Wind River VxWorks Coding Style Guide

Document ID:	VxWorksCodingStyleGuide
Author:	Mati Sauks
Version:	1.01
Version Date:	17 Aug 2019
Status:	Approved

Corporate Headquarters
Wind River
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.
Toll free (U.S.A.): +1-800-545-WIND
Telephone: +1 510 748 4100
Facsimile: +1 510 749 2010

For additional contact information, see the Wind River website:
http://www.windriver.com

For information on how to contact Customer Support, see:
http://www.windriver.com/support

Revision History

Date	Version	By	Description of Change
Mar 03, 2018	0.01	Mati Sauks	Created from VxWorks Coding Standard 2014, also addressing JIRA defects VXWCS-1 to VXWCS-44
Jan 09, 2019	0.02	Mati Sauks	Update from technical review
Apr 27, 2019	0.03	Mati Sauks	Update from editorial review
Apr 29, 2019	1.00	Mati Sauks	Mark approved
Aug 17, 2019	1.01	Mati Sauks	Replace doxygen with apigen markup, JIRA defect VXWCS-80

Table of Contents

1 Introduction 1
1.1 Terminology 1
1.2 Deviations 1
1.3 Legacy Code 2
1.4 Applicable Documents 2

2 Acronyms 3

3 File Format, Encoding, and Heading 4
3.1 File Format and Encoding 4
3.1.1 Text File Format 4
3.1.2 Standard File Encoding 4
3.2 Standard File Heading 5
3.2.1 Title 5
3.2.2 Copyright for Wind River Proprietary Code 5
3.2.3 Copyright for Open Source Code 6
3.2.4 Modification History 6
3.2.5 File Heading Example for C Source Files 12

4 C Coding Style 13
4.1 C Module Layout 13
4.1.1 General Module Documentation 13
4.1.2 Includes 13
4.1.3 Defines 14
4.1.4 Typedefs 14
4.1.5 Globals 14
4.1.6 Locals 14
4.1.7 Forward Declarations 14
4.1.8 C Module Header Example 15
4.2 C Function Layout 16
4.2.1 Banner 16
4.2.2 Title 17
4.2.3 Description 17
4.2.4 Parameters 17
4.2.5 Returns 17
4.2.6 Error Number 17
4.2.7 See Also 18
4.2.8 Examples 18
4.2.9 Standard C Function Layout 18
4.2.10 C Declaration Formats 19
4.2.11 Variables and Types 19
4.2.12 Functions 22
4.2.13 Documenting Macros 22
4.3 Code Formatting and Style 23
4.3.1 Vertical Spacing 23

4.3.2 Horizontal Spacing 24
4.3.3 Indentation 24
4.3.4 Comments 27
4.3.5 Multi-line Quoted Strings 28
4.4 C Naming Conventions 29
4.4.2 errno Codes 30
4.4.3 Special Macros 30
4.4.4 Module Names 30
4.4.5 Function and Variable Names 32
4.4.6 Private, Protected, and Public Interfaces 32
4.4.7 extern keyword usage 33
4.5 C Header File Layout 33
4.5.1 Structural 33
4.5.2 Order of Declaration 34
4.5.3 Standard C Header File Layout 34
4.5.4 C Header File Inclusion Rules 36

5 Assembler Coding Style 38

5.1 Assembler Module Layout 38
5.2 Assembler Function Layout 39
5.3 Assembler Comment Style 39
5.4 Assembler Formatting and Style 39

6 Tcl Coding Style 40

6.1 Tcl Module Layout 40
6.2 General Module Documentation 40
6.2.1 Globals 40
6.2.2 Tcl File and Module Headings 41
6.2.3 Tcl Procedure Layout 41
6.2.4 Tcl Code Outside Procedures 44
6.2.5 Tcl Declaration Formats 45
6.2.6 Code Layout 45
6.2.7 Tcl Style 49

7 C++ Coding Style 52

8 Python Coding Style 54

8.1 Standard File Encoding 54
8.2 Python Module Layout 54
8.3 General Module Documentation 54
8.3.1 Environment 54
8.3.2 Main docstring block 55
8.3.3 Modification History block 55
8.3.4 File Heading Example for Python Files 55
8.4 Code Layout 56
8.4.1 Indentation 56
8.4.2 Maximum line length 56
8.4.3 Blank lines 58
8.5 Imports 58

8.6 Single and double quotes 59
8.7 Comments 59
8.8 Naming 60
8.8.1 Naming styles 60
8.8.2 Single letter variable names 60
8.8.3 ASCII Identifier names 61
8.8.4 "Dunder" Names 61
8.8.5 Documentation String (docstring) 61
8.8.6 Class Layout 62
8.8.7 Function Layout 63
8.8.8 Miscellaneous 64
8.8.9 Comments 66
8.9 Exceptions 67
8.9.1 Catch specific exceptions 67
8.9.2 Process exceptions explicitly 68
8.9.3 Derive exceptions from Exception, not BaseException 68

9 Component Description Language 69

9.1 SYNOPSIS Section 69
9.2 Naming 70

10 APIGEN Reference Standards 71

10.1 Introduction 71
10.2 Content and Organization 71
10.3 NAME Section 73
10.3.1 C Libraries 73
10.3.2 C Routines 73
10.3.3 Tcl Procedures, Scripts and Commands 74
10.4 ROUTINES Section 74
10.5 SYNOPSIS Section 74
10.5.1 C Routines 74
10.5.2 Tcl Procedures, Scripts and Commands 74
10.6 DESCRIPTION Section 75
10.6.1 Parameter Lists 75
10.7 Additional Optional Sections 77
10.8 SMP CONSIDERATIONS 77
10.9 64-bit CONSIDERATIONS 78
10.10 EXAMPLES 78
10.11 INCLUDE FILES 78
10.12 RETURNS 78
10.13 ERRNO or ERRORS 79
10.14 SEE ALSO 79
10.15 DEPRECATED 80
10.16 Format and Style 80
10.17 Headings 82
10.18 Special Words 83
10.18.1 Literal Names of Commands, Global Variables, Files and Other Elements 83
10.18.2 Terminal Keys 84

10.18.3 Routine Names ... 84
10.18.4 Placeholder Text ... 85
10.18.5 Parameters ... 85
10.18.6 Book References ... 86
10.18.7 Cross References to Other Reference Entries ... 86
10.18.8 Special Terms ... 86
10.18.9 Emphasis ... 86
10.19 Lists and Tables ... 87
10.19.1 Short Word Lists ... 87
10.19.2 Item Lists (Definition Lists or Term Lists) ... 88
10.19.3 Marker Lists (Dash or Numbered Lists) ... 88
10.19.4 Tables ... 89
10.20 Code Examples, Syntax Display and Diagrams ... 90
10.20.1 Code Examples ... 90
10.20.2 Command Syntax ... 91
10.20.3 Board Diagrams ... 91
10.20.4 Graphics ... 91
10.21 Directives ... 91
10.21.1 Blocking Text From Publication ... 93
10.21.2 Conditionals ... 95
10.21.3 Other Overrides ... 95
10.21.4 Including Content From Other Files ... 96

11 Versioning in VxWorks ... 99
11.1 Incrementing the Least Significant Digit (x.x.x.X) ... 99
11.2 Incrementing the Third Most Significant Digit (x.x.X.x) ... 100
11.3 Incrementing the Second Most Significant Digit (x.X.x.x) ... 100
11.4 Incrementing the Most Significant Digit (X.x.x.x) ... 101
11.5 Versioning for 3'rd Party Content ... 101
11.6 Versioning for "Unsupported" Content ... 101
11.7 Versioning of VxWorks 7 Product Variants ... 102

List of Tables

Table 1-1 Applicable Documents........2
Table 4-1 Example of naming inside a module........30
Table 4-2 Function Name Examples........32
Table 10-1 Required Sections for APIGEN markup........72
Table 10-2 Apigen Markup........82
Table 10-3Special Text Element Handling........87
Table 10-4 Summary of apigen Directives........93

1 Introduction

This document defines the VxWorks coding style conventions for all C, C++, Python, assembler, Tcl and CDL code.

The VxWorks Coding Rules Guide document provides additional rules covering coding best practices that must be followed.

This document also defines the VxWorks API Reference standard for the documentation comments embedded in VxWorks C, assembler and Tcl code. All VxWorks-related code must conform to this standard because Wind River uses the APIGEN tool to generate reference content.

The conventions are intended, in part, to encourage higher quality code; every source module is required to have certain essential documentation, and code and documentation are 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. Uniformity also allows automated processing of the source to generate reference documentation. For this reason, documentation standards are also described.

While this document describes coding conventions explicitly for C, assembler and Tcl, in practice these conventions are applied broadly to code written in other languages, such as C++ and Java, as well as shellscripts, Perl scripts, and makefiles.

1.1 Terminology

This document uses specific terms to indicate the necessity of the given set of rules. The words below have special meanings throughout the document:

- Shall or must – Indicates that the rule must always be followed, unless a deviation is requested and approved.
- Avoid or should – Indicates that the rule is a recommendation for good practice, but can be deviated without approval as needed.

1.2 Deviations

Deviations from these standards shall be permitted only if **all** the following conditions are met:

1. An architect or technical authority shall approve the deviation.

2. The justifications of the deviations are noted as code comments, preferably at the points of divergence.

1.3 Legacy Code

Some of the code in VxWorks was written before this standard existed, and thus may not comply with some of the rules in this document. The code for a defect fix should follow the style of the original code. All newly written code shall follow the styles in this document.

“New code” does not apply to adding functions or defect fixes to existing libraries.

1.4 Applicable Documents

The table below lists all documents referenced within this document.

Ref.	Title
Ref.1	Wind River VxWorks Coding Rules Guide
Ref.2	Wind River Style Guide for Developers (https://jive.windriver.com/docs/DOC-66941)

Table 1-1 Applicable Documents

2 Acronyms

API	Application Program Interface
APIGEN	Application Program Interface document Generator
ANSI	American National Standards Institute
ASCII	American Standard Code Information Interchange
BSP	Board Support Package
CDF	Component Description File
CDL	Component Description Language
DTS	Device Tree Specification
ED&R	Error Detection and Reporting
HLD	High Level Design
HTML	HyperText Markup Language
ID	Identifier
IP	Intellectual Property
K&R	Kernighan and Ritchie
POSIX	Portable Operating System Interface for Unix
RPM	Red Hat Package Manager
SMP	Symmetric Multiprocessing
SQA	Software Quality Assurance
TCL	Tool Command Language
UNIX	[not an acronym] UNIX is a pun on MULTICS
UTF	Unicode Text File
VSB	VxWorks Source Build
WR	Wind River

3 File Format, Encoding, and Heading

Text-based files in VxWorks, such as header files, resource files, files that implement host tools, library functions or applications, must follow specific formats and encodings, and must contain specific standard file headings. The conventions in this section define these standards.

3.1 File Format and Encoding

3.1.1 Text File Format

A text file must use UNIX file format.

If you are using Windows for development, you can have git automatically convert files to the UNIX format using the following commands in any git repository. Because of the '--local' option, it will only affect that specific repository. If you use the '--global' option, it will affect all git repositories that you commit to.

For Windows:

```
git config --local core.autocrlf true
git config --local core.safecrlf false
git config --local core.filemode false
git config --local core.longpaths true
```

Do NOT use this configuration under Linux.

Text file format can be ascertained under Linux by using the ‘file’ command. Note that “with CRLF line terminators” indicates it is NOT in UNIX file format.

```
>file foo.txt errors.txt
foo.txt:    ASCII text
errors.txt: ASCII text, with CRLF line terminators
```

3.1.2 Standard File Encoding

A text file must use ASCII encoding with the exception that multi-byte characters, in UTF-8 format, for example, can be utilized in strings destined to be output on some device capable of displaying/rendering such multi-byte characters, or HTML sequences that will be fed to a web browser.

3.2 Standard File Heading

The file heading consists of the comment blocks described below. The blocks are separated by one or more empty lines (in scripting languages, empty comment lines) and contain no empty lines within the block. This facilitates automated processing. The example in section [3.2.5] shows a standard file heading from a C source file. The examples shown use Wind River APIGEN markup as described in chapter [10].

3.2.1 Title

A one-line comment containing the tool, library, or application name followed by a short description. The name must be the same as the filename. This line will become the title in generated reference entries and indexes.

For example:

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

3.2.2 Copyright for Wind River Proprietary Code

The copyright notice for all proprietary Wind River source code—that is, code that Wind River does not intend to release as open source—is a multi-line comment containing the appropriate copyright information and a legal notice. It is mandatory that this specified copyright notice be used.

The initial line of the copyright includes a date expressed as a single year, a hyphenated range of years, a comma-separated list of years, or a combination of comma-separated and hyphenated dates. For example, any of the following entries are considered valid: “1997”, “1992-2003”, “2012-2013” “1997, 2001”, or “1997, 2002-2005”.

As a general practice, whenever a Wind River source file is modified, the copyright date should be altered to include the current year. The first year is the date the file was originally created. Additional years or year ranges indicate the date the file was updated; they must be modified as appropriate whenever the file is updated. For example, if a source file was created and last modified in 1997, the date shown is a single entry, “1997”. If that file was then modified again in 2004, the date entry is “1997, 2004”. If the file was created in 1997 and modified in 2002, 2003, and 2004, the date entry is “1997, 2002-2004” (modifications during two or more consecutive years can be included as a hyphenated entry).

The text for the legal notice is boilerplate:

```
/*
 * Copyright (c) 1984, 1996-1997, 2014 Wind River Systems, Inc.
 *
 * The right to copy, distribute, modify, or otherwise make use
 * of this software may be licensed only pursuant to the terms
 * of an applicable Wind River license agreement.
 */
```

NOTE: Early code contains copyright statements that do not conform to this standard. Developers should update copyrights to satisfy these standards if editing the file.

Wind River legacy code may include the file copyright_wrs.h. This include should be removed.

Legacy assembly code may contain the following code fragment that should be removed:

```
.data
.globl copyright_wind_river
.long copyright_wind_river
```

3.2.3 Copyright for Open Source Code

Code that is intended for release as open source should still include a single-line copyright notice as follows (the date entry follows the same conventions described above):

```
/* Copyright (c) 2002-2005 Wind River Systems, Inc. */
```

3.2.3.1 DTS Files

DTS files do not require a copyright.

Existing DTS files from semi-conductor-vendors and any other source can be used as reference material in creating our own DTS files for inclusion in VxWorks.

Follow the rules in this document relating to braces, spaces, and indenting for all DTS files produced.

In the rare case that an open source DTS file is used and it contains a copyright, it is subject to normal IP disclosure/review. Use of open source DTS files should be avoided if possible.

The modification history section for DTS files must be maintained.

3.2.4 Modification History

The modification history should take the for of a multi-line comment in C, and a series of commented lines in scripting languages. Each entry in the modification history consists of:

1. The date of modification in the format 2 digit day, 3 letter month, 2 digit year, comma
2. Trigram of the programmer who oversaw the changes followed by 2 spaces
3. A complete description of the change. If the modification fixes a defect or addresses a user story, then the modification history must include the defect ID, feature ID or user story ID at the end of the entry, in parenthesis, without a following period. In general, use a Rally feature ID instead of a user story ID (unless the feature was some major feature that covered a very large scope).

In the cases where the defect was discovered during software development, and not present in shipped software, the defect ID(s) should be omitted from the modification history. Rally feature IDs should never be omitted.

For JIRA defects, only published defects should be referenced.

For code committed to git, commit messages should always contain a reference to the defect ID, feature ID or user story ID as well, and "git squash" and/or "git rebase" should be used to minimize the number of commits where possible.

Modification history entries must be added for functional changes and defect fixes that are customer visible (e.g., product source code containing the defect has already been released). Interim changes of a minor nature do not necessitate a separate entry in the modification history. For example, if a layer version change is required, a modification history entry is also required. Conversely, an edit to clarify a comment, fix a typo, or refactor unreleased code would not require a modification history line. Discresion is left to the developer and reviewers' judgement, but the deciding factor should be effective, complete, and accurate customer communication.

In cases where a developer makes an edit that is judged to not require a modification history, the copyright dates should still be correct and up to date.

A modification history entry should only apply to a single given feature or defect fix. The first time a file is updated to work on a given feature, the modification entry should be added. Any subsequent updates to the file for the same feature/fix should not result in another modification history entry.

