I

Coding Conventions

I.1 Introduction

This document defines the Wind River Systems standard for all C code and for the accompanying documentation included in source code. The conventions are intended, in part, to encourage higher quality code; every source module is required to have certain essential documentation, and the code and documentation is required to be in a format that has been found to be readable and accessible.

The conventions are also intended to provide a level of uniformity in the code produced by different programmers. Uniformity allows programmers to work on code written by others with less overhead in adjusting to stylistic differences. Also it allows automated processing of the source; tools can be written to generate reference entries, module summaries, change reports, and so on.

The conventions described here are grouped as follows:

- **File Headings.** Regardless of the programming language, a single convention specifies a heading at the top of every source file.
- **C Coding Conventions**

I.2 File Heading

Every file containing C code—whether it is a header file, a resource file, or a file that implements a host tool, a library of routines, or an application—must contain a *standard file heading*. The conventions in this section define the standard for the heading that must come at the beginning of every source file.

The file heading consists of the blocks described below. The blocks are separated by one or more empty lines and contain no empty lines within the block. This facilitates automated processing of the heading.

- **Title:** The title consists of a one-line comment containing the tool, library, or applications name followed by a short description. The name must be the same as the file name. This line will become the title of automatically generated reference entries and indexes.
- **Copyright:** The copyright consists of a single-line comment containing the appropriate copyright information.
- **Modification History:** The modification history consists of a comment block: in C, a multi-line comment. Each entry in the modification history consists of the version number, date of modification, initials of the programmer who made the change, and a complete description of the change. If the modification fixes an SPR, then the modification history must include the SPR number.

  The version number is a two-digit number and a letter (for example, 03c). The letter is incremented for internal changes, and the number is incremented for large changes, especially those that materially affect the module's external interface.

The following example shows a standard file heading from a C source file:

Example I-1 **Standard File Heading (C Version)**

```
/* fooLib.c - foo subroutine library */

/* Copyright 1984-1995 Wind River Systems, Inc. */

/*
modification history
--------------------
02a,15sep92,nfs  added defines MAX_FOOS and MIN_FATS.
01b,15feb86,dnw  added routines fooGet() and fooPut();
                 added check for invalid index in fooFind().
01a,10feb86,dnw  written.
*/
```

I.3 C Coding Conventions

These conventions are divided into the following categories:

- Module Layout
- Subroutine Layout
- Code Layout
- Naming Conventions
- Style
- Header File Layout
- Documentation Generation

I.3.1 C Module Layout

A *module* is any unit of code that resides in a single source file. The conventions in this section define the standard module heading that must come at the beginning of every source module following the standard file heading. The module heading consists of the blocks described below; the blocks should be separated by one or more blank lines.

After the modification history and before the first function or executable code of the module, the following sections are included in the following order, if appropriate:

- **General Module Documentation:** The module documentation is a C comment consisting of a complete description of the overall module purpose and function, especially the external interface. The description includes the heading *INCLUDE FILES:* followed by a list of relevant header files.
- **Includes:** The include block consists of a one-line C comment containing the word *includes* followed by one or more C pre-processor **#include** directives. This block groups all header files included in the module in one place.
- **Defines:** The defines block consists of a one-line C comment containing the word *defines* followed by one or more C pre-processor **#define** directives. This block groups all definitions made in the module in one place.
- **Typedefs:** The typedefs block consists of a one-line C comment containing the word *typedefs* followed by one or more C **typedef** statements, one per line. This block groups all type definitions made in the module in one place.
- **Globals:** The globals block consists of a one-line C comment containing the word *globals* followed by one or more C declarations, one per line. This block

groups together all declarations in the module that are intended to be visible outside the module.

- **Locals:** The locals block consists of a one-line C comment containing the word *locals* followed by one or more C declarations, one per line. This block groups together all declarations in the module that are intended not to be visible outside the module.
- **Forward Declarations:** The forward declarations block consists of a one-line C comment containing the words *forward declarations* followed by one or more ANSI C function prototypes, one per line. This block groups together all the function prototype definitions required in the module. Forward declarations must only apply to local functions; other types of functions belong in a header file.

