# APPENDIX A : Example Standard

<--Prev page | Next page -->

If you have no time to define your own standards, then this appendix offers you a pre-cooked set. They are deliberately brief, firstly because standards should not be too long as no-one will remember them all, and secondly because of space limitations in the book!
They are targeted at a fairly stable organization with about average programmers (neither wizards nor neophytes). In comparison to other known sets of standards they are fairly middling, more detailed than some and less detailed than others.
The brevity means that there are no explanations as to why standards are selected, although reasons are available in the rest of this book. Other space-saving tricks are also used, such as including standards within a template format and emphasizing the basic principles to use in the event of there being no specific standard for a given situation.
As an alternative to taking these standards as a complete set, you may use them as a basis for you own standards, rejecting those parts you really cannot agree with, and adding parts you consider to be missing.

# MegaByte Company Coding Standards

These are the C coding standards for the MegaByte company. All project team are expected to adopt and ensure usage of these standards. Variations are permitted if agreed by the project team council.

## 1. Basic Principles

The fundamental purpose of these standards is to promote maintainability of the code. This means the code must be readable, understandable, testable and portable.

### The Four Cornerposts

Always keep the cornerposts in mind. Where there is conflict, compromise towards readability.

1. Keep it *simple*. Break down complexity into simpler chunks. Clearly comment necessary complexity.
2. Be *explicit*. Avoid implicit or obscure features of the language. Say what you mean.
3. Be *consistent*. Use the same rules as broadly as possible.
4. Minimize *scope*. This includes logical and visual scope. Be more explicit about items with wider scope.

### Other general points

- Minimize performance tweaks. Be aware of real savings made.
- Use a well defined error handling and diagnostics system. Ensure data integrity and enable recovery.
- Make data and process work together to reflect the problem.

- Where you have a coding decision and there is no direct standard, then you should always keep within the *spirit* of the standards.

### Files

- Divide header files to reflect the files they serve. For each subsystem, have: one or more interface files; one common file; lower level files as required.
- #include "megabyte.h" in all files - this includes common items such as definitions of TRUE and FALSE.
- Keep files less than 1000 lines or about 20 pages.
- Make files functionally cohesive. Aim for one public function per file, plus its private functions. Allow more public functions if they share several private functions.
- Minimize coupling between modules.

## 2. Commenting

- Make every comment count (but don't use this as an excuse for under-commenting).
- Keep code and comments visually separate.
- Use header comments for all files and functions. Smaller comments are ok for smaller, private functions.

- Use block comments regularly. Use trailing comments for special items.
- see Layout Template section for examples.

## 3. Naming

- Name separate words each with an initial capital (e.g. VariableName).
- Use abbreviations consistently, and document common usages (eg. 'Buf' for buffer).
- Use two/three capitals plus underscore to show functional prefix on all global items (functions and data) (e.g. XY_VariableName). Document prefixes used.
- Use between 2 and 20 characters per name. Use longer names for items with wider scope.
- Name functions with verb-noun (e.g. ProcessVariable); name variables with noun-abstract_noun (e.g. WallHeight).
- Use the following prefixes to denote special types: p = pointer, a = array, u = unsigned, b = boolean. (e.g. paWall = pointer to an array).

## 4. Layout

- Aim for one action per line.
- Use 'comb' rather than 'toothbrush' layout.
- Use vertical alignment.
- Separate code 'chunks' with blank lines or comments.
- No spaces within object references. No space between unary operator and operand (but space on other side). Balance spacing around binary operators (usually one space).
- Use parentheses to emphasize chunks in expressions.
- Use precedence rules to guide wrapping of expressions.
- When wrapping lines, indent to past logical start of wrapped item.
- Always use braces, even for a null statement.
- See Layout Template below for examples and further standards.