In prior versions of the coding standard, the modification history required the inclusion of the version number. The version number was a two-digit number and a letter (for example, 03c). This is no longer required, and new edits to the file should omit this, and remove the version from old history lines.

The modification history list must remain in chronological order, the most recent history entry first. No attempt should be made to "resort" existing entries. They should be left in unchronological order.

The following is an example modification history:

```
/*
modification history
--------------------
01oct14,rdl  update coding standard for test file format (VXW7-1299)
15sep97,nfs  added defines MAX_FOOS and MIN_FATS.
15feb96,dnw  added functions fooGet() and fooPut();
             added check for invalid index in fooFind().
10feb84,dnw  written.
*/
```

3.2.4.1 Modification History Entries for .spec and .vsbl files

The guidance for the content of .spec and .vxbl history entries should describe what has changed to the meta-data in the file as opposed to what has changed to the source code in the layer.

For .spec files, in order to prevent redundant entries pertaining to “%changelog” see related updates in section [3.2.4.2]. A single modification history entry can be used to summarize the various additions to the “changelog” for a given release. For example, instead of providing the following set of modification history entries pertaining to “changelog” additions for version 2.0.1.0:

```
Modification history
--------------------
31jan19,pbs  update changelog 2.0.1.0 (F10573)
29jan19,syc  update changelog 2.0.1.0 (F2122)
29jan19,swu  update changelog 2.0.1.0 (V7COR-6600)
28jan19,wap  update changelog 2.0.1.0 (V7COR-6559)
25jan19,wap  update changelog 2.0.1.0 (V7COR-6581)
25jan28,lyx  update changelog 2.0.1.0 (F11456)
25jan19,wap  update changelog 2.0.1.0 (V7COR-3581)
14jan19,j_x  update changelog 2.0.1.0 (F7372)
14jan19,wap  update changelog 2.0.1.0 (V7COR-6561)
07jan19,wbe  update changelog 2.0.1.0 (F8913)
04jan19,syc  update changelog 2.0.1.0 (F2122)
10dec18,pfl  update changelog 2.0.1.0 (F11170)
10dec18,npc  update changelog 2.0.1.0 (V7PRO-5130)
10dec18,c_l  update changelog 2.0.1.0 (F10955)
06dec18,zjl  update changelog 2.0.1.0 (F11137)
03dec18,wap  update changelog 2.0.1.0 (V7COR-6499)
27nov18,syc  update changelog 2.0.1.0 (F2122)
07nov18,dlk  update changelog 2.0.1.0 (F10985)
24oct18,syc  update changelog 2.0.1.0 (WB4-7953)
24oct18,jxu  update changelog 2.0.1.0 (V7COR-6312)
23oct18,jdd  update changelog 2.0.1.0 (V7COR-6058)
22oct18,wap  update changelog 2.0.1.0 (V7COR-5274)
05oct18,h_k  update changelog 2.0.1.0 (F7233)
02oct18,dlk  update changelog 2.0.1.0 (F11081)
27sep18.wap  update changelog 2.0.1.0 (F10740)
27sep18,wap  update changelog 2.0.1.0 (V7COR-6239)
20sep18,g_x  update changelog 2.0.1.0 (F10693)
```

The following summary should be used instead:

```
Modification history
--------------------
31jan19,pbs  updated changelog for version 2.0.1.0 (F10693, F10740, F11081,
             F7233, F10985, F2122, F11137, F10955, F11170, F8913, F7372,
             F11456, F10573, defect fixes)
```

Also, the following summary is acceptable in order to restrict the modification history entry to 2 lines:

```
Modification history
--------------------
31jan19,pbs  updated changelog for version 2.0.1.0 (F10693, F10740, F11081,
             F7233, F10985, F2122, F11137, F10955, F11170, F8913
```

The date for the modification history entry shall be the date of the last addition to the %changelog section, and the trigram shall be of the engineer that made the last addition to the %changelog. This applies for the 2-line summary even when no additional features or defect IDs are added to the modification history content.

CORRECT:

```
28mar16,xyb  incremented version to 2.0.1.0; updated changelog(V7COR-3948)
```

INCORRECT:

```
28mar16,xyb  fix memory error and leak in rtpArgCopy()(V7COR-3948)
```

3.2.4.2 .spec Change Logs

RPM spec files shall also maintain a %changelog section to summarize the various changes to the content shipped via the RPM. The following format should be used for the %changelog section:

```
%changelog
* <datestamp> <packager> <version number>
- Overall description of change (Feature ID)
- [LAYER1]
  - description of change (Feature ID)
- [LAYER2]
  - description of defect resolved (Defect ID)
```

Where:
<datestamp> has same format as output by: $ date + "%a %b %d %Y"
<packager> is always "Wind River"; do not utilize trigrams!
<version number> the RPM version number

Examples:

```
%changelog
* Sat Aug 15 2015 Wind River 2.8.15.2
```

```
- Improve "help" related presentation of GSOAP related functionality in
Workbench (F3456)
- [CORE]
  - updated the layer 'HELP' field as presented in WorkBench (F3456)
- [BASE]
  - updated various SYNOPSIS fields in CDF files (F3456)
  - removed bogus increment of messages sent stats (V7COR-0001)

%changelog
* Mon Jun 15 2015 Wind River 1.0.2.0
- Added support for the Freescale i.MX 6 platform (fsl_imx6)(F4497)

%changelog
* Fri Sep 11 2015 Wind River 1.0.0.6
- [RPC]
  - fixed a security vulnerability in the Server-side rpc authenticator
    interface which allowed a maliciously crafted packet to cause an integer
    overflow with the possibility of executing remote code (V7STO-437)
```

Notes:

1. The layer name prefix shall be capitalized, e.g. use the same format as in the .vsbl files.
2. A layer name prefix shall be provided even if the RPM contains a single layer
3. If an RPM ships content that isn't a layer then omit the prefix. The second example above is from the VxWorks BootLoader RPM spec file, and the layer name prefix has been omitted.
4. New entries go at the top, i.e. changelog should grow up (just like the "Modification history" section
5. Defects that are not published or will not be published should not be used as a suffix to a changelog entry.

Important note: The changelog should never mention an RPM version that isn't actually released (i.e. an intermediate version). This can occur when multiple engineers make changes to content shipped by the RPM within a single release development cycle. For example: developer 'A' fixed a defect against RPM WB4 on Oct-5 and logged the following changelog:

```
%changelog
* Mon Oct 5 2015 Wind River 1.0.2.5
- [GUI]
  -fixed "help" related presentation of GSOAP related functionality in
Workbench (WB4-1234)
```

Later developer 'B' added a feature to the same RPM on Oct-9 and logged the following changelog:

```
%changelog
* Fri Oct 9 2015 Wind River 1.0.3.0
- [DEBUGGER]
  - added ARMv8 debugger support (F567)
```

Developer B should "squash" the above pair of changelog entries to the following since a customer will never be provided version 1.0.2.5 of this RPM:

```
%changelog
* Fri Oct 9 2015 Wind River 1.0.3.0
- [GUI]
  -fixed “help” related presentation of GSOAP related functionality in
Workbench (WB4-1234)
- [DEBUGGER]
  - added ARMv8 debugger support (F567)
```

Note that the date used for the squashed result is Oct. 9, i.e. the date of developer ‘B’'s changes.

3.2.4.3 .vsbl meta-data files

The modification history entry for layer meta-data files (.vsbl) should follow a similar guideline as with .spec files, the modification history entry should only describe what has changed in the .vsbl file itself; it should NOT provide significant detail as to the changes to content being delivered via the layer. It is acceptable and desirable to provide a very short description of the changed layer content and Rally feature identifier. The following is an example of a compliant .vsbl modification history entry:

```
 # 16mar16,aeg VERSION bumped to 1.2.2.0; added Quasimodo support (F8945)
```

3.2.5 File Heading Example for C Source Files

This is an example of a complete standard file heading, in C.

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

/*
 * Copyright (c) 1984, 1996-1997, 2014 Wind River Systems, Inc.
 *
 * The right to copy, distribute, modify, or otherwise make use
 * of this software may be licensed only pursuant to the terms
 * of an applicable Wind River license agreement.
 */

/*
modification history
--------------------
01oct14,rdl  update coding standard for test file format (VXW7-1299)
15sep97,nfs  added defines MAX_FOOS and MIN_FATS.
15feb96,dnw  added functions fooGet() and fooPut();
             added check for invalid index in fooFind().
10feb84,dnw  written.
*/
```

4 C Coding Style

This chapter defines the layout, style, comment and naming conventions for C code and header files.

The markup language used in the module and function header comments is processed by a tool named ‘APIGEN’ in order to generate API reference entries. For more information about the APIGEN tool, see the APIGEN markup in chapter [10].

4.1 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 blocks are included in the order shown, when applicable (exceptions to ordering are allowed in certain situations):

4.1.1 General Module Documentation

The module documentation block 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 required to be included by the users of the module.

An optional examples section ” * `EXAMPLES` “ should be included to help users properly use the module.

4.1.2 Includes

The include block consists of a one-line C comment containing the word *includes* followed by one or more C preprocessor `#include` directives. This block groups all header files included in the module in one place.

NOTE: In most cases, include files should be specified using angle brackets (< >) as illustrated in this example. However, double quotes may be used in order to force the compiler to start searching for the include file in the same directory as the source file.

4.1.3 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.

4.1.4 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.

4.1.5 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.

4.1.6 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.

4.1.7 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.

4.1.8 C Module Header Example

This is an example of a C module and file header.

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

/*
 * Copyright (c) 1984, 1996-1997, 2014 Wind River Systems, Inc.
 *
 * The right to copy, distribute, modify, or otherwise make use
 * of this software may be licensed only pursuant to the terms
 * of an applicable Wind River license agreement.
 */

/*
modification history
--------------------
01oct14,rdl  update coding standard for test file format (VXW7-1299)
15sep97,nfs  added defines MAX_FOOS and MIN_FATS.
15feb96,dnw  added functions fooGet() and fooPut();
             added check for invalid index in fooFind().
10feb84,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  112U  /* max # of foo entries */
#define MIN_FATS  2U    /* min # of FAT copies */


/* typedefs */

typedef struct fooMsg   /* FOO_MSG */
    {
    STATUS (*func) /* pointer to function to invoke */
        (
        int32_t arg1,
        int32_t arg2
        );
    int32_t arg [FOO_MAX_ARGS]; /* args for function */
    } FOO_MSG;

/* globals */

char *      pGlobalFoo; /* global foo table */

/* locals */

static uint32_t numFoosLost;   /* count of foos lost */

/* forward declarations */

static uint32_t fooMat (list * aList, int fooBar, BOOL doFoo);
FOO_MSG         fooNext (void);
STATUS          fooPut (FOO_MSG inPar);
```

4.2 C Function Layout

Each function is preceded by a comment heading. It must contain at least the following elements:

- Banner
- Title
- Description
- Parameters
- Returns
- Error Number
- See Also (optional)

For further information on the content and formatting of the above sections, see Content and Organization in section [10.2].

The function declaration immediately follows the function comment heading. There must be no text of any kind, including code and precompiler text, between the comment heading and the function declaration. If such text is needed, it must be placed before the comment heading.

The declaration is used for the documentation section when reference entries are generated automatically. The format of the function declaration is shown in C Declaration Formats in section [4.2.10].

4.2.1 Banner

This is the start of the comment block. The first line consists of a slash character (/) followed by 79 asterisks (*) across the page . This banner is useful as a yardstick to indicate the right margin, as all comments and code should be no wider than 80 characters. Each line for the remainder of the comment block must begin with a single asterisk in column 1, followed by a space. The comment heading is terminated by the end-of-comment character (*/), which must appear on a single line, starting in column 1.

4.2.2 Title

The title consists of a single line containing the function name, followed by a short, one-line description.

The function name and description are separated by a space character, a single hyphen, and another space character: “ - “. The function name in the title must match the declared function name, and should not have parentheses after the name. (Function names referenced elsewhere in documentation must include parentheses; see Function Names in section [4.2.12])

The description should convey action in an imperative mood. This line becomes the title in automatically generated reference entries and indexes. The title line must have a blank line (with a leading asterisk) immediately above and below it. The description portion is a sentence fragment; it must not start with a capital letter and must not end in a period. The title line must not continue on a new line, and it must not extend beyond the 80- character line limit. For details and examples, see NAME in section [10.3].

4.2.3 Description

A full description of what the function does and how to use it. The section title `DESCRIPTION` is optional (and generally omitted).

4.2.4 Parameters

Parameters to the function must be documented as part of the description in the section PARAMETERS. For additional examples see Parameter Lists in section [10.6.1].

4.2.5 Returns

The title RETURNS: followed by a description of the possible result values of the function. If there is no return value (as in the case of functions declared void), enter:

```
RETURNS: N/A
```

Mention only true function returns in this section—not values copied to a buffer given as an argument. Do not indent text that continues on separate lines.

4.2.6 Error Number

The title `ERRNO:` followed by all possible errno values set by the function. Each errno value should be followed by a one-line description ending with a period. Format lists of errno values as shown in this example.

```
ERRNO:
\is
\i S_objLib_OBJ_ID_ERROR
The <msgQId> is invalid.
\ie
```

For more information, see ERRNO or ERRORS in section [10.13].

4.2.7 See Also

The title `SEE ALSO:` followed on the next line by a comma-separated list of names of functions or libraries whose documentation may also be relevant. Do not list the name of the parent library, which is included automatically when the reference entry for this function is generated. See Also is not required; it is used on an as-needed basis.

4.2.8 Examples

An optional examples section `EXAMPLES` should be included to help convey proper use of the function.

4.2.9 Standard C Function Layout

This is an example of a C function header.

```
/*******************************************************************************
*
* fooGet - get an element from a foo
*
* This function finds the element of a specified <index> in a specified
* <foo>. The value of the element found is copied to <pValue>. The value can
* be gronk or blah value, it is the responsibility of the user to know what
* type is contained in foo.
*
* PARAMETERS
* \is
* \i <foo>
* [in] a foo structure array -- foo in which to find the element
*
* \i <index>
* [in] a positive integer -- element to be found in foo
*
* \i <* pValue>
* [out] any integer -- where to put value
* \ie
*
* RETURNS: OK, or ERROR if the element is not found or the index does not exist
*
* ERRNO:
* \is
* \i S_fooLib_BLAH
* The blah element was not found.
*
* \i S_fooLib_GRONK
* The gronk element was not found.
* \ie
*
* SEE ALSO: fooSet(), fooPrint()
*
* \INTERNAL
* fooGlobalLock - fooGet() takes the fooGlobalLock and may
*                 cause some latency as a result
*/

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

4.2.10 C Declaration Formats

Include only one declaration per line. Declarations are indented in accordance with Indentation [4.3.3], and are placed at the current indentation level. Data types, variable names, and comments should be column-aligned:

CORRECT:

```
FOO_NODE fooNode; /* foo node */
char     fooVal;  /* foo value */
```

INCORRECT:

```
FOO_NODE fooNode; /* foo node */
char fooVal; /* foo value */
```

4.2.11 Variables and Types

Integer typed variables should utilize the type definitions available in the header file stdint.h (introduced with C11). It is recommended that, where possible, the “exact-width integer types” be used such that source code is precise about the number of "bits of representation" that are required. The set of “exact-width integer types” are as follows:

```
int8_t
int16_t
int32_t
uint64_t
uint8_t
```

```
      uint16_t
      uint32_t
      int64_t
```

The stdint.h header file also contains type definitions for integers wide enough to hold a pointer value, namely, uintptr_t and intptr_t. The uintptr_t type is a unsigned integer type that has the property that any valid pointer can be converted to this type, and then converted back to a pointer without any loss of information, i.e. a comparison with the original pointer value will result in equality. The intptr_t type is a signed integer with the same property mentioned above for the uintptr_t type.

In situations where it's not possible to use any of the type definitions available in stdint.h, then the native C language types should be used, e.g. (unsigned) char, (unsigned) short, (unsigned) int, and (unsigned) long. These native C language types should be used instead of the VxWorks specific types for unsigned integer types available in vxTypes.h, e.g. uchar_t, ushort_t, uint_t, and ulong_t.

An example of a situation where it's not possible to use the "exact-width integer types", for example, involves code interfacing with existing (possibly standardized) APIs that do not use the "exact-width integer types". For example, an API that accepts an 'unsigned long' typed parameter cannot be called with a value that is typed as uint32_t nor uint64_t. Bear in mind that VxWorks source code, in general, needs to support both the ILP32 and LP64 data models, and the usage of type casting should be minimized.

For any type definitions that are introduced, the name should be suffixed with "_t" and the naming convention in section [4.4] should be followed.

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