The format of these blocks is shown in the following example (which also includes the file heading specified earlier).

Example I-2 **C File and Module Headings**

```
/* fooLib.c - foo subroutine library */

/* Copyright 1984-1995 Wind River Systems, Inc. */

/*
modification history
--------------------
02a,15sep92,nfs  added defines MAX_FOOS and MIN_FATS.
01b,15feb86,dnw  added routines fooGet() and fooPut();
                 added check for invalid index in fooFind().
01a,10feb86,dnw  written.
*/

/*
DESCRIPTION
This module is an example of the Wind River Systems C coding conventions.
...
INCLUDE FILES: fooLib.h
*/

/* includes */

#include "vxWorks.h"
#include "fooLib.h"

/* defines */

#define MAX_FOOS        112  /* max # of foo entries */
#define MIN_FATS        2     * min # of FAT copies */

/* typedefs */
```

```
typedef struct fooMsg       /* FOO_MSG */
    {
    VOIDFUNCPTR func;        /* pointer to function to invoke */
    int arg [FOO_MAX_ARGS];  /* args for function */
    } FOO_MSG;

/* globals */

char *    pGlobalFoo;        /* global foo table */

/* locals */

LOCAL int numFoosLost;       /* count of foos lost */

/* forward declarations */

LOCAL int     fooMat (list * aList, int fooBar, BOOL doFoo);
FOO_MSG       fooNext (void);
STATUS        fooPut (FOO_MSG inPar);
```

I.3.2 C Subroutine Layout

The following conventions define the standard layout for every subroutine.

Each subroutine is preceded by a C comment heading consisting of documentation that includes the following blocks. There should be no blank lines in the heading, but each block should be separated with a line containing a single asterisk (*) in the first column.

- **Banner:** This is the start of a C comment and consists of a slash character (/) followed by 75 asterisks (*) across the page.
- **Title:** One line containing the routine name followed by a short, one-line description. The routine name in the title must match the declared routine name. This line becomes the title of automatically generated reference entries and indexes.
- **Description:** A full description of what the routine does and how to use it.
- **Returns:** The word *RETURNS:* followed by a description of the possible result values of the subroutine. If there is no return value (as in the case of routines declared **void**), enter:

  ```
  RETURNS: N/A
  ```

  Mention only true returns in this section—not values copied to a buffer given as an argument.

- **Error Number:** The word *ERRNO:* followed by all possible **errno** values returned by the function. No description of the **errno** value is given, only the **errno** value and only in the form of a defined constant.[1]

The subroutine documentation heading is terminated by the C end-of-comment character (*/), which must appear on a single line, starting in column one.

The subroutine declaration immediately follows the subroutine heading.[2] The format of the subroutine and parameter declarations is shown in *I.3.3 C Declaration Formats*, p.568.

Example I-3 **Standard C Subroutine Layout:**

```
/*******************************************************************
*
* fooGet - get an element from a foo
*
* This routine finds the element of a specified index in a specified
* foo.  The value of the element found is copied to <pValue>.
*
* RETURNS: OK, or ERROR if the element is not found.
*
* ERRNO:
*  S_fooLib_BLAH
*  S_fooLib_GRONK
*/

STATUS fooGet
    (
    FOO      foo,         /* foo in which to find element */
    int      index,       /* element to be found in foo */
    int *    pValue       /* where to put value */
    )
    {
    ...
    }
```

I.3.3 C Declaration Formats

Include only one declaration per line. Declarations are indented in accordance with *Indentation*, p.572, and are typed at the current indentation level.

The rest of this section describes the declaration formats for variables and subroutines.

1. A list containing the definitions of each **errno** is maintained and documented separately.
2. The declaration is used in the automatic generation of reference entries.

