MCCI Corporation
3520 Krums Corners Road
Ithaca, New York 14850 USA
Phone +1-607-277-1029
Fax +1-607-277-6844
www.mcci.com

MCCI Coding Standards

Engineering Report 950134
Rev. H
Date: 2009/03/26

Document Release History

Rev. E	2005/03/03	Update copyright information
Rev. F	2007/09/09	Update company and copyright information
Rev. G	2008/01/09	Update coding samples
Rev. H	2009/03/26	CJY added discussion of “identifying parameters in …”

TABLE OF CONTENTS

MCCI Coding Standards

1 Preface

MCCI has a recognized set of coding standards that were developed by Terry Moore and which have remained essentially constant since 1982. Following these rules assures better code, easier code walk-throughs, and better products.

These rules were written for C and assembler, but are generally applicable for many other languages (DABIL, MS-BASIC, Pascal, whatever). The C rules were developed mostly around Whitesmith's C compiler, and have since been used on all manner of compilers on microcomputers, VAXen, UNIX workstations, etc. with good effect. The assembler rules were developed mostly around the Intel ASM86 macro assembler, but work fine with MASM and other assemblers of that sort. Obviously portability stands along with accuracy, precision and clarity as one of the primary goals of the standards.

2 References

There are a number of standard reference texts which form the basis for proper C coding practices. In fact, choosing the proper references is an important part of defining the MCCI Standards.

Three that are useful are:

Harbison & Steele, "C: A Reference Manual", 2nd. Ed., Prentice-Hall, 1987

Plauger & Brodie, "Standard C", Microsoft Press, 1989

Plum, "C Programming Guidelines", 2nd. Ed., Plum Hall, Inc. 1995

3 Documentation: Module Header Block

Every module starts with a header block according to this skeleton:

```
/* name.c  Day Mon dd yyyy hh:mm:ss user */

/*

Module:  name.c

Function:
      One- or two-line description.

Version:
      VN.nna Day Mon dd yyyy hh:mm:ss user  Edit level nn
```

```
Copyright notice:
	This file copyright (C) yyyy by

		MCCI Corporation
		3520 Krums Corners Road
		Ithaca, NY  14850

	An unpublished work.  All rights reserved.

	This file is proprietary information, and may not be disclosed or
	copied without the prior permission of MCCI Corporation.

Author:
	FirstName LastName, MCCI	Month yyyy

Revision history:
   1.00A  Day Mon dd yyyy hh:mm:ss  user
	Module created.

*/
```

Along with the above skeleton, there are the following directions:

- Use an editor that maintains real ASCII tabs (0x09) at 8 columns, and that handles form feeds (0x0C). UniPress or equivalent EMACS is preferred but not dictated.
- Do not change the format or position of the various elements in the header.
- Update the date strings when opening an edit, otherwise you'll forget.

This format must be followed exactly, because many of us have editor macros which do the date, version, and edit updates automatically. The "`V`" at the head of the version string, and the date format, are quite important, even to the position and number of tabs and spaces.

"`user`" is the username of the person currently editing the file.

"`Edit level`" is a counter that increments with each edit, to provide a simpler way to order multiple versions of a given source file.

The "`Month yyyy`" date in "`Author`" is the original date the module was created. It is not generally changed once the file is created.

Revision levels of the form "`N.nna`" are interpreted as follows:

`N`	major release
`nn`	minor release, usually incremented, sometimes bumped by 10.
`a`	sub-minor level, incremented alphabetically

The first "`Revision history`" entry is always there -- it is a part of the standard skeleton.

The next part of the skeleton concerns headers. If the file itself is a header file, this part has the `#ifdef` to prevent multiple includes. If it is a code file, this part has the list of `#includes`. The page ends with a literal form-feed.

For a header file:

```
#ifndef _FILENAME_H_         /* prevent multiple includes */
#define _FILENAME_H_
```

For a code file:

```
#include <sysheader.h>       /* compiler-supplied headers */

...

#include <CompuDAS.h>        /* MCCI standard header */
#include <other_mc.h>        /* other MCCI headers */

  ...

#include "localheader.h"           /* project-specific headers */

  ...
```

If the module warrants it, the next page(s) contain text, embedded within comment lines, which explains the contents of the file. This section ends in a literal form-feed. For example:

```
/*******************************************************************************\
|     This module is an example module.
|
|     (Text explanation of contents of file, as long as required.)
|
|     At one time, specially-formatted 'help' program text was embedded here
|     also, to aid in maintaining a separate 'help' facility.
|
\*******************************************************************************/
```

The next section contains manifest constants, macros, and typedefs that are used only within the particular file. This rule applies to code modules, not to header files (because header files normally contain manifest constants, macros and typedefs that are used by other modules). The section begins with a banner.