```
uint32_t rootMemNBytes;  /* memory for TCB and root stack */
TASK_ID  rootTaskId;     /* root task ID */
BOOL     roundRobinOn;   /* boolean for round-robin mode */
```

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

CORRECT:
```
FOO_NODE *    pFooNode;  /* foo node pointer */
FOO_NODE **   ppFooNode; /* pointer to foo node ptr */
```

INCORRECT:
```
FOO_NODE     *pFooNode;  /* foo node pointer */
FOO_NODE    **ppFooNode; /* pointer to foo node ptr */
```

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 in section [4.3.4] 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 */
    uint_t    nSymbols;   /* current number of symbols in table */
    } symtab_t;
```

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

The exception to not 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;
```

or

```
typedef struct tcbInfo
    {
    struct fooInfo * pfooInfo;
    ...
    } tcbInfo_t;
```

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

INCORRECT:

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

4.2.12 Functions

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

For functions that take arguments, the function return type and name appear on the first line, the opening parenthesis on the next, followed by the arguments to the function, 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 functions that take no parameters, the word void in parentheses is required and appears on the same line as the function return type and name (with a single space after the name):

```
STATUS fppProbe (void)
```

NOTE: Some VxWorks header files still contain K&R style function prototypes. For example:

```
#if defined(__STDC__) || defined(__cplusplus)

extern STATUS     ptyDrv (void);

#else /*__STDC__*/

extern STATUS     ptyDrv ();

#endif /*__STDC__*/
```

Such constructs should be simplified to remove the K&R function declaration and the conditional preprocessor directives, as below:

```
extern STATUS     ptyDrv (void);
```

4.2.13 Documenting Macros

Normally, comments for macros should follow the standard practice for internal comments (see Comments in section [4.2]).

However, in some cases, it is desirable to publish macros as if they were callable functions. This can be done by giving the macro a standard function-style comment heading (see C Function Layout in section [4.2]) and including sections showing the calling syntax as if it were a C function. For example:

```
/*******************************************************************************
*
* assert - put diagnostics into programs (ANSI)
```

```
*
* SYNOPSIS
* void assert
*     (
*     expression
*     )
*
* DESCRIPTION
* If the C <expression> is false (that is, equal to zero), the assert() macro
* creates an error string which includes the expression, the source file name
* and the source line number. It then calls __assert().
*
* RETURNS: N/A
*
* ERRNO: N/A
*/

#define assert(exp) …
```

NOTE: When documenting macros using this method, the DESCRIPTION heading is not optional as it is in other cases. That is, when an explicit Synopsis section is included, it must also include an explicit DESCRIPTION heading before the description paragraph. For more information, see Content and Organization in section [10.2].

4.3 Code Formatting and Style

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

- vertical spacing
- horizontal spacing
- indentation
- comments
- multi-line quoted strings

4.3.1 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:

(1) the for statement, where the initial, conditional, and loop statements can go on a single line:

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

or (2) the switch statement if the actions are short and nearly identical (see the switch statement format in Indentation [4.3.3]).

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 a dedicated line.

4.3.2 Horizontal Spacing

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

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. Put a space after a type cast.

For example:

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

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))
...
```

4.3.3 Indentation

Indentation levels are every four characters. Indentation should be performed with spaces only.

For the viewing of legacy source code properly, the editor used should specify a tab stop of 8.

To implement this with 'vim', place the following into your .vimrc:

```
set tabstop = 8
set shiftwidth = 4
set softtabstop = 4
set expandtab
set autoindent
```

To implement this with 'emacs', set the following in your .emacs:

```
(defconst vx-c-style
  '((c-basic-offset . 4)
    (c-comment-only-line . 0)
    (tab-width . 8)
    (indent-tabs-mode . nil)
    (c-offsets-alist . ((topmost-intro-cont . 4)
                        (arglist-intro . c-lineup-arglist-close-under-paren)
                        (arglist-cont . c-lineup-arglist-close-under-paren)
                        (arglist-close . c-lineup-arglist-close-under-paren)
                        (brace-list-open . 4)
                        (brace-list-intro . 0)
                        (brace-list-close . 0)
                        (class-open . 4)
                        (inclass . 0)
                        (class-close . c-lineup-arglist-close-under-paren)
                        (statement-block-intro . 0)
                        (inextern-lang . 0)
                        (defun-open . 4)
                        (defun-block-intro . 0)
                        (block-open . 4))))
  "VxWorks C Programming Style")
(c-add-style "VxWorks" vx-c-style) )
```

The Linux command **expand** should be used to turn all existing tabs in a file into 8 spaces, which should result in a properly indented source file.

Module headings, function headings, and function declarations start in column one. Indent one indentation level after:

- function 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;
    }
```

Similarly for a simple one line if statement:

```
if (input) x = aVar;
```

Comments have the same indentation level as the section of code to which they apply (see Comments [4.3.4]).

Section braces '`{`' and '`}`' have the same indentation as the code they enclose.

4.3.4 Comments

Comments in code should allow a programmer who is unfamiliar with the code to follow what is happening and where it is happening. Any code fragment that could be construed as clever or unconventional should be documented clearly.

Code should not be "commented out" and the use of #if 0 and #if 1 to exclude/include code shall not be used.

The character sequences "/*" and "//" shall not be used within a comment, use "/@" and "@/" instead, where the '@' will be replaced by a '*' when documentation is generated.

C code shall not use the C++ "//" character sequence for comments.

Insufficiently commented code is unacceptable. Comments must be meaningful and helpful; for example, comments such as the following that state the obvious are an unnecessary distraction:

INCORRECT:

```
/* clear x */

x = 0;

/* reset myFoo */

fooReset (myFoo);
```

The following format rules apply to all comments in code:

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. The first word should not be captialized and the sentence should not end with a period 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 a space followed by an asterisk (*). The first word should be capitalized and the sentence ended with a period as in the following:

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

foo = MIN_FOO;
```

Compose comments in declarations with one or more one-line comments, opened and closed on the same line. The comments should be left-aligned. Declarations for all public functions must be fully commented. Example:

```
int32_t foo
    (
    int32_t  a,  /* this is the correct format for a */
                 /* multi-line comment in a declaration */
    BOOL     b   /* standard comment at the end of a line */
    )
```

Avoid using same-line comments with code statements unless absolutely necessary. Code statements should be preceded by full comment blocks.

CAUTION: Comments intended to divide the file into sections must not use the same banner used in function headings. Doing so confuses the tools used to generate reference entries. Instead, use some other device within the comment, such as a row of hyphens.

4.3.5 Multi-line Quoted Strings

In order to comply with the ANSI standard for C and C++, each string literal in a multi-line string must be terminated with a double quote before the end-of-line and started with another double quote on the next source code line. In this case, the compiler concatenates the adjacent strings into a single string. Multi-line strings that do not start and terminate with double quotes will not compile correctly using the current Wind River compilers. For example:

CORRECT:
```
char * pMultiLineString = "this is\n"
                          "a valid\n"
                          "multi-line string";
```

INCORRECT:
```
char * pMultiLineString = "this is
                          an invalid
                          multi-line string";
```

4.4 C Naming Conventions

The following conventions define the standards for naming files, functions, variables, constants, macros, types, and structure and union members. The first section describes the basic convention; subsequent sections provide details and describe exceptions.

When creating names, remember that the code is written only once, but read many times. Assign names that are meaningful and readable, but not overly long; avoid obscure abbreviations.

Identifiers must be sufficiently unique and distinct.

4.4.1.1 Basics

All naming follows either of two basic conventions, mixed case or all uppercase.

4.4.1.2 Mixed Case

The mixed-case convention applies to:

- filenames
- function names
- variables names
- structure and union members and tags
- goto labels

Names that follow the mixed-case convention contain upper- and lowercase letters and numbers (if necessary), but no underscores or any other symbols. Each “word” except the first begins with an uppercase letter.

Examples: taskLib.c, taskInit ( ), pResult

4.4.1.3 All Uppercase:

The all-uppercase convention applies to:

- constants
- macros
- defined types
- CDF components and parameters

Names that follow the all-uppercase convention contain uppercase letters, numbers, and underscores.

Names begin with an uppercase letter. “Words” are separated by underscores.

Example:

```
INCLUDE_NETWORK
```

4.4.2 errno Codes

An important variant of the all-uppercase convention is the format for Wind River-defined errno codes. These codes always begin with S_ followed by the library name in mixed case. The remainder of the name follows that standard all-uppercase convention.

Examples:

```
S_semLib_INVALID_STATE
```

```
S_dmsLib_DRIVER_UNKNOWN
```

4.4.3 Special Macros

Every header file defined a preprocessor symbol, called a multiple inclusion guard 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 fooLib.h, the multiple inclusion guard symbol is:

```
__INCfooLibh
```

4.4.4 Module Names

Every module has a short prefix (two to five characters). The prefix is attached to the module name and all externally available functions, variables, constants, macros, and typedefs. (Names not available externally do not follow this convention.) Avoid the use of non-alphanumeric characters in the filename.

Examples:

Name	**Type of Name**
fooLib.c	module
fooObjFind ()	function
fooCount	Variable
FOO_MAX_COUNT	constant
FOO_NODE	type

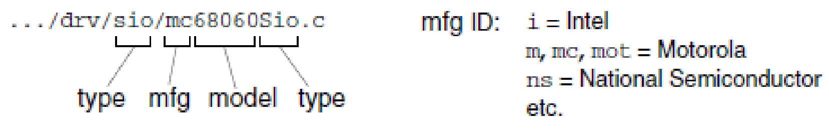
Table 4-1 Example of naming inside a module

Module names should end with a standard suffix. The most common are:

- Lib – for most modules

- Show – for functions that display internal data
- Drv – for most driver modules
- Sio – for serial I/O drivers

Drivers and their directory locations are named according to the convention shown in the following example:



4.4.5 Function and Variable Names

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

Function Name	**Interpretation**	**Module Name**
`fooObjFind ( )`	`foo - object - find`	`fooLib.c`
`sysNvRamGet ( )`	`system - NVRAM - get`	`sysLib.c`
`taskSwitchHookAdd ( )`	`task - switch hook – add`	`taskLib.c`

Table 4-2 Function Name Examples

Names of functions and variables visible outside the module must begin with the module's prefix.

Internal functions and variables that are not static should follow the following convention, either starting with a double underscore with a non-capitalized letter, or a single underscore character followed by a capital letter. For example:

```
__myJimmyGet ();
_MyJimmyGet ();
```

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

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

4.4.6 Private, Protected, and Public Interfaces

- Private interfaces are functions and data that are internal to a layer and do not form part of the layer's external user interface. Definitions of private interfaces shall be placed in a header file that remains in the layer's installation directory, i.e. the file nor the directory containing the private header file is specified in any of the macros available in a layer's Makefile for exporting header files. The private header will not be copied into the VSB project directory and thus will not be accessible by any source files outside of the layer itself.

- Protected interfaces are functions and data that are exported to a select set of layers known as "friends". Definitions of protected interfaces shall be placed in a header file that is exported by one of the following macros available to a layer's Makefile:

  ```
  KERNEL_PROTECTED_H_DIRS
  SHARED_PROTECTED_H_DIRS
  USER_PROTECTED_H_DIRS

  KERNEL_PROTECTED_H_FILES
  SHARED_PROTECTED_H_FILES
  USER_PROTECTED_H_FILES
  ```

In addition, the exporting layer shall use the FRIEND keyword in the layer.vsbl file to specify the set of friend layers. Header files specified in any of the above macros will be copied into the VSB project, but will only be accessible by the set of friend layers specified via the FRIEND keyword.

- Public interfaces are functions and data that are available to any other layer and to applications. Definitions of public interfaces shall be placed in a header file that is exported by one of the following macros available to a layer's Makefile:

```
KERNEL_PUBLIC_H_DIRS
SHARED_PUBLIC_H_DIRS
USER_PUBLIC_H_DIRS

KERNEL_PUBLIC_H_FILES
SHARED_PUBLIC_H_FILES
USER_PUBLIC_H_FILES
```

  Header files specified in any of the above macros will be copied into the VSB project, and will be accessible to all other source files.

4.4.7 extern keyword usage

All externs for a subsystem should be in the subsystem header files. Users of the subsystem should include the subsystem header file and NOT declare any subsystem externs in their own header file(s) or source file(s).

4.5 C Header File Layout

Header files, denoted by a .h file extension, contain definitions of status codes, type definitions, function prototypes, and other declarations that are to be used (through #header file inclusion) 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.

4.5.1 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 C Naming Conventions in section [4.4] 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:

The *cplusplus ifdef* is placed around external variable declarations and function prototypes, following the typedefs.

```
#ifdef __cplusplus
extern "C" {
#endif

/* external variables and function prototype declarations */

#ifdef __cplusplus
}
#endif /* __cplusplus */
```

4.5.2 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.

4.5.3 Standard C Header File Layout

This is an example of a C header file layout.

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

/*
 * Copyright (c) 1990-1992, 2018 Wind River Systems, Inc.
 *
 * The right to copy, distribute, modify, or otherwise make use
 * of this software may be licensed only pursuant to the terms
 * of an applicable Wind River license agreement.
 */
```

```
/*
modification history
--------------------
05may18,foo  remove version from mod hist (VXWCS-22)
22sep92,rrr  added support for c++.
04jul92,jcf  cleaned up.
26may92,rrr  the tree shuffle.
04oct91,rrr  passed through the ansification filter,
                 -changed VOID to void
                 -changed copyright notice
05oct90,shl  added ANSI function prototypes;
                 added copyright notice.
10aug90,dnw  added declaration of bootParamsErrorPrint ().
18jul90,dnw  written.
*/

#ifndef  __INCbootLibh
#define  __INCbootLibh

#include <sysInc.h>

/*
 * BOOT_PARAMS is a structure containing all the fields of the
 * VxWorks boot line. The functions 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 */
    uint32_t procNum;                   /* processor number */
    uint32_t flags;                     /* configuration flags */
    } BOOT_PARAMS;

#ifdef  __cplusplus
extern “C” {
#endif /*  __cplusplus */

/* function declarations */

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

#ifdef  __cplusplus
}
#endif /*  __cplusplus */

#endif /* __INCbootLibh */
```

4.5.4 C Header File Inclusion Rules

4.5.4.1 Explicit Inclusion

A source file must include header files explicitly if it uses the definitions provided by those headers.

The definitions provided by a header can be ambiguous. That is, a header can include other headers and definitions can be provided by the included headers. For example, vxWorks.h includes vxWorksCommon.h and it is valid to provide the definitions in vxWorksCommon.h by including vxWorks.h. However, in the example where spinLockLib.h includes time.h, it is not valid to use the definitions in time.h by including spinLockLib.h.

4.5.4.2 Header File Independence

Unless a header is always intended to be included with other header files (for example, vxWorksCommon.h), the header file must include all necessary header files such that it can be included alone from a C file without causing any compilation errors due to undefined elements such as macros, typedefs, and so forth.

4.5.4.3 Private Header Files

A public header file must not include private header files.

5 Assembler Coding Style

Note: Assembler files are pre-processed using the C compiler pre-processor.

5.1 Assembler 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 assembler source module following the standard file heading (one line description comment, copyright notice and modification history).

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 blocks are included in the order shown, if appropriate (exceptions to ordering are allowed in certain situations):

- **General Module Documentation:** The module documentation block is a C comment consisting of a complete description of the overall module purpose and function, especially the external interface. (See section [4.3.4] for C comment formatting)
- **Includes:** The include block consists of a one-line C comment containing the word *includes* followed by "#define _ASMLANGUAGE" and one or more C preprocessor **#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.
- **Globals:** The globals block consists of a one-line C comment containing the word *globals* followed by one or more assembler declarations, one per line. This block groups together all declarations in the module that are intended to be visible outside the module. Labels that are to be exported as functions entry points should use the FUNC_EXPORT macro e.g. FUNC_EXPORT (fastSyscallHandler). Labels that are to be exported as data should use the DATA_EXPORT macro e.g. DATA_EXPORT (privGuestIrq)
- **Externs:** The externs block consists of a one-line C comment containing the word *externs* followed by one or more assembler declarations, one per line. This block groups together all declarations in the module that are imported from other files. Labels that are to be imported as function entry points should use the FUNC_IMPORT macro e.g. FUNC_IMPORT (archUpdateVmmu). Labels that are to be imported as data should use the DATA_IMPORT macro e.g. DATA_IMPORT(ctx)

5.2 Assembler Function Layout

The assembler function layout should be identical to the C function layout (See section 4.2). In addition the description should declare the register usage, and the C prototype for the function.

Each function label should be declared using the FUNC_LABEL macro:

```
FUNC_LABEL(fastIntArchSet)
```

The end of the function should be declared using the FUNC_END macro:

```
FUNC_END(fastIntArchSet)
```

5.3 Assembler Comment Style

Since the C preprocessor is being used, the same comment style will be used for assembler as was used for C (see section [4.3.4]).

5.4 Assembler Formatting and Style

Assembler code should be indented by 8 space characters, except for labels which should begin in the first column. Otherwise follow the same coding conventions that apply to the C code (as applicable).

6 Tcl Coding Style

This chapter defines the layout, style, comment, and naming conventions for Tcl code.

6.1 Tcl Module Layout

A module is any unit of code that resides in a single Tcl file. The conventions in this section define the standard module heading that must come at the beginning of every Tcl module following the standard file heading. The module heading consists of the blocks described below; the blocks are 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.

6.2 General Module Documentation

The module documentation is a block of single-line Tcl comments beginning with the keyword DESCRIPTION and consisting of a complete description of the overall module purpose and function, especially the external interface. The description includes the heading RESOURCE FILES followed by a list of relevant Tcl files sourced inside the file.

6.2.1 Globals

The globals block consists of a one-line Tcl comment containing the word globals followed by one or more Tcl declarations, one per line. This block groups together all declarations in the module that are intended to be visible outside the module.

6.2.2 Tcl File and Module Headings

This is an example of a complete standard file heading, in Tcl.