Variables

- For basic type variables, the type appears first on the line and is separated from the identifier by a tab. Complete the declaration with a meaningful one-line comment. For example:

```
unsigned    rootMemNBytes;   /* memory for TCB and root stack */
int         rootTaskId;      /* root task ID */
BOOL        roundRobinOn;    /* boolean for round-robin mode */
```

- The * and ** pointer declarators *belong* with the type. For example:

```
FOO_NODE *  pFooNode;        /* foo node pointer */
FOO_NODE ** ppFooNode;       /* pointer to the foo node pointer */
```

- Structures are formatted as follows: the keyword **struct** appears on the first line with the structure tag. The opening brace appears on the next line, followed by the elements of the structure. Each structure element is placed on a separate line with the appropriate indentation and comment. If necessary, the comments can extend over more than one line; see *Comments*, p.574, for details. The declaration is concluded by a line containing the closing brace, the type name, and the ending semicolon. Always define structures (and unions) with a **typedef** declaration, and always include the structure tag as well as the type name. Never use a structure (or union) definition to declare a variable directly. The following is an example of acceptable style:

```
typedef struct symtab   /* SYMTAB - symbol table */
    {
    OBJ_CORE    objCore;        /* object maintanance */
    HASH_ID     nameHashId;     /* hash table for names */
    SEMAPHORE   symMutex;       /* symbol table mutual exclusion sem */
    PART_ID     symPartId;      /* memory partition id for symbols */
    BOOL        sameNameOk;     /* symbol table name clash policy */
    int         nSymbols;       /* current number of symbols in table */
    } SYMTAB;
```

  This format is used for other composite type declarations such as **union** and **enum**.

  The exception to never using a structure definition to declare a variable directly is structure definitions that contain pointers to structures, which effectively declare another **typedef**. This exception allows structures to store pointers to related structures without requiring the inclusion of a header that defines the type.

For example, the following compiles without including the header that defines **struct fooInfo** (so long as the surrounding code never delves inside this structure):

CORRECT:
```
typedef struct tcbInfo
    {
    struct fooInfo *  pfooInfo;
    ...
    }  TCB_INFO;
```

By contrast, the following cannot compile without including a header file to define the type **FOO_INFO**:

INCORRECT:
```
typedef struct tcbInfo
    {
    FOO_INFO *  pfooInfo;
    ...
    }  TCB_INFO;
```

Subroutines

There are two formats for subroutine declarations, depending on whether the subroutine takes arguments.

- For subroutines that take arguments, the subroutine return type and name appear on the first line, the opening parenthesis on the next, followed by the arguments to the routine, each on a separate line. The declaration is concluded by a line containing the closing parenthesis. For example:

```
int lstFind
    (
    LIST *   pList,   /* list in which to search */
    NODE *   pNode    /* pointer to node to search for */
    )
```

- For subroutines that take no parameters, the word *void* in parentheses is required and appears on the same line as the subroutine return type and name. For example:

```
STATUS fppProbe (void)
```

I.3.4 C Code Layout

The maximum length for any line of code is 80 characters.

The rest of this section describes the conventions for the graphic layout of C code, and covers the following elements:

- vertical spacing
- horizontal spacing
- indentation
- comments

Vertical Spacing

- Use blank lines to make code more readable and to group logically related sections of code together. Put a blank line before and after comment lines.
- Do not put more than one declaration on a line. Each variable and function argument must be declared on a separate line. Do not use comma-separated lists to declare multiple identifiers.
- Do not put more than one statement on a line. The only exceptions are the **for** statement, where the initial, conditional, and loop statements can go on a single line:

  ```
  for (i = 0; i < count; i++)
  ```

  or the **switch** statement if the actions are short and nearly identical (see the **switch** statement format in *Indentation*, p.572).

  The **if** statement is not an exception: the executed statement always goes on a separate line from the conditional expression:

  ```
  if (i > count)
      i = count;
  ```

- Braces ({ and }) and **case** labels always have their own line.