## 5. Usage

- Use the appropriate construct for the appropriate situation.
- Avoid deep nesting (of statements, parentheses and structures). Aim for normal maximum of three levels (deeper excursions should be short and infrequent).
- Minimize use of conditional expressions.
- Don't use implicit evaluation order in equality-expressions (use separate statements instead).
- Minimize use of comma operator.
- Use casts to maintain type in expressions, assignments and parameters.
- In 'if..else', put the major action in the 'if' clause.
- In control loops, aim for the controlling equality-expression to remain true for most of the loop.
- Use 'for' expressions only for control of the loop.
- Make 'case's short and independent of one another. Always use a 'default'.
- Only use 'goto' for error handling, jumping outwards and downwards.
- Use 'break' to escape a loop under abnormal, rather than normal, conditions.
- Avoid 'continue'. Use 'if..else' instead.
- Always declare function type, even 'int' and 'void'.
- Use function prototypes over 'original' style declarations.
- Avoid functions with large numbers of parameters.
- Match the type of actual and formal parameters (possibly using casts).
- Don't re-#define 'C'.
- Minimize macro usage, especially tricky usage. Parenthesize parameters and the overall expression. If it contains multiple statements, use 'do { macro body } while( FALSE )'.

- In declaring integers, default to using 'int'. Use 'char' and 'short' only to save space and only use 'unsigned' and 'long' where explicitly needed. Use up to two 'register' variables only at specific performance enhancement points.
- In declaring floating points, default to 'double', only using 'float' to save space.
- Compare floating point numbers only with '<=' or '>='.
- Use 'typedef' over #define or 'struct Tag {..};'.
- Do 'typedef' common types, but don't over-use 'typedef'.

- Minimize use of global data. Rather, hide data with or in owning functions.
- Keep structures functionally cohesive. Nest them where appropriate, using a consistent approach (nesting actual structures or pointers).
- Minimize use of unions.
- Use array indexing over pointers into arrays.
- Use NULL for the null pointer. Cast it, too.
- Always cast pointers when moving between types.
- Never use 'magic numbers' - always #define. (Exception: simple usage, such as initializing a count to 0).
- Use enum over #define.

# 6. Layout template

## 6.1 Header File

```
/*H*****************************************************************
* FILENAME :        filename.h              DESIGN REF: XXXXNN
*
* DESCRIPTION :
*       Simple description of functional areas served
*
* USAGE :
*       Description of when file should be #include'd
*
* NOTES :
*       Other help for the reader, including references.
*
*       Copyright (c) MegaByte Co. 1990, 1992.  All rights reserved.
*
* CHANGES :
*
* REF NO  DATE     WHO      DETAIL
*         DDMMMYY  Name     First version
* XXNNNN  DDMMMYY  Name     Name of item changed
*
*H*/

#ifndef filename_h
#define filename_h

/*****************************************************************************
*   INCLUDE FILES
*****************************************************************************/
/*---- system and platform files -----------------------------------------*/
#include <filename.h> header files not defined within the program
...
/*---- program files -----------------------------------------------------*/
#include "megabyte.h" Company standard: includes TRUE, FALSE, etc.
#include "program.h"  'global' header file for program
#include "filename.h" other files required
...
/*****************************************************************************
*   FILE CONTEXT
*****************************************************************************/
#define's, typedef's, struct's, union's, enum's for use in
more than one functional area below.

<page break>

/*A*****************************************************************
* NAME:     Simple name for functional area served by this section
*
* USAGE:    Description of where, how, etc. items below are used
*
* NOTES:    Other information to help the reader.
*
* CHANGES :
* REF NO  DATE     WHO      DETAIL
*         DDMMMYY  Name     First version
* XXNNNN  DDMMMYY  Name     Name of item changed
*
*A*/
```

```
/*---- context ------------------------------------------------*/
#define's, typedef's, struct's, union's, enum's for use in
more than one item below.
...
/*---- data descriptions --------------------------------------*/
Use typedef's over struct's and #define's
Use enum's over #define's
...
/*---- extern data declarations -------------------------------*/
extern  type    VariableName; /* comment        */
...
/*---- extern function prototypes -----------------------------*/
extern  FunctionName( type Parm1, type Parm2 )
...

<page break>

/*A*****************************************************************
* NAME:     next functional area (as previous section)
...

#endif /* filename_h ------ END OF FILE ------*/
```