Source file also includes any forward declarations or function prototypes required for functions local to this module. This section ends with a form-feed.

```
/****************************************************************************\
|
|           Manifest constants & typedefs.
|
|     This is strictly for private types and constants which will not
|     be exported.
|
\****************************************************************************/
```

The next section contains initialized constants which are not modified during program operation. This allows them to be placed in ROM-able objects. This section only applies to code modules, and begins with a banner.

```
/****************************************************************************\
|
|     Read-only data.
|
|     If program is to be ROM-able, these must all be tagged read-only
|     using the ROM storage class; they may be global.
|
\****************************************************************************/
```

The next section contains initialized variables and "unititialized" variables. The latter are actually initialized with a keyword "`BSS`", which is `#defined` in a standard header as the null string or "`{}`" as needed by the particular compiler. This section only applies to code modules, and begins with a banner.

```
/****************************************************************************\
|
|     VARIABLES:
|
|     If program is to be ROM-able, these must be initialized
|     using the BSS keyword.  (This allows for compilers that require
|     every variable to have an initializer.)  Note that only those
|     variables owned by this module should be declared here, using the BSS
|     keyword; this allows for linkers that dislike multiple declarations
|     of objects.
|
\****************************************************************************/
```

Comments on the Main Header Block

Since all the header info and banners are in the skeleton, they cost zero coding time. There is a small overhead dealing with the date and revision history, but that has to be done in any case. Any reasonable editor can be programmed to automatically update the date strings with the current date and time.

Although the banners for the various sections are a bit verbose, they serve a useful function in partitioning the beginning part of the module, where too often programmers throw a jumble of variables, typedefs, and so forth. MCCI has had good results from retro-fitting this header onto sources that did not come with it, and re-arranging the variables and such. It's surprising what an edit session like that can turn up.

Furthermore, the extra text serves as helpful information to reviewers who may not be familiar with the code.

Documentation: Function Header Block

Every function begins with a header block, according to this skeleton:

```
/*

Name: name()

Function:
      One-line description of the function.

Definition:
      TYPE name(
            TYPE arg1, brief description of arg1
            TYPE arg2  brief description of arg2
            );

Description:
      Full description of the function.

Returns:
      Description of the return value and its interpretation.

Notes:
      Notes

*/
```

The header should contain every piece of information required to use the function. A successful header makes it unnecessary to look into the code to see how to call it or employ its return value.

The "`Definition`" is the ANSI C function prototype. This tells the reader how to call the function. If the brief descriptions of the args require expansion, then that can be done in the "`description`."

The "`Description`" is a text discussion of the operation of the function, first from the caller's point of view, and then giving the internal details as needed. For large functions this could easily run a page or more of text; for truly self-explanatory functions, this may be a single sentence. Stylistically, this is usually indented if it is only a paragraph or two long. If it is much longer, it may be more practical to move the left margin back to the first column, to allow more line length.

The “`Returns`” gives the possible return values. There are a few conventions on how this is phrased, for consistency and rapid readability.

Functions returning type `BOOL`:	`TRUE if condition ... else FALSE`
Functions returning type `ERRCODE`:	`0 if no error, else ... list ...`
Functions returning type `void`:	`Nothing.`

The practice of writing vestigial headers for trivial or self-explanatory functions is discouraged, but may be occasionally excused under circumstances of extreme time pressure. However, for production code, it is deemed unacceptable, e.g.;

```
/*
** int getfoo (int n) - - return the nth foo.
*/
```

Usage and Style Conventions

This is a collection of the preferred coding practices that do not necessarily affect the eventual binary object, but affect the format and readability of the source code.

These are the items that are fairly MCCI-specific. General guidelines, such as govern data-hiding, modularity, structuring, and so forth are available in reference texts.