```
# Browser.tcl - Browser Tcl implementation file
#
# Copyright 1994, 1995 Wind River Systems, Inc.
#
# The right to copy, distribute, modify, or otherwise make use
# of this software may be licensed only pursuant to the terms
# of an applicable Wind River license agreement.
#
# modification history
# --------------------
# 30oct95,jco  added About menu and source browser.tcl in .wind.
# 02sep95,pad  fixed communications loss with license daemon (SPR #1234).
# 05mar95,jcf  upgraded spy dialog
# 08feb95,p_m  take care of loadFlags in wtxObjModuleInfoGet.
# 06dec94,c_s  written.
#
# DESCRIPTION
# This module is the Tcl code for the browser. It creates the main window
# and initializes the objects in it, such as the task list and memory
# charts.
#
# RESOURCE FILES
# wpwr/host/resource/tcl/shelbrws.tcl
# wpwr/host/resource/tcl/app-config/Browser/*.tcl
# ...
#

# globals

set browserUpdate  0     ;# no auto update by default
```

6.2.3 Tcl Procedure Layout

The following conventions define the standard layout for every procedure in a module. Each procedure is preceded by the procedure documentation, a series of Tcl comments that includes the following blocks. The documentation contains no blank lines, but each block is delimited with a line containing a single pound symbol (#) in the first column.

6.2.3.1 Tcl Procedure Banner

A Tcl comment that consists of 80 pound symbols (#) across the page.

6.2.3.2 Tcl Procedure Title

One line containing the function name followed by a short, one-line description. The function name in the title must match the declared function name. This line becomes the title of automatically generated reference entries and indexes.

6.2.3.3 Tcl Procedure Description

A full description of what the function does and how to use it. This should follow the markup line

```
# DESCRIPTION
```

6.2.3.4 Tcl Procedure Synopsis

The markup line

```
# SYNOPSIS
```

followed by a the synopsis of the procedure

6.2.3.5 Tcl Procedure Parameters

TCL parameters start with the markup line

```
# PARAMETERS
```

For each parameter, the markup "`# \i` " followed by the parameter name on one line, followed by its complete description on the next line. The next line should start with `[in], [out]` or `[in/out]` for the usage of the parameter. Include the default value and domain of definition in each parameter description. For an example, see this example [6.2.3.8]. For details on item lists, see Parameter Lists [10.6.1].

6.2.3.6 Tcl Procedure Returns

The word markup line

```
# RETURNS:
```

followed by a description of the possible explicit result values of the function (that is, values returned with the Tcl return command).

```
# RETURNS:
A list of 11 items: vxTicks taskId status priority pc sp errno
timeout entry priNormal name
```

If the return value is meaningless, enter N/A:

```
# RETURNS: N/A
```

6.2.3.7 Tcl Procedure Errors

The markup word "`# ERRORS:` " followed by all the error messages or error code (or both, if necessary) raised in the procedure by the Tcl error command.

```
# ERRORS:
# \is
# \i <prjFile>: no such project (<status>)
# The project file could not be opened.  The <status> should tell why
# \ie
```

If no error statement is invoked in the procedure, enter N/A.

```
# ERRORS: N/A
```

The procedure documentation ends with an empty Tcl comment starting in column one. The procedure declaration follows the procedure heading and is separated from the documentation block by a single blank line. The format of the procedure and parameter declarations is shown in C Declaration Formats.

6.2.3.8 Standard Tcl Procedure Layout

```
#####################################################################
#
# browse - browse an object, given its ID
#
# DESCRIPTION
# This function is bound to the "Show" button, and is invoked when
# that button is clicked. If the argument (the contents of...
#
# SYNOPSIS
# browse [objAddr | symbol | &symbol]
#
# PARAMETERS
# \is
# \i <objAddr>
# [in] The address of an object to browse.
#
# \i <symbol>
# [in] A symbolic address whose contents is the address of
# an object to browse.
#
# \i <&symbol>
# [in] A symbolic address that is the address of an object to browse.
# \ie
#
# RETURNS: N/A
#
# ERRORS: N/A
#

proc browse {args} {
    ...
}
```

6.2.4 Tcl Code Outside Procedures

Tcl allows code that is not in a procedure. This code is interpreted immediately when the file is read by the Tcl interpreter. Aside from the global-variable initialization done in the globals block near the top of the file, collect all such material at the bottom of the file.

However, it improves clarity—when possible—to collect any initialization code in an initialization procedure, leaving only a single call to that procedure at the bottom of the file. This is especially true for dialog creation and initialization, and more generally for all commands related to graphic objects.

Tcl code outside procedures must also have a documentation heading, including the following blocks.

6.2.4.1 Banner

A Tcl comment that consists of 78 pound symbols (#) across the page.

6.2.4.2 Title

One line containing the file name followed by a short, one-line description. The file name in the title must match the file name in the file heading.

6.2.4.3 Description

A description of the out-of-procedure code.

6.2.4.4 Heading for Out-Of-Procedure Tcl Code

This is an example of a heading for out-of-procedure Tcl code:

```
######################################################################
#
# 01Spy.tcl - Initialization code
#
# This code is executed when the file is sourced. It executes the module
# entry function which does all the necessary initialization to get a
# runnable spy utility.
#

# Call the entry point for the module

spyInit
```

6.2.5 Tcl Declaration Formats

Include only one declaration per line. Declarations are indented in accordance with Indentation and begin at the current indentation level. The remainder of this section describes the declaration formats for variables and procedures.

6.2.5.1 Variables

For global variables, the Tcl set command appears first on the line, separated from the identifier by a space character. Complete the declaration with a meaningful comment at the end of the same line. Variables, values, and comments should be aligned, as in the following example:

```
set rootMemNBytes  0 ;# memory for TCB and root stack
set rootTaskId          0 ;# root task ID
set symSortByName  1 ;# boolean for alphabetical sort
```

6.2.5.2 Procedures

The procedure name and list of parameters appear on the first line, followed by the opening curly brace. The declarations of global variables used inside the procedure begin on the next line, one on each separate line. The rest of the procedure code begins after a blank line.

For example:

```
proc lstFind {list node} {
    global firstNode
    global lastNode
    ...
}
```

6.2.6 Code Layout

The maximum length for any line of code is 80 characters. If more than 80 characters are required, use the backslash character to continue on the next line.

The rest of this section describes conventions for the graphic layout of Tcl code.

6.2.6.1 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 put more than one statement on a line. The only exceptions are:

- A for statement where the initial, conditional, and loop statements can be written on a single line:

```
      for {set i 0} {$i < 10} {incr i 3} {
```

- A switch statement whose actions are short and nearly identical (see the switch statement format in Indentation [6.2.6.3]).

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

```
if {$i > $count} {
    set i $count
}
```

• Opening braces "{", defining a command body, are always on the same line as the command itself.

• Closing braces "}" and switch patterns always have their own line.

6.2.6.2 Horizontal Spacing

Put spaces around binary operators. Put spaces before an open parenthesis, open brace and open square bracket if it follows a command or assignment statement. For example:

```
set status [fooGet $foo [expr $i + 3] $value]
if {&value & &mask} {
```

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

```
set a [expr ($b + $c) * \
            ($d + $e)]
set status [fooList $foo $a $b $c \
                    $d $e]
if {($a == $b) && \
    ($c == $d)} {
    ...
}
```

6.2.6.3 Indentation

• Indentation levels are every four characters (columns 5, 9, 13, …) Indentation should be performed with spaces only (no tabs).

• The module and procedure headings and the procedure declarations start in column one.

• The closing brace of a command body is always aligned on the same column as the command it is related to:

```
while { condition }{
    statements
}


foreach i $elem {
    statements
}
```

- Add one more indentation level after any of the following:

  - procedure
  - declarations
  - conditionals (see below)
  - looping constructs
  - switch statements
  - switch patterns

- The else of a conditional is on the same line as the closing brace of the first command body.

It is followed by the opening brace of the second command body. Thus the form of the conditional is:

```
if { condition }{
    statements
} else {
    statements
}
```

The form of the conditional statement with an elseif is:

```
if { condition } {
    statements
} elseif { condition } {
    statements
} else {
    statements
}
```

- The general form of the switch statement is:

```
switch [flags] value {
    a    {
         statements
         }
    b    {
         statements
         }
    default {
        statements
        }
}
```

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

```
switch [flags] value {
    a    {set x $aVar}
    b    {set x $bVar}
    c    {set x $cVar}
}
```

• Comments have the same indentation level as the section of code to which they refer.

• Opening body braces "{" have no specific indentation; they follow the command on the same line.

6.2.6.4 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 pound symbol as in the following:

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

set foo 0
```

- Multi-line comments have each line beginning with the pound symbol as in the example below. Do not use a backslash to continue a comment across lines.

```
# This is the CORRECT format for a multi-line comment
# in a section of code.

set foo 0

# This is the INCORRECT format for a multi-line comment \
in a section of code.

set foo 0
```

• Comments on global variables appear on the same line as the variable declaration, using the semicolon (;) character:

```
set day     night ;# This is a global variable
```

6.2.6.5 Naming Conventions

The following conventions define the standards for naming modules, functions and variables. The purpose of these conventions is uniformity and readability of code.

When creating names, remember that code is written once but read many times. Make names meaningful and readable. Avoid obscure abbreviations.

Names of functions and variables are composed of upper- and lowercase characters and no underscores. Capitalize each “word” except the first:

```
aVariableName
```

Every module has a short prefix (two to five characters). The prefix is attached to the module name and to all externally available procedures and variables. (Names that are not available externally need not follow this convention.)

```
fooLib.tcl         module name
fooObjFind         procedure name
fooCount           variable name
```

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

```
fooObjFind         foo-object-find
sysNvRamGet        system-nonvolatile RAM-get
taskInfoGet        task-info-get
```

6.2.7 Tcl Style

The following conventions define additional standards of programming style:

- Comments: Insufficiently commented code is unacceptable.
- Procedure Length: Procedures should have a reasonably small number of lines, less than 50 if possible.
- Case Statement: Do not use the case keyword. Use switch instead.
- expr and Control Flow Commands: Do not use expr in commands such as if, for or while

except to convert a variable from one format to another:

CORRECT:
```
if {$id != 0} {


if {[expr $id] != 0} {
```

INCORRECT:
```
if {[expr $id != 0]} {
```

• expr and incr: Do not use expr to increment or decrement the value of a variable. Use incr instead.

CORRECT:
```
incr index

incr index -4
```

INCORRECT:
```
set index [expr $index + 1]
```

• catch Command: The catch command is very useful to intercept errors raised by underlying procedures so that a script does not abort prematurely. However, use the catch command with caution. It can obscure the real source of a problem, thus causing errors that are particularly hard to diagnose. In particular, do not use catch to capture the return value of a command without testing it. Note also that if the intercepted error cannot be handled, the error must be resubmitted exactly as it was received (or translated to one of the defined errors in the current procedure):

CORRECT:

```
if [catch "dataFetch $list" result] {
    if {$result == "known problem"} {
        specialCaseHandle
    } else {
        error $result
    }
```

INCORRECT:

```
catch "dataFetch $list" result
```

• if then else Statement: In an if command, you may omit the keyword then before the first command body; but do not omit else if there is a second command body.

CORRECT:

```
if {$id != 0} {
    ...
} else {
    ...
}
```

INCORRECT:

```
if {$id !=0} then {
    ...

} {
    ...
}
```

• Return Values: Tcl procedures only return strings; whatever meaning the string has (a list for instance) is up to the application. Therefore each constant value that a procedure can return must be described in the procedure documentation, in the RETURNS: block. If a complex element is returned, provide a complete description of the element layout. Do not use the return statement to indicate that an abnormal situation has occurred; use the error statement in that situation.

The following illustrates a complex return value consisting of a description:

```
# Return a list of 11 items: vxTicks taskId status priority pc
# sp errno timeout entry priNormal name

return [concat [lrange $tiList 0 1] [lrange $tiList 3 end]]
```

The following illustrates and simple return value:

```
# This code checks whether the VxMP component is installed:
if [catch "wtxSymFind -name smObjPoolMinusOne" result] {
    if {[wtxErrorName $result] == "SYMTBL_SYMBOL_NOT_FOUND"} {
        return -1  # VxMP is not installed
    } else {
        error $result
    }
} else {
    return 0# VxMP is installed
}
```

• Error Conditions: The Tcl error command raises an error condition that can be trapped by the catch command. If not caught, an error condition terminates script execution. For example:

```
if {$defaultTaskId == 0} {
    error "No default task has been established."
}
```

Because every error message and error code must be described in the procedure header in the ERRORS: block, it is sometimes useful to call error in order to replace an underlying

error message with an error expressed in terms directly relevant to the current procedure. For example:

```
if [catch "wtxObjModuleLoad $periodModule" status] {
    error "Cannot add period support module to Target ($status)"
}
```

7 C++ Coding Style

The formating of C++ code is to follow that of C code.