Horizontal Spacing

- Put spaces around binary operators, after commas, and before an open parenthesis. Do not put spaces around structure members and pointer operators. Put spaces before open brackets of array subscripts; however, if a

subscript is only one or two characters long, the space can be omitted. For example:

```
status = fooGet (foo, i + 3, &value);
foo.index
pFoo->index
fooArray [(max + min) / 2]
string[0]
```

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

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

status = fooList (foo, a, b, c,
                  d, e);

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

Indentation

- Indentation levels are every four characters (columns 1, 5, 9, 13, …).
- The module and subroutine headings and the subroutine declarations start in column one.
- Indent one indentation level after:
  - subroutine declarations
  - conditionals (see below)
  - looping constructs
  - switch statements
  - case labels
  - structure definitions in a **typedef**
- The **else** of a conditional has the same indentation as the corresponding **if**. Thus the form of the conditional is:

```
if ( condition )
    {
    statements
    }
else
    {
    statements
    }
```

The form of the conditional statement with an **else if** is:

```
if ( condition )
    {
    statements
    }
else if ( condition )
    {
    statements
    }
else
    {
    statements
    }
```

- The general form of the **switch** statement is:

```
switch ( input )
    {
    case 'a':
        ...
        break;
    case 'b':
        ...
        break;
    default:
        ...
        break;
    }
```

If the actions are very short and nearly identical in all cases, an alternate form of the switch statement is acceptable:

```
switch ( input )
    {
    case 'a': x = aVar; break;
    case 'b': x = bVar; break;
    case 'c': x = cVar; break;
    default: x = defaultVar; break;
    }
```

- Comments have the same indentation level as the section of code to which they refer (see *Comments,* p.574).
- Section braces ({ and }) have the same indentation as the code they enclose.

Comments

- Place comments within code so that they precede the section of code to which they refer and have the same level of indentation. Separate such comments from the code by a single blank line.
  - Begin single-line comments with the open-comment and end with the close-comment, as in the following:

```
/* This is the correct format for a single-line comment */

foo = MAX_FOO;
```

  - Begin and end multi-line comments with the open-comment and close-comment on separate lines, and precede each line of the comment with an asterisk (*), as in the following:

```
/*
 * This is the correct format for a multiline comment
 * in a section of code.
 */

foo = MIN_FOO;
```

- Compose multi-line comments in declarations and at the end of code statements with one or more one-line comments, opened and closed on the same line. For example:

```
int foo
    (
    int     a,      /* this is the correct format for a */
                    /* multiline comment in a declaration */
    BOOL    b       /* standard comment at the end of a line */
    )

    {
    day = night;    /* when necessary, a comment about a line */
                    /* of code can be done this way */
    }
```

I.3.5 C Naming Conventions

The following conventions define the standards for naming modules, routines, variables, constants, macros, types, and structure and union members. The purpose of these conventions is uniformity and readability of code.

- When creating names, remember that the code is written only once, but read many times. Assign names that are meaningful and readable; avoid obscure abbreviations.
- Names of routines, variables, and structure and union members are composed of upper- and lowercase characters and no underbars. Capitalize each "word" except the first:

  aVariableName

- Names of defined types (defined with **typedef**), and constants and macros (defined with **#define**), are all uppercase with underbars separating the words in the name:

  A_CONSTANT_VALUE

- Every module has a short prefix (two to five characters). The prefix is attached to the module name and all externally available routines, variables, constants, macros, and **typedefs**. (Names not available externally do not follow this convention.)

fooLib.c	module name
fooObjFind	subroutine name
fooCount	variable name
FOO_MAX_COUNT	constant
FOO_NODE	type

- Names of routines follow the *module-noun-verb* rule. Start the routine name with the module prefix, followed by the noun or object that the routine manipulates. Conclude the name with the verb or action the routine performs:

fooObjFind	foo - object - find
sysNvRamGet	system - NVRAM - get
taskSwitchHookAdd	task - switch hook - add

- Every header file defines a preprocessor symbol that prevents the file from being included more than once. This symbol is formed from the header file name by prefixing **__INC** and removing the dot (**.**). For example, if the header file is called **fooLib.h**, the *multiple inclusion guard symbol* is:

  __INCfooLibh

- Pointer variable names have the prefix *p* for each level of indirection. For example:

```
FOO_NODE *      pFooNode;
FOO_NODE **     ppFooNode;
FOO_NODE ***    pppFooNode;
```

I.3.6 C Style

The following conventions define additional standards of programming style:

- **Comments:** Insufficiently commented code is unacceptable.
- **Numeric Constants:** Use **#define** to define meaningful names for constants. Do not use numeric constants in code or declarations (except for obvious uses of small constants like 0 and 1).
- **Boolean Tests:** Do not test non-booleans as you test a boolean. For example, where **x** is an integer:

  CORRECT: `if (x == 0)`

  INCORRECT: `if (! x)`

  Similarly, do not test booleans as non-booleans. For example, where **libInstalled** is declared as **BOOL**:

  CORRECT: `if (libInstalled)`

  INCORRECT: `if (libInstalled == TRUE)`

- **Private Interfaces:** Private interfaces are functions and data that are internal to an application or library and do not form part of the intended external user interface. Place private interfaces in a header file residing in a directory named **private**. End the name of the header file with an uppercase *P* (for *private*). For example, the private function prototypes and data for the commonly used internal functions in the library **blahLib** would be placed in the file **private/private/blahLibP.h**.
- **Passing and Returning Structures:** Always pass and return pointers to structures. Never pass or return structures directly.
- **Return Status Values:** Routines that return status values should return either **OK** or **ERROR** (defined in **vxWorks.h**). The specific type of error is identified by setting **errno**. Routines that do not return any values should return **void**.
- **Use Defined Names:** Use the names defined in **vxWorks.h** wherever possible. In particular, note the following definitions:
  - Use **TRUE** and **FALSE** for boolean assignment.
  - Use **EOS** for end-of-string tests.
  - Use **NULL** for zero pointer tests.
  - Use **IMPORT** for **extern** variables.
  - Use **LOCAL** for **static** variables.
  - Use **FUNCPTR** or **VOIDFUNCPTR** for pointer-to-function types.

I.3.7 C Header File Layout

Header files, denoted by a **.h** extension, contain definitions of status codes, type definitions, function prototypes, and other declarations that are to be used (through **#include**) by one or more modules. In common with other files, header files must have a *standard file heading* at the top. The conventions in this section define the header file contents that follow the standard file heading.

Structural

The following structural conventions ensure that generic header files can be used in as wide a range of circumstances as possible, without running into problems associated with multiple inclusion or differences between ANSI C and C++.

- To ensure that a header file is not included more than once, the following must bracket all code in the header file. This follows the standard file heading, with the **#endif** appearing on the last line in the file.

```
#ifndef __INCfooLibh
#define __INCfooLibh
    ...
#endif /* __INCfooLibh */
```

  See *I.3.5 C Naming Conventions*, p.574, for the convention for naming preprocessor symbols used to prevent multiple inclusion.

- To ensure C++ compatibility, header files that are compiled in both a C and C++ environment must use the following code as a nested bracket structure, subordinate to the statements defined above:

```
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
    ...
#ifdef __cplusplus
}
#endif /* __cplusplus */
```

Order of Declaration

The following order is recommended for declarations within a header file:

(1) Statements that include other header files.

(2) Simple defines of such items as error status codes and macro definitions.

(3) Type definitions.

(4) Function prototype declarations.