Indentation and Blocking

Each level of indent is 8 columns, and is accomplished with an ASCII TAB character (0x09). Blocking is done with the block indented one level from the line above it, so that the indented code and its surrounding braces are at the same indent level. Braces are not required around blocks of only one line, but are strongly suggested to avoid subtle bugs.

```
int getfoo(
        int n
        )
        {
        int ii;

        whatever(n);
        if (n > 0)
                {
                something(n);
                for (ii = 0; ii < n; ++ii)
                        {
                        int var;

                        var = other_thing(ii);
                        another_thing(var, ii);
                        }
```

```
            }
      else
            {
            return(0);
            }
      return(do_foo(n));
      }
```

As shown, place a single blank line after automatic variable declarations and before the code.

Incidentally, WoodDuck Computing (http://www.wdcc.com/) has a "C beautifier" program for MS-DOS that will take nearly any C code and produce output like that shown above.

Programmer-Defined Identifiers (Symbol Names)

Recommendations here are subject to modification when building components for given operating environments.

Programmer-defined identifiers follow these guidelines:

- Use all lowercase for variables and functions (except as noted for Mixed case below):

  ```
  int foo;
  ```

- Use all uppercase for manifest constants and macros, except macros that intentionally mask library functions of the same name, which must necessarily match the case of the masked function:

  ```
  #define MAXLIMIT  1000
  #define HI(val)   (((val) >> 8) & 0xFF)
  #define strlen    my_strlen
  ```

- Use mixed case for global variables and functions having long names, for readability. Capitalize the first letter of each word comprising the identifier:

  ```
  long MaxRepCount;
  ```

- Consider beginning global variable names with "`g`".

  ```
  long gMaxRepCount;
  ```

- Consider pre-coding pointer names with "`p`".

  ```
  TEXT *pName;
  ```

- Global pointers could begin "`gp`" not "`pg`" because "`g`" means "`global`" and "`p`" means "`point to`".

- Do not generate distinct identifiers that differ only in the case of their letters:

```
int value, Value; /* very bad idea */
```

- Do not generate an identifier that overrides a standard library name:

```
int time;         /* very bad idea */
```

- Consider using embedded underscores instead of mixed case for portable readability - - some compilers and linkers force all uppercase on external identifiers. Watch compiler/linker switches on those that accept mixed case:

```
int SymbolWithLongName;       /* common modern usage */
int symbol_with_long_name;    /* but this is more portable */
```

- Do not use leading underscores on normal identifiers: they are likely to collide with compiler macros. A single leading underscore is reserved for special, hidden, or implementation-dependent compiler macros; a double leading underscore is reserved for hidden or pre-defined compiler macros, or for library use.

- Name your loop indices meaningfully when possible. However, trivial integer indices can be `“i”, “j”,` etc.

- If you are very concerned with portability, you may need to keep your identifiers unique in the first six or eight characters. Although most compilers accept long names, some linkers only consider the first six or eight characters of external names to be significant. However, MCCI practice is to rely on compiler or (if necessary) pre-processing tools to work around these issues.

Expressions and Conditions

- Write expressions that are meaningful in terms of the data types involved, without forcing the compiler to make conversions. For example:

Avoid	Preferred
`if (!count)`	`if (count == 0)`
`while (*ptr)`	`while (*ptr != '\0')`
`if (strcmp (s1, s2))`	`if (strcmp(s1,s1) != 0)`
`if (fp = fopen("foo","r"))`	`if ((fp = fopen("foo","R")) != NULL)`

- Although this is a matter of personal coding style, the use of the `CompuDAS.H #define` macros for unary negation (`NOT` instead of `!`) and the binary equality comparisons (`EQ` instead of `==`, `NE` instead of `!=`) is strongly encouraged. They enhanced readability, and reduce confusion and typos.

- Enclose macro arguments and values in “extra” sets of parentheses to avoid accidental recombination with the context in which they are used. The classic simple example is:

```
#define EOF     -1                    /* bad idea */
#define EOF     (-1)                  /* correct */
```

But a more useful and complex example is:

```
        #define strenq(s1,s2) strcmp(s1,s2)==0        /* bad idea */
        #define strenq(s1,s2) ((strcmp((s1),(s2))==0) /* correct */
```

Note that each argument is enclosed, and the overall value is enclosed.