In order to facilitate the correct formatting of the code, the clang-format program is to be used on new or modified code to correctly format the code.

```
clang-format -style=file <source file>
```

Unfortunately clang-format does not currently support Whitesmiths indentation and bracing. In the interim the astyle program (http://astyle.sourceforge.net/) can be used.

```
astyle –style=whitesmiths –s4
```

The rules for clang-format should be placed into a .clang-format file as follows:

```
---
Language:        Cpp
# BasedOnStyle:  WebKit
AccessModifierOffset: -4
AlignAfterOpenBracket: true
AlignConsecutiveAssignments: false
AlignConsecutiveDeclarations: true
AlignEscapedNewlinesLeft: false
AlignOperands:   true
AlignTrailingComments: true
AllowAllParametersOfDeclarationOnNextLine: true
AllowShortBlocksOnASingleLine: false
AllowShortCaseLabelsOnASingleLine: false
AllowShortFunctionsOnASingleLine: None
AllowShortIfStatementsOnASingleLine: false
AllowShortLoopsOnASingleLine: false
AlwaysBreakAfterDefinitionReturnType: None
AlwaysBreakAfterReturnType: None
AlwaysBreakBeforeMultilineStrings: true
AlwaysBreakTemplateDeclarations: true
BinPackArguments: false
BinPackParameters: false
BraceWrapping:
  AfterClass:      false
  AfterControlStatement: false
  AfterEnum:       false
  AfterFunction:   true
  AfterNamespace:  false
  AfterObjCDeclaration: false
  AfterStruct:     false
  AfterUnion:      false
  BeforeCatch:     false
  BeforeElse:      false
  IndentBraces:    false
BreakBeforeBinaryOperators: All
```

```
BreakBeforeBraces: Stroustrup
BreakBeforeTernaryOperators: true
BreakConstructorInitializersBeforeComma: true
ColumnLimit:     80
CommentPragmas:  '^ IWYU pragma:'
ConstructorInitializerAllOnOneLineOrOnePerLine: false
ConstructorInitializerIndentWidth: 4
ContinuationIndentWidth: 4
Cpp11BracedListStyle: false
DerivePointerAlignment: false
DisableFormat:   false
ExperimentalAutoDetectBinPacking: false
ForEachMacros:   [ foreach, Q_FOREACH, BOOST_FOREACH ]
IncludeCategories:
  - Regex:           '^"(llvm|llvm-c|clang|clang-c)/'
    Priority:        2
  - Regex:           '^(<|"(gtest|isl|json)/)'
    Priority:        3
  - Regex:           '.*'
    Priority:        1
IndentCaseLabels: false
IndentWidth:     4
IndentWrappedFunctionNames: false
KeepEmptyLinesAtTheStartOfBlocks: true
MacroBlockBegin: ''
MacroBlockEnd:   ''
MaxEmptyLinesToKeep: 1
NamespaceIndentation: Inner
ObjCBlockIndentWidth: 4
ObjCSpaceAfterProperty: true
ObjCSpaceBeforeProtocolList: true
PenaltyBreakBeforeFirstCallParameter: 19
PenaltyBreakComment: 300
PenaltyBreakFirstLessLess: 120
PenaltyBreakString: 1000
PenaltyExcessCharacter: 1000000
PenaltyReturnTypeOnItsOwnLine: 60
PointerAlignment: Middle
ReflowComments:  true
SortIncludes:    true
SpaceAfterCStyleCast: false
SpaceBeforeAssignmentOperators: true
SpaceBeforeParens: Always
SpaceInEmptyParentheses: false
SpacesBeforeTrailingComments: 1
SpacesInAngles:  false
SpacesInContainerLiterals: true
SpacesInCStyleCastParentheses: false
SpacesInParentheses: false
SpacesInSquareBrackets: false
Standard:        Cpp11
TabWidth:        8
UseTab:          Never
```

8 Python Coding Style

This chapter defines the layout, style, comment, and naming conventions for Python code. It is mostly based on the Python Enhancement Proposal 8 (PEP-8) located at https://www.python.org/dev/peps/pep-0008/.

8.1 Standard File Encoding

A text file must use UTF-8 encoding for Python 3 and ASCII for Python 2 however all identifiers must only use ASCII.

8.2 Python Module Layout

A module is any unit of code that resides in a single Python file. The conventions in this section define the standard module heading that must come at the beginning of every Python module following the standard file heading. The module heading consists of the blocks described below; the blocks are 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.

8.3 General Module Documentation

The file heading consists of up to three blocks:

- An optional "shebang" line (eg: `#!/usr/bin/env python`) when the file contains main line code such as a sections starting with `if __name__ == "__main__"`.
- A main documentation string (docstring) containing a single line module description, the copyright notice and the general module documentation
- An optional modification history docstring.

The comment blocks are further described below. The blocks are separated by one empty line

8.3.1 Environment

When the Python file contains main line code, such as code inside an `if __name__ == "__main__"` statement, it is necessary to include a line in the form `#!/usr/bin/env python` so it can be executed from the shell. This line is necessary for Posix compatibility and is not needed for files only running on Windows however, only supporting Windows is strongly discouraged.

8.3.2 Main docstring block

The main doc string consists of three sections; a file title, a copyright notice and a general module description, all described below.

8.3.2.1 File title

The file title consists of a one-line comment containing the tool, library, or application name followed by a short description. The name must be the same as the filename. This line will become the title in generated reference entries and indexes. For example

```
      fabulator.py - Fabulator class and utility functions
```

8.3.2.2 Copyright for Wind River Proprietary Code

The copyright notice shall follow that used for C code see section [3.2.2].

8.3.2.3 Copyright for Open Source Code

The copyright nofice shall follow that used for C code see section [3.2.3].

8.3.2.4 General module documentation

This section contains the general module documentation. It should contain enough information to describe module but must avoid describing implementation details as they may change over time.

8.3.3 Modification History block

The modification history documentation shall follow that used for C code see section [3.2.4].

8.3.4 File Heading Example for Python Files

This is an example of a complete standard file heading, in Python.

```
#!/usr/env/bin python
"""
fabulator.py - Fabulator class and utility functions
"""

"""
Copyright (c) 2012, 2014-2015, 2018 Wind River Systems, Inc.

The right to copy, distribute, modify, or otherwise make use
of this software may be licensed only pursuant to the terms
of an applicable Wind River license agreement.
"""

"""
modification history
--------------------
```

```
01oct18,rdl update coding standard for test file format (VXW7-1299)
15sep15,nfs added defines MAX_FOOS and MIN_FATS.
15feb14,dnw added functions fooGet() and fooPut();
            added check for invalid index in fooFind().
10feb12,dnw written.

"""
```

8.4 Code Layout

The following sections describe how to layout code so it is visually consistent.

8.4.1 Indentation

Use 4 spaces for every indentation level. Do not use tab characters. Do not mix tab and spaces for indentation.

Align multi-line statements so they align with the opening delimiter. It is possible the alignment does not fall on a multiple of 4 spaces, this is OK.

CORRECT:

```
def myFunc(arg_1, arg_2,
           arg_3, arg_4):
```

CORRECT:

```
def myFunc(arg_1,
           arg_2,
           arg_3,
           arg_4):
```

INCORRECT:

```
def myFunc(arg_1, arg_2,
    arg_3, arg_4):
```

8.4.2 Maximum line length

Lines must be limited to 79 characters maximum. Should a line be longer than 79 characters, implicit line continuation should be used to restrict lines to 79 characters.

When splitting strings into multiple lines and there are spaces present, prefer splitting on a non-space character boundary and include the space in the continued line.

INCORRECT:

```
errorMsg = "Your operation resulted in an error. Please consult the
manual and retry."
```

CORRECT:

```
errorMsg = ("Your operation resulted in an error."
            " Please consult the manual and retry.")
```

Do not use the backslash ('\') character to split lines unless absolutely necessary. The backslash character must not be used to split strings. Should a string need to be split, prefer using parentheses and use Python's built-in automatic string concatenation.

INCORRECT:

```
errorMsg = "Your operation resulted in an error. Please "\
           " consult the manual and retry."
```

CORRECT:

```
errorMsg = ("Your operation resulted in an error. Please"
            " consult the manual and retry.")
```

It is acceptable to use the backslash character for long multiple statements but should be avoid if possible

ACCEPTABLE:

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
     open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

When splitting lines containing a binary operator, it is preferable to include the binary operator as the first character of the continuation line. For new code, only use the preferred method.

ACCEPTABLE:

```
total = (numberPasses +
         numberFails +
         numberSkipped +
         numberNotRun +
         numberExceptions)
```

PREFFERED:

```
total = (numberPasses
         + numberFails
         + numberSkipped
         + numberNotRun
         + numberExceptions)
```

8.4.3 Blank lines

Top level functions and class definitions must be preceded by two blank lines.

Method definitions inside a class are preceded by a single blank line

Use blank lines to separate logical blocks inside a function or a method to improve readability when necessary

8.5 Imports

Import a single module per line.

Import lines must be in alphabetical order.

INCORRECT:

```
import os, sys, json
```

CORRECT:

```
import json
import os
import sys
```

Imports are grouped as follows, each group is separated by a blank line

- standard library imports
- related third party imports
- local application / library specific imports

INCORRECT:

```
import os
import sys
import json
import argparser
from datetime import time
import thridpartylibrary
```

CORRECT:

```
import argparser
import json
import os
import sys

import thridpartylibrary
```

```
      from datetime import time
```

8.6 Single and double quotes

Do not use the backslash character to escape a quote or double quote character inside a string. Instead, use the appropriate quotes, single or double for quoting.

INCORRECT:

```
      msg = "The user name is \"%s\"" % userName
```

CORRECT:

```
      msg = 'The user name is "%s"' % userName
```

CORRECT:

```
     msg = "The user's current status is invalid for this operation."
```

8.7 Comments

When commenting code, focus on describing the high level operation and omit implementation details as implementation will change over time. Be as terse as possible and keep comments to a few lines. Be as verbose as needed when describing data structure of complex code blocks.

Avoid making specific references to the implementation to avoid having to change comments during rework when functionality remains the same.

INCORRECT:

```
      # set the counter to 0 to reinitialize it
      counter = 0
```

CORRECT:

```
      # reinitialize counter
      counter = 0
```

Avoid using inline comments unless necessary. Inline comments start with a hash sign ('#') followed by a single space

AVOID:

```
      counter = counter + 1    # increment counter
```

Each line of a comment block start with a hash sign and a single space. More spaces can be present to indent text sections within the comment.

INCORRECT:

```
    #The following code validates command line parameters and ensure all
    #specified files can be found
```

INCORRECT:

```
    # The following code validates command line parameters and ensure all
    # specified files can be found
```

8.8 Naming

8.8.1 Naming styles

Only the following naming styles should be used. When modifying existing files, continue using the current naming styles so to keep a consistent look

- b (single lowercase letter)
- B (single uppercase letter)
- lowercase
- lower_case_with_underscores
- UPPERCASE
- UPPER_CASE_WITH_UNDERSCORES
- CapitalizedWords (aka CapWords, CamelCase or StudlyCaps)
    - Note: When using acronyms in CapWords, capitalize all the letters of the acronym. Thus HTTPServerError is better than HttpServerError.
- mixedCase (differs from CapitalizedWords by initial lowercase character!)

When a name conflicts with a python keyword, a trailing underscore should be added to the new identifier. For example, use `type_` or `class_` to differentiate the identifiers from the Python keywords `type` and `class` respectively.

Use one leading underscore for local identifiers. For example: `_localVariable`

Use two leading underscores for class attributes for invoking name mangling within a class. For example inside class FooBar, `__boo` becomes `_FooBar__boo`;

8.8.2 Single letter variable names

Do not use l (lower case 'el'), o (lower case 'oh'), O (uppercase 'oh') and I (uppercase 'eye') for single letter variable names as they can be mistaken for number 1 (one) and 0 (zero) when some fonts are used.

8.8.3 ASCII Identifier names

Use only ASCII characters in identifier names.

8.8.4 "Dunder" Names

"Dunder" names start and end with two underscores (e.g.: __name__) must be placed after the module docstring but before any code in a module.

8.8.5 Documentation String (docstring)

A documentation string, or docstring is a string literal defined immediately after the definition of a class, method, function or module. The string literal must be the first statement for it to be properly interpreted and thus be made available to documentation tools. A docstring begins and ends with triple double quotes (`"""`) which contains documentation about the class, method, function or module it refers to. Text begins one space after the initial triple double quote.

Single line doc strings end with a triple double quote on the same line

```
def kos_root():
    """ Return the pathname of the KOS root directory. """

    global _kos_root
    if _kos_root:
        return _kos_root
    ...
```

Multi-line doc string continues as follows. Subsequent lines are left justified an aligned with the initial quote. The final triple double quote ends the doc string on its own line.

```
def complex(real=0.0, imag=0.0):
    """ Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """

    if imag == 0.0 and real == 0.0:
        return complex_zero
    ...
```

A blank line follows the docstring.

A docstring should not repeat the class, method or function signature so to reduce maintenance when parameters change.

It is acceptable to use raw triple-quoted strings when docstrings contain the backslash ('\') character. For example:

```
    r""" Print a row of backslash ('\') characters """"
```

8.8.6 Class Layout

The following conventions define the standard layout for every class in a module. Class names start with an upper case letter and should use the *CapWords* convention. In Python 3, class names need only to include parentheses if they extend and existing class, otherwise, they may be omitted. In Python 2, parentheses must always be used.

Python3 examples

```
class BaseClass:
    ...

class MyClass(BaseClass):
    ...
```

Python2 examples

```
class MyClass(object):
    ...
```

The class definition immediately follows the class' docstring.

8.8.6.1 Class Doc String Format

A Python class docstring should only describe what the class does and not how it is implemented as implementation will change over time. Public attributes, if any, should be listed and described. If a class subclasses another class, the docstring should indicate the changes this class provides over the parent class.

8.8.6.2 Methods

A single blank line must be present before a method declaration. Should the method be decorated, the blank line must be above the top-most decorator and there shall be no blank line between the last decorator and the method definition.

Method should always use `self` as the first argument for an instance method and `cls` for a class method.

Method docstring should describe the method`s behavior, input arguments and returned data. If the method raises exceptions, they should be listed and conditions raising them must be described. There is no need to list the `self` or `cls` argument in the docstring. Below is a sample method docstring:

```
class MyClass:
    """ This is a sample class """

    def isValid(self, attrName, validValues):
        """ Check if attribute value is valid by looking in provided list
of valid values.

        attrName - The name of attribute to check
        validValues - list of valid values for the attribute

        Returns True if the attribute value is present in validValues.
                False otherwise
        """

        try
            return getattr(self, attrName) in validValues
        except AttributeError:
            # handle non-existent attribute
            return False
```

8.8.7 Function Layout

A function definition is preceded by two blank lines unless decorated by one or many decorators. In that case, the two blank lines precede the function's top-most decorator.

When a function definition line is greater than 79 characters, it must be split into multiple lines each having 79 characters or less.

A docstring immediately follows the function definition. There shall be no blank line between the definition and the docstring. Follow the docstring style information for formatting:

Unless a function is extremely simple and can be described in a single line, the docstring is likely to span many lines. It format is as follows:

- One to many lines describing the function
- one blank line
- list of parameters and their descriptions
- one blank line
- the capitalized word `Returns` on a line by itself (or followed by the word `None` when there are no explicit returned values)
- An indented list of returned values

If the function raises exceptions, the doc string must continue with

- one blank line
- The capitalized word `Raises` on a line by itself
- an indented list of raised exceptions

8.8.7.1 Simple function declaration

```
def wasteTime(timeSec):
    """ Waste a certain amount of time spinning

    timeSec - The amount of time to waste in seconds

    Returns None

    Raises
       ValueError if timeSec cannot be converted to a float
    """

    ...
```

8.8.7.2 Complex function declaration

```
def storeAttachments(recordId, attachments, db=DATABASE, user=USER,
                     pass=PASSWORD):
    """ Store attachements to the specified record. Alternate database
    can be specified.

    recordId - record identifier
    attachments - list of fully qualified file names of attachments
    db - alternate database
    user - alternate database user name
    pass - alternate database password


puteDeceleration(initialVelocity, frictionCoefficientTimeSec,
brakingForce):
    """ Waste a certain amount of time spinning

    timeSec - The amount of time to waste in seconds

    Returns None

    Raises
        ValueError if timeSec cannot be converted to a float
    """

    ...
```

8.8.8 Miscellaneous

8.8.8.1 White spaces

Keep white spaces to a minimum while following these rules

DO put a space before and after an equal sign '=' except when used inside parentheses to indicate a keyword argument or a default parameter value.

YES:

```
        x = x + offset
        foo(bar='junk')
```

NO:

```
        x=x+1
        foo(bar = 'junk')
```

DO put a space after a comma

YES:

```
        print(x, y)
```

NO:

```
        print(x,y)
```

DO put a space after a colon except when slicing

YES:

```
        if name=='foo': found = True
        part = table[0:4]
```

NO:

```
        if name == 'foo' : found = True
        part = table[0 : 4]
```

DO NOT put a space before opening a parenthesis or a bracket

YES:

```
        print('Name')
        value = data['age']
```

NO:

```
        print ('Name')
        value = data ['age']
```

DO NOT use spaces to align operators

YES:

```
        longitude = 45.0
        latitude = 75.0
        speed = 3
```

NO:

```
        longitude = 45.0
        latitude  = 75.0
        speed     = 3
```

DO use spaces to logically group. Always have the same number of spaces on either side of an operator

YES:

```
x = x + 1
c = (a*a + b*b) ** 0.5
```

NO:

```
x = x+1
c = (a * a + b * b)**0.5
```

8.8.8.2 Semi-colons and compound statements

DO NOT put semi-colons at the end of a line

YES:

```
print(a)
```

NO:

```
print(a);
```

AVOID putting multiple statements on a single line

PREFERRED:

```
step(1)
step(2)
step(3)
if age < 3:
      print('Too young')
```

DISCOURAGED:

```
step(1); step(2); step(3)
if age < 3: print('Too young')
```

NO:

```
if isValid: doValidThing()
else: printErrorMessage()

try: x/y
except DivisionByZeroError: printDivByZeroMsg()
```

8.8.9 Comments

8.8.9.1 General comments

Comments must be used to clarify area of code that are complex or are not easily deducted from the source code. Write comments in general terms and avoid describing the code implementation as it is likely to change over time.