Example I-4 **Sample C Header File**

The following header file demonstrates the conventions described above:

```
/* bootLib.h - boot support subroutine library */

/* Copyright 1984-1993 Wind River Systems, Inc. */

/*
modification history
--------------------
01g,22sep92,rrr  added support for c++.
01f,04jul92,jcf  cleaned up.
01e,26may92,rrr  the tree shuffle.
01d,04oct91,rrr  passed through the ansification filter,
                 -changed VOID to void
                 -changed copyright notice
01c,05oct90,shl  added ANSI function prototypes;
                 added copyright notice.
01b,10aug90,dnw  added declaration of bootParamsErrorPrint().
01a,18jul90,dnw  written.
*/

#ifndef __INCbootLibh
#define __INCbootLibh
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

/*
 * BOOT_PARAMS is a structure containing all the fields of the
 * VxWorks boot line. The routines in bootLib convert this structure
 * to and from the boot line ASCII string.
 */

/* defines */

#define BOOT_DEV_LEN            20      /* max chars in device name */
#define BOOT_HOST_LEN           20      /* max chars in host name */
#define BOOT_ADDR_LEN           30      /* max chars in net addr */
#define BOOT_FILE_LEN           80      /* max chars in file name */
#define BOOT_USR_LEN            20      /* max chars in user name */
#define BOOT_PASSWORD_LEN       20      /* max chars in password */
#define BOOT_OTHER_LEN          80      /* max chars in "other" field */
#define BOOT_FIELD_LEN          80      /* max chars in boot field */

/* typedefs */

typedef struct bootParams               /* BOOT_PARAMS */
    {
    char bootDev [BOOT_DEV_LEN];        /* boot device code */
```

```
    char hostName [BOOT_HOST_LEN];      /* name of host */
    char targetName [BOOT_HOST_LEN];    /* name of target */
    char ead [BOOT_ADDR_LEN];           /* ethernet internet addr */
    char bad [BOOT_ADDR_LEN];           /* backplane internet addr */
    char had [BOOT_ADDR_LEN];           /* host internet addr */
    char gad [BOOT_ADDR_LEN];           /* gateway internet addr */
    char bootFile [BOOT_FILE_LEN];      /* name of boot file */
    char startupScript [BOOT_FILE_LEN]; /* name of startup script */
    char usr [BOOT_USR_LEN];            /* user name */
    char passwd [BOOT_PASSWORD_LEN];    /* password */
    char other [BOOT_OTHER_LEN];        /* avail to application */
    int  procNum;                       /* processor number */
    int  flags;                         /* configuration flags */
    } BOOT_PARAMS;

/* function declarations */

extern STATUS bootBpAnchorExtract (char * string, char ** pAnchorAdrs);
extern STATUS bootNetmaskExtract (char * string, int * pNetmask);
extern STATUS bootScanNum (char ** ppString, int * pValue, BOOL hex);
extern STATUS bootStructToString (char * paramString, BOOT_PARAMS *
                                  pBootParams);
extern char * bootStringToStruct (char * bootString, BOOT_PARAMS *
                                  pBootParams);
extern void   bootParamsErrorPrint (char * bootString, char * pError);
extern void   bootParamsPrompt (char * string);
extern void   bootParamsShow (char * paramString);

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* __INCbootLibh */
```


I.3.8 Documentation Format Conventions for C

This section specifies the text-formatting conventions for source-code derived documentation. The WRS tool **refgen** is used to generate reference entries (in HTML format) for every module automatically. All modules must be able to generate valid reference entries. This section is a summary of basic documentation format issues; for a more detailed discussion, see the *Tornado BSP Developer's Kit User's Guide: Documentation Guidelines*.

Layout

To work with **refgen**, the documentation in source modules must be laid out following a few simple principles. The file **sample.c** in

installDir/**target/unsupported/tools/mangen** provides an example and more information.

Lines of text should fill out the full line length (assume about 75 characters); do not start every sentence on a new line.

Format Commands

Documentation in source modules can be formatted with UNIX **nroff**/**troff** formatting commands, including the standard **man** macros and several WRS extensions to the **man** macros. Some examples are described in the sections below. Such commands should be used sparingly.

Any macro (or "dot command") must appear on a line by itself, and the dot (**.**) must be the first character on the logical line (in the case of subroutines, this is column 3, because subroutine comment sections begin each line with an asterisk plus a space character).

Special Elements

- **Parameters:** When referring to a parameter in text, surround the name with the angle brackets, **<** and **>**. For example, if the routine ***getName*( )** had the following declaration:

```
void getName
    (
    int     tid,    /* task ID */
    char *  pTname  /* task name */
    )
```

  You might write something like the following:

```
This routine gets the name associated with a specified task ID and copies
it to <pTname>.
```

- **Subroutines:** Include parentheses with all subroutine names, even those generally construed as shell commands. Do not put a space between the parentheses or after the name (unlike the WRS convention for code):

  CORRECT: **taskSpawn()**

  INCORRECT: **taskSpawn (), taskSpawn( ), taskSpawn**

Note that there is one major exception to this rule. In the subroutine title, do not include the parentheses in the name of the subroutine being defined:

CORRECT:
```
/*********************************************
*
* xxxFunc - do such and such
```

INCORRECT:
```
/*********************************************
*
* xxxFunc() - do such and such
```

Avoid using a library, driver, or routine name as the first word in a sentence, but if you must, do not capitalize it.