## 6.2 Header file for global data

```
/*H*****************************************************************
* FILENAME :        filename.h              DESIGN REF: XXXXNN
*
...  Header comment same format as normal header file
*H*/

#ifndef GLOBAL
#define GLOBAL extern
#endif

#ifndef filename_h
#define filename_h

Layout file in similar manner to normal header file

GLOBAL type    VariableName;  Prefix all declarations with 'GLOBAL'
...
#endif /* filename_h */
#undef GLOBAL /*------- END OF FILE ------*/
```

**Then, to define data in global data file:**

```
#define GLOBAL
#include "filename.h"
```

**..and to declare external data in source code file:**

```
#include "filename.h"
```

## 6.3 Code file

```
/*H*****************************************************************
* FILENAME :        filename.c              DESIGN REF: nnnnnn
*
* DESCRIPTION :
*       Simple description of main function of file
*
```

```
* PUBLIC FUNCTIONS :
*       type    FunctionName( Parameter1, Parameter2 )
*
* NOTES :
*       Other help for the reader, including references.
*
*       Copyright (c) MegaByte Co. 1990, 1992.  All rights reserved.
*
* CHANGES :
*
* REF NO  DATE     WHO      DETAIL
*         DDMMMYY  Name     First version
* F.NNNN  DDMMMYY  Name     Name of function/item changed
*
*H*/

static char ID_filename[] = "filename:Copyright MegaByte Co. 1990, 1992"

/*************************************************************************
*   INCLUDE FILES
*************************************************************************/
/*---- system and platform files -----------------------------------------*/
#include <filename.h> All files not defined within the program
...
/*---- program files -----------------------------------------------------*/
#include "megabyte.h" Company standard: includes TRUE, FALSE, etc.
#include "program.h"  'global' header file for program
#include "filename.h" interface files for called subsystems
...
#include "filename.h" local files for current subsystem
...

/*************************************************************************
*   EXTERNAL REFEFENCES      NOTE: only use if not available in header file
*************************************************************************/
/*---- data declarations -------------------------------------------------*/
extern   ...
/*---- function prototypes -----------------------------------------------*/
extern   ...
/*************************************************************************
*   PUBLIC DECLARATIONS      Defined here, used elsewhere
*************************************************************************/
/*---- context -----------------------------------------------------------*/
#define's, typedef's, struct's, union's, enum's for use in declarations and
functions
...
/*---- data declarations -------------------------------------------------*/
...
/*---- function prototypes -----------------------------------------------*/
declare in same order as as definitions, below
...

/*************************************************************************
*   PRIVATE DECLARATIONS     Defined here, used only here
*************************************************************************/
/*---- context -----------------------------------------------------------*/
#define's, typedef's, struct's, union's, enum's for use in declarations and
functions
...
/*---- data declarations -------------------------------------------------*/
```

```
static   ...
...
/*---- function prototypes ------------------------------------------------*/
declare in same order as as definitions, below
static   ...
...
/*---- macros -------------------------------------------------------------*/
...
```

