_user.h header file
**
**
** FUNCTION OUTPUTS:
**    None
**
*****************************************************************************
*/
PUBLIC void
appendix_a_function_1(void)
{
      /*
       *    Due some work here
       */
      appendix_a_function_2();
}
```

```
/*
*****************************************************************************
**
** FUNCTION AUTHOR(S):
**    Michael Drew
**
** FUNCTION NAME:
**    appendix_a_function_2
**
** FUNCTION INPUTS:
**    None
**
** FUNCTION DESCRIPTION
**    This is one of the worker functions for this module.
**    This function is not known to the outside world.
**
** FUNCTION OUTPUTS:
**    None
**
*****************************************************************************
*/
PRIVATE void
appendix_a_function_2(void)
{
     /*
      *    Due some work here
      */
}
```

```
/*******************************************************************/
/*                                                                 */
/*              WARNING!  WARNING! WARNING! WARNING!               */
/*                     UNIT TEST CODE FOLLOWS                      */
/*                                                                 */
/*******************************************************************/
#ifdef UNIT_TEST_APPENDIX_A

/*
****************************************************************************
**
** FUNCTION AUTHOR(S):
**   Michael Drew
**
** FUNCTION NAME:
**   main
**
** FUNCTION INPUTS:
**   None
**
** FUNCTION DESCRIPTION
**   This is main function for unit testing this module.
**
** FUNCTION OUTPUTS:
**   None
**
****************************************************************************
*/
void
main()
{
     /*
      *   Code to exercise appendix a
      */
     appendix_a_function_1();
     printf("It worked!\n");
}

#endif     /* UNIT_TEST_APPENDIX_A */
```