NO:

```
# go through list and remove objects with status equal to False
```

YES:

```
# remove objects with invalid status from list
```

8.8.9.2 Inline comments

Generally speaking, inline comments are not necessary and are distracting when stating the obvious.

NO:

```
x = x + 1                           # increment x
```

ACCEPTABLE:

```
leftMargin = leftMargin + 1         # indent from the margin
```

8.8.9.3 Block comments

Block comments consist of multiple lines starting with a '#' followed by one space. Paragraphs are separated by a line starting with a '#' without a space following it. A single blank line must precede a block comment.

```
...

# check for debug mode being active:
#   User may set environment variable tracked by FOO_BAR_DEBUG_MODE
#   but they can turn off all debugging with environment variable
#   FOO_BAR_DEBUG_OVERRIDE
#
# Note that other variables may turn on DEBUG mode ...
...
```

8.9 Exceptions

8.9.1 Catch specific exceptions

Not catching specific exceptions can hide problems as unwanted exceptions.

KeyboardInterrupt for example can be inadvertently caught .

NO

```
try:
        myOperation()
    except:
        logger.critical('My operation failed')
```

YES

```
    try:
        myOperation()
    except ValueError:
        logger.critical('Invalid value provided to my operation')
```

8.9.2 Process exceptions explicitly

Avoid the use of the `pass` keyword when processing exception unless it is intended as such.

NO

```
    try:
        myOperation()
    except:
        pass
```

YES

```
    try:
        myOperation()
    except ValueError as exc:
        logging.critical('Invalid value provided to my operation (%s)',
                         exc)
    except IndexError:
        # harmless exception, ignore it
        pass
```

8.9.3 Derive exceptions from Exception, not BaseException

Unless required, all new exceptions should be derived from the Exception class rather than the BaseException class.

9 Component Description Language

This section provides coding standards for VxWorks' component description language (CDL). CDL is used to create Component Descriptor (CDF) Files, which are used in many places of the system to describe VxWorks components and their interdependencies.

9.1 SYNOPSIS Section

Each VxWorks CDL object (Api, Bsp, Bundle, Component, Folder, Parameter, Profile, or Selection) should contain both a NAME property and a SYNOPSIS property.

The NAME field is used by Workbench to display brief information about the element (such as in a pull-down menu, or in a hierarchical display of components). Because of the way in which Workbench displays the NAME field, the NAME field is required to be relatively short, and should always be contained within a single line.

The SYNOPSIS field is used by Workbench to display extra information about the element. SYNOPSIS entries should be descriptive, useful and not just a repetition of the NAME field. The SYNOPSIS field can be multiple lines in length.

When composing a long SYNOPSIS field use the backslash notation to limit the source file line length to 80 characters, and use indentation to align subsequent lines of text.

For example:

```
Component INCLUDE_PROTECT_TEXT {
    NAME        Write-protect program text


    SYNOPSIS    Write protect the text segment using the MMU.             \
                Note that this will cause padding be inserted             \
                between the end of the text segment and the beginning     \
                of the data segment so that the data segment starts on a  \
                page boundary.
   ...
}
```

At present, there is no way to insert special characters such as linefeeds into the SYNOPSIS string, so the SYNOPSIS is displayed by Workbench as a paragraph of text, with Workbench performing its own line wrapping of the contents of the SYNOPSIS string. Workbench collapses adjacent white space characters into a single white space, that is, the indentation used in the above example will be displayed as a single space.

9.2 Naming

In order to obtain naming consistency of Components that appear in the kernel configurator within WorkBench (and also from the command line interface), the following namespace rules should be followed.

Note that CDL defines the following keywords: Api, Bsp, Bundle, Component, Folder, Parameter, Profile, and Selection. All of these constructs have a NAME and a SYNOPSIS .

The following rules should generally apply to all of these CDL types:

1. The word “component(s)” shall not be used since it's clearly apparent that the user is dealing with components (it's the kernel configurator) .

2. Only the first word of a NAME or SYNOPSIS shall be capitalized (except for acronyms and brand names, e.g. ED&R).

3. The SYNOPSIS shall not be a cut-and-paste of the NAME.

4. The SYNPOSIS shall be a complete sentence, or several sentences, not just a phrase or a sentence fragment.

5. Be consistent in the usage of terms and/or acronyms throughout a folder hierarchy for the NAME and SYNOPSIS field. If ED&R is used in the NAME of the FOLDER then use ED&R (not Error Detection and Reporting) throughout the hierarchy.

6. A Component NAME need not repeat keywords used in the FOLDER NAME. In the FOLDER named “POSIX”, don’t apply a name of “POSIX" to a component within the folder; instead using just “Semaphores” will suffice. This is not a hard requirement. It's up to the judgment of the approver of the new CDL constructs whether it's appropriate to repeat the keyword.

7. Use the plural form of objects in the NAME and SYNOPSIS, e.g. “Semaphores” instead of “Semaphore”

10APIGEN Reference Standards

This chapter defines standards for the creation and documentation of published VxWorks APIs using a markup language that is processed by the program “apigen”.

10.1 Introduction

Wind River uses the utility “apigen” to convert source code files into API reference documentation, known as reference entries. Most of that reference information is taken from comments, and some is taken from source code statements. In order for “apigen” to recognize and process this information, files must follow Wind River’s layout conventions, and the comments must follow Wind River’s standard format and markup. All modules must be able to generate valid reference entries. This section covers the following topics:

- Content and Organization – required and optional sections, and where they appear.
- Format and Style – text formatting markup and usage standards.
- Directives – controls for special actions, such as including information from other files, hiding internal information, or overriding defaults.

For more detail on formatting, markup, and directives, see the reference entry for apigen. For details on documentation style standards (spelling, punctuation, grammar, and word usage), see the Technical Publications Style Guide. Both are available on the Tech Pubs web site.

10.2 Content and Organization

The documentation segments for the module and the routines give rise to both internal and external reference documentation. As such, they should avoid providing background information that is not specific to Wind River products, but should give sufficient detail to enable anyone with the appropriate technical background to use the module fully and correctly. Usage examples are essential.

Reference entries contain a number of required sections, which can vary depending on the type of API, whether a C library, a C routine, a Tcl procedure, a shellscript, or other type. Required sections use standard headings and appear in a standard order, shown in this table. Special considerations for these sections are discussed below.

For details on the markup described in this section, see Format and Style in Section 10.16. This table lists standard sections for reference entries, in required order.

Section Name	C Library	C Routine	Tcl Proc	Description
NAME	X	X	X	The title line, containing the name of the element and a short, one-line description.
ROUTINES	X			The summary of routines provided by this library, generated automatically by **apigen**.
SYNOPSIS		X	X	For Tcl procedures and scripts, the calling syntax. For C routines, the declaration, generated automatically by **apigen**.
DESCRIPTION	X	X	X	An overall description of the module.
Optional Headings	X	X	X	Additional sections, as needed.
EXAMPLES		X	X	Examples of routine or command invocation.
SMP CONSIDERATIONS	X	X		This section describes any differences between the uniprocessor and the SMP implementation of a library or routine with regard to behavior or use.
64-BIT CONSIDERATIONS	X	X		This section describes any differences between the 32-bit and 64-bit implementation of a library or routine with regard to behavior or use.
INCLUDE FILES	X			The **.h** files that must be **#include**d in code.
RESOURCE FILES			X	Files sourced by the procedure or script.
RETURNS		X	X	The values returned.
ERRNO		X		The list of ERRNO values set.
ERRORS			X	The list of error code or error messages raised.
SEE ALSO	X	X	X	Cross-references to other reference entries, or other user manuals.

Table 10-1 Required Sections for APIGEN markup

10.3 NAME Section

This section is generated automatically. For libraries and other file modules, the text is taken from the one-line title that appears in the first comment block of the file. For C routines, it is taken from the title in the routine's header.

The title takes the following form: the name of the library, routine, procedure, method, or command, followed by a space + hyphen + space, followed by a summary description. Rules:

- The hyphen is a single hyphen—no backslashes or double hyphens.
- The description is a sentence fragment—it must not start with a capital letter and must not end with a period.
- The entire title must fit on a single line and can be no more than 80 characters.

10.3.1 C Libraries

Describe briefly what this collection of routines does. The general format is:

nameLib.c - the such-and-such library

For example:
```
taskLib.c - task management library
```

Be sure to include the filename extension (.c, .s); but note that the apigen process strips it off so that it does not appear in the generated reference entry.

10.3.2 C Routines

For the one-line heading/definition, use the imperative mood and convey action. The general format is:

name - do such and such

For example:

CORRECT:
```
sysMemTop - get the address of the top of memory
```

INCORRECT:
```
sysMemTop - gets the address of the top of memory
sysMemTop - this routine gets the address of the top of memory
```

Do not include the routine parentheses in the heading; the apigen process adds them in; they will appear in the generated reference entry. The title must not continue on a new line, and it must not extend beyond the 80-character line limit.

10.3.3 Tcl Procedures, Scripts and Commands

Use the imperative mood and convey action, as with C routines.

10.4 ROUTINES Section

This section appears in C library entries and is generated automatically. It lists all routines in the library that are not declared LOCAL or marked \NOMANUAL.

For C++ libraries the section is called METHODS. For Tcl modules, the section is called TCL PROCEDURES.

10.5 SYNOPSIS Section

10.5.1 C Routines

For a C routine, this section is the declaration. The section heading is generated automatically and the text is picked up from the declaration in the code, along with the short comments describing each parameter. In unusual cases where the code declaration is not appropriate, a SYNOPSIS section can be entered manually in the routine-header comment block. If “apigen” sees a manually entered SYNOPSIS, it replaces the one encountered in the routine code. This technique can be used for documenting macros as if they were C routines; see Documenting Macros .

CAUTION: When the SYNOPSIS section is added explicitly as described above, the description section that typically follows must be preceded by an explicit DESCRIPTION heading. The DESCRIPTION heading cannot be omitted as it is in other cases. For more information, see DESCRIPTION Section.

10.5.2 Tcl Procedures, Scripts and Commands

For Tcl procedures, scripts, and other commands, this section is the execution syntax; it must be entered manually, using the following conventions:

- Enter the calling syntax and parameters between the tags \ss and \se.
- Show parameters that are optional in square brackets.
- Use the bar character (|) to indicate “or”.
- Represent a variable list of arguments with three dots (...)
- Bracket arguments between angle brackets (< and >) when they are placeholders for user-supplied values.

• If angle brackets are meant to indicate redirection of standard input/output, surround them with space characters.

Example command or script input:

```
# SYNOPSIS
# \ss
# hex [-a <adrs>] [-l] [-v] [-p <pc>] [-s <sp>] <file>
# \se
```

Resulting output:

SYNOPSIS

hex [-a adrs] [-l] [-v] [-p pc] [-s sp] file

10.6 DESCRIPTION Section

This section contains the overall description of the module or routine. Start the description with a sentence that begins This library or This routine or This command as appropriate rather than repeat the name of the facility. Use the word routine, not subroutine or function. The description should be a summary of what the facility does or provides, and in more depth than the title line (NAME line).

The heading word DESCRIPTION can be omitted; apigen puts it in automatically. (More explicitly, if the first text section that appears following the title line is not an all-caps heading, apigen will supply the heading DESCRIPTION automatically, otherwise it will simply output whatever all-caps heading it finds.) However, the DESCRIPTION heading must appear if it is not the first section in the routine or library—for example, if it is preceded by a manually entered SYNOPSIS section.

10.6.1 Parameter Lists

The DESCRIPTION section of a routine or command should list and define all parameters. The automatically published routine declaration includes a short comment for each parameter, which serves as a useful overview or memory jogger. However, these short comments are typically not sufficient for thorough documentation. The parameter list in the DESCRIPTION section should provide more information and detail.

Begin the parameter list with the word “PARAMETERS”. Format the list with the item-list tags \is, \i, and \ie (for more information, see Item Lists (Definition Lists or Terms Lists)).

Each parameter should follow the following format:

- The \i tag followed by the parameter name in angle brackets.

- The next line should start with [in],[out] or [in,out] to indicate an input or output or input and output parameters followed by at least one space.
- Followed by the allowed range of values even if it is all values for the type.
- Followed by a double hyphen (--) separator surrounded by spaces.
- Finishing with a description of how the parameter is used. If optional, detail what the default value is.

For example, consider the routine unixDiskDevCreate( ) with the following declaration:

```
BLK_DEV * unixDiskDevCreate
    (
    char *  unixFile,     /* name of the UNIX file */
    int     bytesPerBlk,  /* number of bytes per block */
    int     blksPerTrack, /* number of blcoks per track */
    int     nBlocks       /* number of blocks on this device */
    )
```

The following shows how the parameters would be described in the DESCRIPTION section:

```
* PARAMETERS
* \is
* \i <unixFile>
* [in] A filename string. -- The name of the UNIX file for the disk device.
*
* \i <bytesPerBlk>
* [in] Positive integer which is a power of 2. -- The size of each logical
* block on the disk. If zero, the default is 512.
*
* \i <blksPerTrack>
* [in] Positive integer. -- The number of blocks on each logical track of
* the disk. If zero, the count of blocks per track is set to <nBlocks>;
* that is, the disk is defined as having only a single track.
*
* \i <nBlocks>
* [in] Positive integer. -- The size of the disk in blocks. If zero, a
* default size is used;
* the default is calculated as the size of the UNIX disk divided by
* the number of bytes per block.
* \ie
```

When generated, the above will appear as follows:

```
PARAMETERS

unixFile
    [in] A filename string. -- The name of the UNIX file for the disk device.
```

```
bytesPerBlk
    [in] Positive integer which is a power of 2. -- The size of each logical
block on the disk. If zero, the default is 512.

blksPerTrack
    [in] Positive integer. -- The number of blocks on each logical track of the
disk. If zero, the count of blocks per track is set to nBlocks; that is, the
disk is defined as having only a single track.

nBlocks
    [in] Positive integer. -- The size of the disk in blocks. If zero, a
default size is used; the default is calculated as the size of the UNIX disk
divided by the number of bytes per block.
```

The text immediately following a parameter is a sentence fragment, not a complete sentence; however, it should start with a capital and end with a period. Do not start the sentence fragment with "specifies the ..."; this is understood. Do not reiterate the name of the parameter. Do not omit articles (the words the, a, and an).

CORRECT:
```
 The name of the UNIX file for the disk device.
```

INCORRECT:
```
 Specifies the name of the UNIX file for the disk device.
```

INCORRECT:
```
 <unixFile> specifies the name of the UNIX file for the disk device.
```

INCORRECT:
```
name of UNIX file for disk device.
```

If the parameter is to have a limited set of values, these should be indicated by inserting the range limit before the double hyphen enclosed in square brackets with space delimiters. For example

```
* \i <fid>
* [in] Positive integer. [0-MAX_FILES] -- The file to be closed.
```

10.7 Additional Optional Sections

Additional sections may be added after the DESCRIPTION section, if needed.

10.8 SMP CONSIDERATIONS

This section describes any differences between the uniprocessor and the SMP implementation of a library or routine with regard to behavior or use.

10.9 64-bit CONSIDERATIONS

This section describes any differences between the 32-bit and 64-bit implementation of a library or routine with regard to behavior or use.

10.10 EXAMPLES

This section should provide contextual examples of how a routine is called or a command is invoked. Examples are a key means of effectively conveying how facilities are meant to be used. Use the \cs and \ce tags to display example code or commands.