*<page break>*

```
/*****************************************************************
*   PUBLIC FUNCTION DEFINITIONS
*****************************************************************/

/*F************************************************************* NAME :
type FunctionName( Parm1, Parm2)
*
* DESCRIPTION :     Simple description of function's function
*
* INPUTS :
*       PARAMETERS:
*           type    VariableName     Description of usage
*       GLOBALS :
*           type    VariableName     Description of usage
*
* OUTPUTS :
*       PARAMETERS:
*           type    VariableName     Description of usage
*       GLOBALS :
*           type    VariableName     Description of usage
*       RETURN :
*            VALUE1           Description of value
*            VALUE2           Description of value
*
* NOTES :           Other assumptions, hints, for the reader
*
* CHANGES :
* REF NO  DATE     WHO      DETAIL
*         DDMMMYY  Name     First version
* XXNNNN  DDMMMYY  Name     Description of change
*F*/

type
FunctionName( type Parm1, type Parm2 )
{
/******* DATA *********************************************/

One declaration per line.

type    Variable1;           /* comment on purpose of Variable1          */
type    Variable2;           /* comment on purpose of Variable2          */

/******* CODE *********************************************/
    statement;
    statement;                          /* trailing comment only where..  */
                                        /* ..it is necessary. Default     */
                                        /* ..start in col.40              */
    statement;

/*====== single line comment, major point in code ===============*/
    statement;
```

```
    statement;

/*------ single line comment, minor point in code ---------------*/
    statement;
    statement;

/*------ block comment ------------------------------------------*
*   Comment on what has just happened and/or what is about to happen.
*   Start in column 1 to help programmable editor to auto-insert block.
*-------------------------------------------------------------------*/
    statement;
    statement;
}

/*********************************************************************
*   PRIVATE FUNCTION DEFINITIONS
*********************************************************************/

/*f*****************************************************************/

PRIVATE type
FunctionName(
    type    Variable1,    /* Description of variable  */
    type    Variable2     /* Description of variable  */
);

/*
* PURPOSE : Brief description of what function does
*
* RETURN :  Complete description of values of return
*
* NOTES :   Other hints, etc. for the reader
*f*/

{
/******* DATA ********************************************************/
type    Variable1;            /* comment on purpose of Variable1          */
type    Variable2;            /* comment on purpose of Variable2          */

/******* CODE ********************************************************/
    statement;
    statement;
}
```

<page break>

```
/*f*****************************************************************/
Next definition of private function, etc. ...
Constructs:

if ( expression )     'if'
{
    statement;
    statement;
}

if ( long expression which wraps
     onto the next line )
{
    statement;
    statement;
```

```
}

if ( expression )
{                 Always use braces
    statement;
}

if ( expression )     Even on null statement..
{                 ..always use braces
}

if ( expression )     'if..else'
{
    statement;
    statement;
}
else
{
    statement;
    statement;
}

if ( expression )     'if..else if....else'
{
    statement;
    statement;
}
else if
{
    statement;
    statement;
}
else if
{
    statement;
    statement;
}
else              Always use default 'else'
{
    statement;
    statement;
}

while ( expression )      'while'
{
    statement;
    statement;
}

while ( long expression which wraps
        onto the next line )
{
    statement;
    statement;
}

while ( TRUE )            Infinite loop
{
    statement;
```

```
    statement;
}

for ( expression; expression; expression )    'for'
{
    statement;
    statement;
}

for ( expression;
        expression;
        expression )
{
    statement;
    statement;
}

for (   expression which wraps
            onto next line;
        expression which wraps
            onto next line;
        expression which wraps
            onto next line )
{
    statement;
    statement;
}

do /* until xyz happens */    'do'
{
    statement;
    statement;

} while ( expression )

switch ( expression )     'switch'
{
case VALUE1:
    statement;
    statement;
    break;            Always use 'break'

case VALUE2
    statement;
    /********** FALLTHROUGH **********/

case VALUE2:
case VALUE3:
    statement;
    statement;
    break;

default:              Always handle default case gracefully
    statement;
    break;
} /* end switch ( expression ) */ Closing comment on end of all long blocks
                  'expression' is optional
```

```
Variable =  ( expression )    Minimize use of conditional expression
            ? expression
            : expression ;

FunctionCall( Parm1,      Use when many parameters or multiple line wrap
                Parm2,
                Parm3 );

struct TAG_NAME           'struct's, 'union's and 'enum's use similar layout
{
    type    VariableName1;
    type    VariableName2;
};

char *  pVariableName1;       Separate '*' in pointer declarations
struct TAG_NAME           Use comb layout to preserve vertical alignment
        VariableName2;
```