- **Terminal Keys:** Enter the names of terminal keys in all uppercase, as in **TAB** or **ESC**. Prefix the names of control characters with **CTRL+**; for example, **CTRL+C**.
- **References to Publications:** References to chapters of publications should take the form *Title*: *Chapter*. For example, you might say:

  For more information, see the *VxWorks Programmer's Guide: I/O System*.

  References to documentation volumes should be set off in italics. For general cases, use the **.I** macro. However, in SEE ALSO sections, use the **.pG** and **.tG** macros for the *VxWorks Programmer's Guide* and *Tornado User's Guide*, respectively.
- **Section-Number Cross-References:** Do not use the UNIX parentheses-plus-number scheme to cross-reference the documentation sections for libraries and routines:

  CORRECT: **sysLib, vxTas()**

  INCORRECT: **sysLib(1), vxTas(2)**

Table I-1 **Format of Special Elements**

Component	Input	Output (mangen + troff)
library in title	**sysLib.c**	sysLib
library in text	**sysLib**	(same)
subroutine in title	**sysMemTop**	sysMemTop()
subroutine in text	**sysMemTop()**	(same)
subroutine parameter	**<ptid>**	(same)

Table I-1 **Format of Special Elements**

Component	Input	Output (mangen + troff)
terminal key	`TAB, ESC, CTRL+C`	(same)
publication	`.I "Tornado User's Guide"`	*Tornado User's Guide*
VxWorks Programmer's Guide in **SEE ALSO**	`.pG "Configuration"`	*VxWorks Programmer's Guide: Configuration*
Tornado User's Guide in **SEE ALSO**	`.tG "Cross-Development"`	*Tornado User's Guide: Shell*
emphasis	`\f2must\fP`	*must*

Formatting Displays

- **Code:** Use the **.CS** and **.CE** macros for displays of code or terminal input/output. Introduce the display with the **.CS** macro; end the display with **.CE**. Indent such displays by four spaces from the left margin. For example:

```
* .CS
*     struct stat statStruct;
*     fd = open ("file", READ);
*     status = ioctl (fd, FIOFSTATGET, &statStruct);
* .CE
```

- **Board Diagrams:** Use the **.bS** and **.bE** macros to display board diagrams under the BOARD LAYOUT heading in the **target.nr** module for a BSP. Introduce the display with the **.bS** macro; end the display with **.bE**.

- **Tables:** Tables built with **tbl** are easy as long as you stick to basics, which suffice in almost all cases. Tables always start with the **.TS** macro and end with a **.TE**. The **.TS** should be followed immediately by a line of options terminated by a semicolon (**;**); then by one or more lines of column specification commands followed by a dot (**.**). For more details on table commands, refer to any UNIX documentation on **tbl**. The following is a basic example:

```
.TS
center; tab(|);
lf3 lf3
l l.
Command         | Op Code
_
INQUIRY         | (0x12)
```

```
REQUEST SENSE   | (0x03)
TEST UNIT READY | (0x00)
.TE
```

General stylistic considerations are as follows:

- Redefine the tab character using the **tab** option; keyboard tabs cannot be used by **tbl** tables. Typically the pipe character (|) is used.
- Center small tables on the page.
- Expand wide tables to the current line length.
- Make column headings bold.
- Separate column headings from the table body with a single line.
- Align columns visually.

Do not use **.CS**/**.CE** to build tables. This markup is reserved for code examples.

- **Lists:** List items are easily created using the standard **man** macro **.IP**. Do not use the **.CS**/**.CE** macros to build lists. The following is a basic example:

```
.IP "FIODISKFORMAT"
Formats the entire disk with appropriate hardware track and
sector marks.
.IP "FIODISKINIT"
Initializes a DOS file system on the disk volume.
```