10.11 INCLUDE FILES

In C library entries, the heading INCLUDE FILES should provide a comma-separated list of relevant header files. List include files only when users need to #include them in their code explicitly to use the library. For example:

```
INCLUDE FILES: sysLib.h, specialLib.h
```

10.12 RETURNS

Include a RETURNS section in all routines. If there is no return value (as in the case of a void) simply enter “N/A” without a period, as in the following:

```
RETURNS: N/A
```

Mention only true function returns in this section, not values copied to a buffer given as an argument. (However, do describe the latter in the DESCRIPTION section.)

Do not treat return values as complete sentences; the subject and verb are understood. However, always start the return-value statement with a capital and end it with a period; and again, do not use abbreviated English. For example:

```
RETURNS: The address of the top of memory.
```

Keep return statements in present tense, even if the conditions that cause an ERROR or any other return value may be thought of as “past” once ERROR is returned.

CORRECT:

```
RETURNS: OK, or ERROR if memory is not available
```

INCORRECT:

```
RETURNS: OK, or ERROR if memory was not available.
```

In STATUS returns, ERROR must be followed by a qualifying statement. For example:

```
RETURNS: OK, or ERROR if memory is insufficient.
```

In some cases the return value will be “OK, always” and “ERROR, always.”

Do not preface lines of text with extra leading spaces. An input line whose first character is a space will cause a line break.

10.13 ERRNO or ERRORS

For C routines, you must list any errno values set directly or indirectly set by the routine.

For Tcl procedures, list all the error messages or error codes (or both, if necessary) raised in the procedure by the Tcl error command.

Format the list with the item-list tags \is, \i, and \ie (for more information, see Item Lists (Definition

Lists or Terms Lists)) Tag each error name with the \i tag. For example:

```
ERRNO
\is
\i S_objLib_OBJ_ID_ERROR
<msgQId> is invalid.

\i S_objLib_OBJ_Deleted
The message queue was deleted while waiting to receive a message.

\i S_objLib_OBJ_TIMEOUT
No messages were received in <timeout> ticks.
\ie
```

10.14 SEE ALSO

The SEE ALSO section is optional. For C routines, Tcl procedures, and C++ methods, this section is output automatically and includes a reference to the parent library, class, or module name. If the SEE ALSO section is entered explicitly in these cases, the parent name is added automatically to the list of references.

The SEE ALSO section should be the last section of a reference entry. Its purpose is to provide cross-references to other relevant documentation, other reference entries, other Wind River manuals, or non-Wind River documentation.

Do not include manual section numbers using the UNIX parentheses-plus-number scheme; however, do append parentheses to routine names per the documentation standard (see Routine Names):

CORRECT:

```
SEE ALSO: sysLib, vxTas()
```

INCORRECT:

```
SEE ALSO: sysLib(1), vxTas(2)
```

Include cross-references to books by using apigen's \tb tag. Each \tb tag must appear on a separate line. For more information about this tag, see Special Words References to chapters of Wind River manuals should take the form Publication Title: Chapter Name. Do not include the chapter number or page number. For example:

```
SEE ALSO: someLib, anotherLib, someRoutine(),
\tb Tornado User's Guide: Establishing Your Environment,
\tb Motorola MC68020 User's Manual
```

As this example illustrates, cross-references to other reference entries should come first; cross-references to books should come last.

Note the commas at the ends of the first two lines in the above example; the comma is necessary because these references will be run together on output. Alternatively, you can separate the references with blank lines to keep each book on a line by itself—this approach is preferable when there are three or more books. Note also that the list is not terminated with a period.

10.15 DEPRECATED

The DEPRECATED keyword is for an API that is deprecated, and which will be released in a subsequent release. This ensures that the customer knows that an API is slated to be removed or replaced.

```
DEPRECATED: This function has been deprecated. Use the farble() function
instead.
```

10.16 Format and Style

This section describes apigen markup and text-input conventions. The formatting elements are few and straightforward. One of the goals of source-code documentation standards is to promote internal readability; minimal markup supports this.

To work with apigen, source modules and their documentation must be laid out in accordance with a few simple principles, following Wind River's standard layout as described in the coding conventions. The files sampleC.c, sampleTcl.tcl, and sampleCpp.cpp provide C-file and Tcl file examples and further information.

Source-file text should fill out the full line width (80 characters maximum).

Formatting is controlled by special text markup, summarized in this table. Some markup consists of format commands called tags, which begin with a backslash and are followed by letters. Some markup elements are inline; that is, they can appear anywhere in the line of text. Other markup elements must start in text column 1. Format tags, except \lib, always start in column 1. See the note below.

NOTE: For the purpose of describing doc markup, "column 1" really means column 1 of text. In other words, where comment blocks are delineated with a comment indicator at the beginning of each line, such as a C routine, Tcl procedure, shell script, or C++ module, the first column is really the first character after the *+space, #+space, or //+space. Thus, for example, in the doc header for a C routine "column 1" really means column 3.

Markup	Usage	Location
blank line	Paragraph separator.	
initial spaces	Preserve all spaces and line breaks for this input line.	col 1 only
all capital letters	Section heading.	col 1 only
\&all-caps heading	Escape that prevents the special interpretation of an all-caps line as a heading. Suppressed when at the start of an input line, it otherwise generates a plain ampersand.	col 1 only
\h heading	Explicit section heading for use when lowercase characters are required.	col 1 only
\shheading	Subheading, always mixed case with initial caps.	col 1 only
'text' or 'text'	Bold text, for literal names: filenames, commands, keywords, global variables, structure members, and so on. Text can be multiple words if on the same input line.	inline

<text>	Italic text, for arguments, placeholders, emphasis, special terms. Text can be multiple words if on the same input line.	inline
\lib library	Explicit markup for a library name if the name is non- standard (not nameLib, nameDrv, nameShow, nameSio, or if_name).	inline
\tb booktitle	The remainder of the line is a book title reference.	col 1 only
\< \> \` \'	Plain characters <, >, `, and '.	inline
\\	Plain character\ (backslash).	inline
\cs ... \ce	Code example or terminal session - preformatted display in fixed-width.	col 1 only
\bs … \be	Board diagram - preformatted display in reduced fixed-width.	col 1 only
\ss … \se	Syntax display - preformatted display in fixed-width font and containing markup.	col 1 only
\is \i *item* \ie	Item list, also known as a definition list. Each item is a word or phrase followed by an explanation starting on the next line.	col 1 only
\ms \m *mark* \me	Numbered or dash list (marker list).	col 1 only
\ts … \te	Table	col 1 only
|	Column delimiter in a table.	inline
\\|	The character | in a table.	inline
\”	Comment, ignored by apigen.	col 1 only

Table 10-2 Apigen Markup

10.17 Headings

Headings consisting entirely of capital letters, numbers, and hyphens, are automatically flagged as headings, provided they begin with either an upper case letter or a number, contain at least one upper case letter, and have a minimum of 3 characters following the first upper case letter. apigen interprets either of the following types of input as a heading:

A group of all-uppercase words on a line by itself. Underscores and numbers are also permitted. For example:

```
THIS IS A HEADING
This is the text that follows....
```

A group of all-uppercase words at the start of a line and followed by a colon, optionally followed by non-heading text on the same line. For example:

```
THIS IS A HEADING:
This is the text that follows....
```

In special cases where such input should not be interpreted as a heading, the words can be preceded with \&. For example:

```
\&THIS IS NOT A HEADING
```

Occasionally, it is necessary to have a heading in which the rules above do not apply. For such exceptions, use the \h tag. For example:

```
\h ARCHITECTURE NOTE FOR x86
This is the text that follows...
```

Longer reference entries sometimes call for subheadings. Type subheadings in mixed-case on a separate line and apply the \sh tag. Capitalize words following standard capitalization practices for mixed-case headings. (Standard practice: always capitalize the first and last word, and capitalize all other words except articles, prepositions, short conjunctions (fewer than five letters), and the word to.)

For example:

```
\sh Title for a Subheading
This is the text that follows...
```

10.18 Special Words

10.18.1 Literal Names of Commands, Global Variables, Files and Other Elements

The literal names of many system elements are automatically made bold:

- words ending in an empty pair of parentheses (routine names)
- words ending in .c, .h, .o, and .tcl (file types)

- words ending in Lib, Drv, Show, or Sio (library names)
- words beginning with if_ (network interface library names)
- words in all uppercase with one or more underscore characters (constants)
- words that begin with S_ and end with an uppercase string (errno names)

Set off all other literal names with single quotes (‘). These include filenames, tools, commands, operators, C keywords, global variables, structure members, network interfaces, and so on.

Example input:

```
When semTake() returns due to timeout, it sets ’errno’ to S_objLib_OBJ_TIMEOUT
(defined in objLib.h).
```

Resulting output:

When **semTake()** returns due to timeout, it sets **errno** to **S_objLib_OBJ_TIMEOUT**
(defined in **objLib.h**).

Some library names do not end in the standard suffixes listed above; however, they need tobe flagged as libraries in order for the build-time htmlLink utility to create hyperlinks to theirreference entries. Flag such names with the \lib tag. Note that unlike all other tags, the \lib tag can be inline. For example:

```
The interface between BSD IP and the MUX is described in \lib ipProto.
```

10.18.2 Terminal Keys

Enter the names of terminal keys in all uppercase; for example, RETURN, ESC. Prefix the names of control characters with CTRL+; for example, CTRL+C.

10.18.3 Routine Names

Include parentheses with all routine names (except in the one line description of a function). Do not separate the parentheses from the name with a space character (unlike the Wind River convention for code). Do not put a space between the parentheses.

CORRECT:
```
taskSpawn()
```

INCORRECT:
```
taskSpawn( ), taskSpawn (), taskSpawn
```

Even routines generally construed as VxWorks or WindSh "shell commands" must include the parentheses.

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

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

Avoid using the name of a routine (or library, command, or other facility) as the first word in a sentence. Names are case-specific and capitalization must never be changed.

10.18.4 Placeholder Text

A placeholder (also known as a text variable) is a word that represents a value that is to be supplied by the user, such as a command argument, routine parameter, or a portion of a directory path. Text variables are employed most frequently in syntax displays or pathnames. Surround placeholder words with the angle brackets < and >. In the following example, hostOs is a value supplied by the reader:

```
The script is located in the directory 'host/<hostOs>/bin'.
```

Resulting output:
The script is located in the directory **host/***hostOs***/bin**.

10.18.5 Parameters

When referring to routine parameters, treat them as placeholders; that is, bracket the argument name with the angle brackets < and >. For example, consider a routine getName( ) with the following declaration:

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

For the description, you might say something like the following:

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

Although C routine parameters are variables from the perspective of the code's author, they are placeholders from the perspective of the user; therefore we format them as any other placeholder and apply angle brackets. Note, however, that global variables and structure members should be treated as literals, not placeholders; see Literal Names of Commands, Global Variables, Files, and Other Elements

10.18.6 Book References

References to books or book chapters should be tagged with \tb, which sets them in italics. For more information about standards for book references, see SEE ALSO Section Example input:

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

Resulting output:

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

10.18.7 Cross References to Other Reference Entries

Do not use the UNIX-style parentheses-plus-number scheme to cross-reference the documentation sections for libraries and routines:

CORRECT:
```
 sysLib, vxTas()
```

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

10.18.8 Special Terms

When introducing or defining a special term, bracket the word in angle brackets (< and >) at first usage. In output, these words will appear in italics.

10.18.9 Emphasis

In general, avoid applying emphasis to words; a well-cast sentence should be sufficient to convey emphasis. However, if emphasizing a word is necessary, bracket it with the angle brackets (< and

>). In output, these words will appear in italics. Never use uppercase to convey emphasis.

For a summary of how various special text elements are handled, see this table.

Component	Input	Output
library in title	sysLib.c	sysLib
library in text	sysLib.c	sysLib
name with .c extension	sysLib.c	sysLib.c
header file	objLib.h	objLib.h
routine in title	sysMemTop	sysMemTop()
routine in text	sysMemTop()	sysMemTop()
constant,option	INCLUDE_SCSI	INCLUDE_SCSI
other bold elements	errno'	errno
placeholder,routine parameter	<ptid>	ptid
book title	\tb Programmer's Guide	*Programmer's Guide*
emphasis,special terms	<must>	*must*

Table 10-3Special Text Element Handling

10.19 Lists and Tables

Do not use the \cs and \ce tags to build lists or tables.

CAUTION: Nesting of lists is not supported.

10.19.1 Short Word Lists

A simple list of words or short phrases can be created simply by putting each word on a line by itself and indenting it with space characters. Any line that begins with a space causes a line breaks to be preserved for that line only. The line remains part of the paragraph, and so no vertical space is added.

Example input:

```
The first three words in the international phonetic alphabet are:
      alpha
      bravo
      charlie
```

Resulting output:

```
The first three words in the international phonetic alphabet are:
      alpha
      bravo
      charlie
```

Do not use this mechanism for line items that contain more than three words.

10.19.2 Item Lists (Definition Lists or Term Lists)

Item lists, also known a definition lists and terms lists, are lists of special elements—parameters, constants, routines, commands, and so on—and their descriptions. Introduce an item list with the \is tag. Tag each item name with the \i tag, and enter the description on the following line. End the list with \ie. To preserve the readability of the input, separate each item with a blank line; the items are separated by blank space in the output, regardless. Example input:

```
\is
\i 'FIODISKFORMAT'
Formats the entire disk with appropriate hardware track and
sector marks. No file system is initialized on the disk by
this request.

\i 'FIODISKINIT' Initializes a DOS file system on the disk volume.
\ie
```

Resulting output:

```
FIODISKFORMAT Formats the entire disk with appropriate hardware track and
sector marks.
No file system is initialized on the disk by this request.

FIODISKINIT Initializes a DOS file system on the disk volume.
```

10.19.3 Marker Lists (Dash or Numbered Lists)

Use the marker list tags to create lists with a specified “mark,” typically a number or hyphen (apigen does not recognize any symbol for a bullet or en- or em-dash). Introduce a marker list with the \ms tag. Tag each number or hyphen with the \m tag, and enter the text on the following line. End it with \me.

Example input:

```
\ms
\m 1.
Design the program.
\m 2.
Write the code.
\m 3.
Test the system.
\me
```

Resulting output:

```
1. Design the program
2. Write the code.
3. Test the system.
```

10.19.4 Tables

Start a table with the \ts tag and end it with \te. The following conventions describe how tables should be formatted:

- Tables have a heading section and a body section; these are delimited by a horizontal line containing only the characters -(hyphen), + (plus), and | (pipe or bar).
- Table columns are delimited with the bar (|) character. To output a literal | character, escape it with a backslash (\|). Align columns visually so that input is easy to read and maintain.
- A newline marks the end of a row in either the heading or body.

Example input:

```
\ts
Key | Name   | Meaning
----|------------+--------
\&  | ampersand | bitwise AND
\|  | pipe / bar | bitwise OR
#     | pound sign | bitwise NAND
\te
```

Resulting output:

Key	Name	Meaning
&	ampersand	bitwise AND
|	pipe / bar	bitwise OR
#	pound sign	bitwise NAND

10.20 Code Examples, Syntax Display and Diagrams

10.20.1 Code Examples

Display code or terminal input/output with the \cs and \ce tags.

Text between these tags is interpreted as preformatted text; therefore, markup such as angle brackets (< and >) and backslashes (\) is not interpreted, but passed through as entered. Thus markup characters must not be escaped with a backslash.

The one exception is that /@ and @/ are converted to /* and */. In C files, all example comments should be bracketed with /@ and @/. C compilers are generally unfriendly toward nested comments.

Code displays should be indented by four spaces from column 1. The following example shows how a code example would appear in a C routine section:

```
* \cs
*     /@ Get file status information @/
*
*     struct stat statStruct;
*     fd = open (“file”, READ);
*     status = ioctl (fd, FIOFSTATGET, &statStruct);
* \ce
*
```

Resulting output:

```
/* Get file status information */

struct stat statStruct;
fd = open (“file”, READ);
status = ioctl (fd, FIOFSTATGET, &statStruct);
```

Because backslashes are not interpreted as an escape in \cs blocks, the backslash itself must not be escaped. For example:

```
\cs
    -> copy < DOS1:\subdir\file1
\ce
```

Resulting output:

```
> copy < DOS1:\subdir\file1
```

The \cs and \ce tags can also serve as a general mechanism for creating ASCII diagrams.