Initialization

- Initialize all global variables. Use the `BSS` keyword for "uninitialized" variables.
- Within a function, assign values to automatic locals with explicit code, rather than in the declaration. This avoids problems due to presumed initialization, by making the initialization explicit:

```
          Avoid                     Preferred               Also OK
    void foo(void)             void foo(void)          void foo(void)
      {                        {                       {
      int localvar = 0;        int localvar;           int const localvar = 0;
      bar (localvar);          localvar = 0;           bar(localvar);
      }                        bar(localvar);          }
                               }
```

- For pointers that are initialized to strings (or other arrays), use an array rather than a string "constant" for the storage block, since a string constant may be placed in read-only memory.

  Always assume that string constants are read-only. If you need a `const char*` variable that is initialized to point to a string, by all means use:

```
 const char *foo = "initial value";
```

  or

```
  const char * const foo = "initial value";
```

  [the latter declaration makes `foo` itself read-only; the former does not.]

  But if you need a `char*` variable that is initialized, but which will be updated in place, please use an initialized array. E.g. either:

```
  char foo[] = "initial value";
```

  or

```
  char foo_val[] = "initial value";

  char *foo = foo_val;
```

  Depending, of course, on how you need to use `foo`. In addition, it is somewhat more efficient to say: `const char foo[] = "initial value";` if you really don't need to have a separate pointer variable.

Data Types

- Use the `typedef` data types from MCCI's `libport` wherever possible for readability, portability and convergence with existing MCCI code. This is particularly important when considering integer values which might vary in length (16/32) with the environment.

- Wherever appropriate, specify the length of an integer type explicitly using `UINT16`, or `UINT32`, etc. for the libport library. This is especially important for portability between 16 and 32 bit environments, since the default width of the `int`, `short`, and `long` types will vary. For trivial loop indices, it's all right to use int or unsigned.

- Be careful to use unsigned types unless the value needs to represent negative values. Unsigned is almost always more efficient for loop indices over non-negative-definite ranges.

```
            Bad                              Better

int i;                            unsigned i;

for (i=0; i<10; ++ i)             for (i=0; i<10, ++i)

      {                                 {

      foo (i)                           foo (i);

      }                                 }
```

By the way, this is an example in which `int` or `unsigned` are preferable to `UINT16. Frequently,` computation in `int/unsigned` are more efficient than computation in values of limited bit widths. From the perspective of portability, therefore, `int/unsigned` are preferred in such contexts

Identifying the meaning of parameters passed in a function call

When calling a function, the comment that indicates what a given parameter does is included before the parameter, because this is a call. This is just normal style. Read it as: "pass pProbeFn as NULL", etc. Sample:

```
USBPUMP_HOST_DRIVER_CLASS_INIT_NODE_INIT_V1(
   /* pProbeFn */ NULL,
   /* pInitFn */ UsbPumpUsbdiClassGeneric_Initialize ,
   /* pConfig */ &sk_UsbPumpUsbdiGcd_ClassConfig,
   /* pPrivateConfig */ &sk_UsbPumpUsbdGD_PrivateConfig,
   /* DebugFlags */ UDMASK_ANY | UDMASK_ERRORS
   )
```

Identifying the meaning of parameters in a type declaration

For the example:

```
/* Preferred form */
__TMS_FNTYPE_DEF(
   USBPUMP_USBDI_GENERIC_GET_DEVICE_STATE_CB_FN,
   __TMS_VOID,
   (
   __TMS_VOID * /* pClientContext */,
   __TMS_VOID * /* pRequestHandle */,
   __TMS_USBPUMP_USBDI_GENERIC_STATUS /* ErrorCode */,
   __TMS_USBPUMP_USBDI_GENERIC_DEVICE_STATE * /* pDeviceState */
   ));
```

In this case, the comment is after because this statement is a declaration. This is a substitute for the more normal:

```
/* Bad form */
typedef VOID USBPUMP_USBDI_GENERIC_GET_DEVICE_STATE_CB_FN(
   VOID *pClientContext,
   VOID *pRequestHandle,
   USBPUMP_USBDI_GENERIC_STATUS ErrorCode,
   USBPUMP_USBDI_GENERIC_DEVICE_STATE *pDeviceState
   );
```

The reason we don't write the more normal way (with pClientContext NOT in a comment), is because pClientContext must be unbound at compile time, both in CPP and in C. (The user might have a macro pClientContext – and in that case the declaration will fail. If pClientContext is the name of a typedef or of an enum value, you may also have problems. So it's just a subtle portability practice.