10.20.2 Command Syntax

Set off command syntax (for example, in shell scripts or Tcl procedures) with the \ss and \se tags. Although nearly the same as a \cs block, the \ss block gives different results. In contrast to \cs, angle-bracket markup (< and >) is interpreted within an \ss block, as long as the angle brackets surround and touch a word. Example input:

```
\ss
deflate < <infile> > <outfile>
\se
```

Resulting output:

```
deflate < infile > outfile
```

10.20.3 Board Diagrams

Bracket plain-text board diagrams with the \bs and \be tags. These tags are exactly like \cs and \ce except that the output text is in a smaller font. Board diagrams are principally used in the BOARD LAYOUT section of a BSP's target.ref file.

10.20.4 Graphics

Insert graphics files using the directive \IMAGE filename. If the output format is HTML, this image file is inserted into an <img> tag. If the output format is anything else, the file is referenced by name in the text. The path of filename must be relative to the directory containing the source file. Currently this directive is used only in BSP target.ref files. Examples:

```
\IMAGE board.jpg
\IMAGE switches.gif
```

10.21 Directives

Directives are apigen controls for special non-formatting actions, such as including information from other files, hiding internal information, or overriding default behavior.

Directives must begin in column 1, and must be the only text on the line. All directives begin with a backslash and the remaining letters are in upper case.

The backslash is a significant deviation from refgen, apigen's predecessor. Most refgen directives required no backslash. However, the backslash requirement makes the markup considerably more resistant to ambiguities.

NOTE: For backward compatibility, directives that were supported by refgen are by default converted internally to the new form, and thus still work. However, they should be changed when encountered and avoided in future work.

The directives recognized by apigen are summarized in this table. Details are provided in the sections that follow.

Directive Name	**Usage**
\APPEND filename	Append contents of another source file.
\EXPAND filename typename	Insert the definition of a typedef'ed struct or enum.
\EXPANDPATH dir:...:dir	Specify search path for files named in \EXPAND directives.
\IMAGE filename	Include an image (a reference to an image file) in the output—normally used only for BSP board diagrams.
\IMPLEMENTS filename	Include the contents of another source file in place of the \IMPLEMENTS directive. Typically used only for class declarations in C++
\INTERNAL [title]	Do not print the following title and section unless the -internal flag is specified when apigen is executed. If no title is given, the title is "INTERNAL."
\IFSET conditions	Generate this routine or library entry only when the -set flag is set to any of the specified conditions. Current legal values for condition are KERNEL and USER.
\IFSET_START conditions \IFSET_END	Output the text between these directives only when the -set flag is set to any of the specified conditions.
\LANGUAGE languagename	Specifies the computer language of the input file. Overrides the filename extension and -lang flag.
\NOMANUAL	Do not generate this routine or library entry unless the -internal flag is specified.
\NOROUTINES	Specifies that all routines are internal, and should be documented only if the -internal flag is specified. Legal only in library comments.

\TITLE name - shortdescription	Overrides the file's title line. Typically used only in target.ref files.

Table 10-4 Summary of apigen Directives

10.21.1 Blocking Text From Publication

The following directives can be used to prevent the publication of specified sections of text. In all cases, the masking action can be overridden by running apigen with the -internal flag. This flag permits all content to be generated, including C routines declared static or LOCAL, which are masked automatically. The -internal flag is typically given to generate documentation for company-internal code reviews.

Note that these directives should not be used to suppress the publication of an entire file. The standard way to prevent a file from being processed is to omit it from the WR makefile variable DOC_FILES.

10.21.1.1 \INTERNAL

\INTERNAL [title]

This directive specifies that the following section is internal documentation and should not be output. An internal section ends with the next heading or the end of the comment block. If title is specified, it becomes the section title if apigen is run with -internal; otherwise the section title is "INTERNAL." Examples:

```
\INTERNAL
\INTERNAL IMPLEMENTATION DETAILS
```

10.21.1.2 \NOMANUAL

\NOMANUAL

This directive suppresses the entire comment block of a routine or library in which it occurs. Routines that are declared static or LOCAL are automatically \NOMANUAL. If all routines in a library are \NOMANUAL, static, or LOCAL, the ROUTINES section of the library entry will be generated with the message "No user-callable routines."

Use of this directive in a library section is rare, but can be useful for masking the library comment block of an \APPENDed file (for example). Putting \NOMANUAL in a library section does not make its routines internal. The standard way to prevent an entire file from being processed is to omit it from a makefile variable.

Example:

```
/**************************************************************
*
* stateReset - reset the state machine
*
* This routine returns the internal state to it initial
* state. It should not be called by users.
...
* \NOMANUAL
*/
```

Although the \NOMANUAL line will be interpreted regardless of where it appears, the standard practice is to place it at the end of the comment block.

10.21.1.3 \NOROUTINES

\NOROUTINES

This directive specifies that all routines in a file are internal. It does not generate a ROUTINES section saying “No user-callable routines” and is thus not equivalent to marking every routine \NOMANUAL. The \NOROUTINES directive is typically used when documenting a utility for which a C file is the source. Example:

```
/* vxencrypt.c - encryption program for loginLib */

/* Copyright (C) 1999 Wind River Systems, Inc. */

/*
modification history
--------------------
01a,31oct99,abc written
*/

/*
\NOROUTINES
SYNOPSIS
\ss
vxencrypt
\se

DESCRIPTION
This tool generates the encrypted equivalent of a
supplied password. ...
*/
```

10.21.2 Conditionals

The following directives allow whole reference entries, or blocks of text within them, to be generated based on the setting of the apigen command-line flag -set.

10.21.2.1 \IFSET

\IFSET conditions

A routine or library entry in which this directive appears is generated only when any condition in the conditions list is specified by the -set flag. Although the \IFSET line will be interpreted regardless of where it appears, the standard practice is to place it at the end of a library or routine's comment block.

10.21.2.2 \IFSET_START

```
\IFSET_START
conditions
\IFSET_END
```

Text bracketed between these two directives is generated only when any condition in the conditions list is specified by the -set flag.

Currently sanctioned values for conditions are KERNEL and USER. Multiple conditions may be specified. If any condition value in the conditions list is specified with the -set flag, the entry or text block will be generated.

For example, the following appears in the documentation comment block for myRoutine( ):

```
*
* \IFSET KERNEL
*
```

A reference entry for myRoutine( ) will be generated if apigen is run as follows:

```
% apigen -set KERNEL CONDITION_2
```

10.21.3 Other Overrides

10.21.3.1 \LANGUAGE

\LANGUAGE languagename

The \LANGUAGE directive specifies the computer language of the input file. Currently valid values are: asm, bsp, c, cpp, ctcl, idl, pcl, perl, shell, and tcl. The letters in the language name can be upper- or lowercase. This directive overrides the file extension and any flags given to apigen. It is useful mainly for CTcl, which is used for C files whose routines must be documented as Tcl procedures. This directive is valid only in the library comments. Example:

```
\LANGUAGE CTcl
```

10.21.3.2 \TITLE

\TITLE name - shortdescription

The \TITLE directive replaces a file's title line (the title in the file's first comment block) with whatever follows the directive. Currently, it is used exclusively in BSPs to give the file a more descriptive name than target.ref. The new name is used in searching for possible hyperlinks to the output file, but does not affect the name of the output file. Example:

```
\TITLE pc386/486 - BSP for PC 386/486
```

10.21.4 Including Content From Other Files

The following directives provide a means for including text, code, or images from other files. Except as noted, when the filename begins with a slash, the path is treated as relative to the directory specified by the WIND_BASE environment variable; otherwise, the path is assumed to be relative to the directory of the file in which the directive occurs. An error is generated if WIND_BASE is not defined.

10.21.4.1 \APPEND

\APPEND filename

This directive appends the contents of the specified file to the end of the current file before processing. Files appended in this way may contain further \APPEND directives. Examples:

```
\APPEND moreRoutines.c
\APPEND subdir/moreRoutines.c
\APPEND /target/src/drv/sio/ambaSio.c
```

10.21.4.2 \EXPAND

\EXPAND filename typename

This directive takes the name of a header file and the name of a typedef'ed struct or enum within that header file. The header file is searched, and if the named typedef is found, its definition is inserted in the reference text as a code display. Examples:

```
\EXPAND taskLib.h WDB_INFO
\EXPAND /target/h/taskLib.h WIND_LCB
```

10.21.4.3 \EXPANDPATH

\EXPANDPATH dirname:...:dirname

This directive specifies a colon-delimited list of directories to be used as base directories, in search order, when looking for files specified in \EXPAND directives. Examples:

```
\EXPANDPATH /target/h:/target/h/sys:.
\EXPANDPATH .:./headers
```

10.21.4.4 \IMAGE

\IMAGE filename

This directive specifies an image file. If the output format is HTML, this image file is inserted into an <img> tag. If the output format is anything else, the file is referenced by name in the text. The path of filename must be relative to the directory containing the source file. Typically this directive is used only in the target.ref files of BSPs. Examples:

```
\IMAGE board.jpg
\IMAGE switches.gif
```

10.21.4.5 \IMPLEMENTS

\IMPLEMENTS filename

This directive specifies a file to be imported at the position of the directive. It is like the \APPEND directive in all respects except where the text is added. It is meant for including C++ header files containing class declarations. Examples:

```
\IMPLEMENTS vectorClass.h
\IMPLEMENTS /target/h/zinc/zaf.hpp
```

11 Versioning in VxWorks

This chapter outlines the standards for managing the version information in VxWorks 7 layers and RPMs. Previous to VxWorks 7, VxWorks was shipped as a monolithic release and the following terms were utilized to categorize releases:

Major Release
VxWorks 6.0 is considered to be a Major Release (relative to the releases: VxWorks AE 1.1 or VxWorks 5.5)

Minor Release
VxWorks 6.9 is considered to be a Minor Release (relative to the releases: VxWorks 6.8, VxWorks 6.7, etc.)

Update Release
VxWorks 6.9.4 is considered to be an Update Release (relative to the releases: VxWorks 6.9.3, VxWorks 6.9.2, etc.)

Service Release
VxWorks 6.9.4.10 is considered to be a Service Release (relative to the releases VxWorks 6.9.4.9, VxWorks 6.9.4.8, etc.)

In general, the version of a layer or RPM is comprised of 4 digits (0 to 9), where the initial version for a layer shall be 1.0.0.0; there are some exceptions as outlined in [11.5], [11.6], and [11.7].

On occasion, in VxWorks 7, the most significant version digit (left most digit) is termed the “major digit” or “first digit”, the second most significant version digit is termed the “minor digit” or “second digit”, the third most significant version digit is termed the “update digit” or “third digit”, and the least significant version digit is termed the “maintenance digit” or “fourth digit”.

11.1 Incrementing the Least Significant Digit (x.x.x.X)

The least significant digit (aka fourth digit) is incremented when all of the following conditions apply:

- Changes are restricted to one or more defect fixes where:
  - No new functionality has been added
  - No new public/protected APIs have been added
- Public/protected APIs have not changed their signature nor behavior
- Future: The exported interface (public or protected) remains binary compatible with all previous 4’th digit releases of the layer assuming that the compiler version used to

generated the binaries does not change. For example, a layer with version 1.0.1.5 has a public and protected interface that is binary compatible with the following previous versions of the layer: 1.0.1.4, 1.0.1.3, 1.0.1.2, 1.0.1.1, and 1.0.1.0; assuming that the same compiler version was used to generated the binaries provided in layer versions 1.0.1.5, 1.0.1.4, 1.0.1.3, 1.0.1.2, 1.0.1.1, and 1.0.1.0.

  - If the only practical implementation for a bug fix breaks binary compatibility, then the third most significant digit (x.x.X.x) should be incremented (even though no new functionality is being introduced)

11.2 Incrementing the Third Most Significant Digit (x.x.X.x)

The third most significant digit (aka third digit) is incremented when one or more of the conditions in the "Incrementing the Least Significant Digit" section doesn't apply, but all of the following conditions apply:

- The added functionality does not introduce a new dependency on another layer/RPM
- Existing public/protected APIs have not changed their signature nor behavior
  - New public/protected APIs are allowed

11.3 Incrementing the Second Most Significant Digit (x.X.x.x)

The second most significant digit (aka second digit) is incremented when one or more of the conditions in the "Incrementing the Third Significant Digit" section doesn't apply, but all of the following conditions apply:

- The added functionality introduces a dependency on an additional (new or existing) layer/RPM
- Existing public/protected APIs have not changed their signature nor behavior
  - New public/protected APIs are allowed
  - Existing public/protected APIs can be marked as deprecated

Declaration of a deprecated C-language API must then be modified to use the _WRS_DEPRECATED() attribute. This will result in compile-time warnings, flagging application code that still references the API.

```
extern STATUS symByValueAndTypeFind (SYMTAB_ID symTblId, ...)
    _WRS_DEPRECATED ("please use symFind() instead");
```

In our code (for example, in the file where the function is implemented) the warning can be suppressed with the CC_WARNINGS_IGNORE_DEPRECATED build flag:

```
CFLAGS_vmBaseLib.o = $(CC_WARNINGS_IGNORE_DEPRECATED)
```

The function documentation should also be marked as DEPRECATED. See the section on the DEPRECATED keyword for more information [**Error! Reference source not found.**].

The replacement APIs (if any) need to be checked into the same release that you are deprecating the old APIs in, at the same time you add the deprecation notice. If an API is to be deprecated and replaced, the deprecation must always coexist with a functioning replacement API.

11.4 Incrementing the Most Significant Digit (X.x.x.x)

The most significant digit (aka first digit) is incremented when the following conditions apply:

- The added functionality results in a public/protected API change that results in a source code incompatibility. The source code incompatibility can be introduced due to:
    - The removal of a public/protected API (which should have been previously marked deprecated via a second digit release of the RPM/layer)
    - The signature or behavior of an public/protected API has changed.

In either case, public/protect API changes must be approved by the architect team.

11.5 Versioning for 3'rd Party Content

For layers that deliver 3'rd party content, the rules outlined above are optionally modified to retain the original version of the 3'rd party content along with suffixing at least one additional digit to track Wind River specific changes, if any, to the content. For example, a layer providing 3'rd party content whose original version was 2.7 would specify an initial layer version of 2.7.0.0, i.e. a pair of additional version digits are added resulting in a total of 4 version digits. The increment of the pair of least significant digits follow the above rules; typically only the least significant digit would be incremented to track Wind River applied bug fixes.

For 3'rd party content whose version already contains 4, or even more, version digits, the layer version shall be suffixed with a single version digit. For example, a layer providing 3'rd party content whose original version was 4.5.11.2 would specify an initial layer version of 4.5.11.2.0. The increment of the least significant version digit is performed to track Wind River applied bug fixes.

11.6 Versioning for "Unsupported" Content

The content of a layer can be classified as "unsupported". This is specified by setting the layer meta-data field LAYER_STATUS to "UNSUPPORTED". The initial layer version for "unsupported" content is 0.1.0.0. The most significant version digit shall remain as "0" as long as the content is classified as "unsupported". The increment of the remaining version digits follow the above rules as stated in section [11.1] to [11.3]. In the event, a source code incompatibility is introduced, an increment of the second most significant digit (aka second digit) is sufficient, i.e. do not increment the most significant digit from “0” to “1”.

The initial version of the layer when the content is no longer classified as “unsupported” shall be 1.0.0.0

11.7 Versioning of VxWorks 7 Product Variants

A “VxWorks 7 Product Variant” typically refers to a snapshot of VxWorks 7 functionality that will be certified. Given that these certified variants have a “long shelf life” and content from the standard VxWorks 7 product is generally not regularly merged into the variant, the software content effectively diverges. Thus in order to clearly distinguish the differing content and the subsequent evolution of a variant product, the layer version from the standard VxWorks 7 product shall be suffixed with a unique character string.

For example, the CORE layer in the standard VxWorks 7 product is versioned at 2.0.1.0. A new branch is created from the “standard VxWorks 7 product branch” when commencing certification activities, to effectively freeze the content of VxWorks 7. All of the layers in this new branch should immediately be updated to apply a “.cert” version suffix (or some other approved suffix to clearly represent the VxWorks 7 Product Variant). So, the CORE layer would attain the version 2.0.1.0.cert. Any subsequent changes to the layer in the new branch shall follow the rules in section [11.1] to section [11.4].