1 Overview ........................................................................................................... 1
  1.1 Introduction ..................................................................................................... 1
  1.2 Terminology .................................................................................................... 2
  1.3 Related Documentation ................................................................................. 4

2 VxWorks and Development Environment Changes .......................... 5
  2.1 Introduction ..................................................................................................... 5
  2.2 Environment Variables and Development Shell ..................................... 6
  2.3 Directory Structure ......................................................................................... 7
    2.3.1 Location of Header Files ......................................................................... 7
  2.4 Build Infrastructure .......................................................................................... 8
    2.4.1 Makefile and make Changes ................................................................. 8
  2.5 Compilers ......................................................................................................... 9
    2.5.1 Default Compiler Change ....................................................................... 9
    2.5.2 Stricter Syntax Checking ....................................................................... 9
    General Guidelines ............................................................................................. 9
    Wind River Compiler Warnings ....................................................................... 10
2.5.3 Extra Compiler Command-line Flags .............................................. 11
2.5.4 Additional Build Flags ............................................................... 11
2.5.5 GNU Compiler Switches ............................................................... 11
2.5.6 objCopy Replaces Wind River Utilities ........................................ 12
2.5.7 Unsupported Optimization Levels ............................................... 12

2.6 C and C++ Libraries ....................................................................... 12
   2.6.1 Dinkum C and C++ Libraries ................................................... 13
   2.6.2 Standard Template Library ....................................................... 13

2.7 IDE and CLI Configuration and Build Tools .................................... 14
   2.7.1 Configuration and Build Using config.h .................................. 14
   2.7.2 vxWorks.st Image Type ............................................................ 15

2.8 VxWorks 6.6 Facilities .................................................................... 15
   2.8.1 Unsupported Facilities ............................................................... 16
   2.8.2 Boot Loaders ............................................................................. 16
     Boot Loader M and N Commands .................................................... 17
     Custom Commands for the Boot Loader Shell ................................. 17
   2.8.3 Default VxWorks Configuration ................................................. 18
   2.8.4 Initialization Routines ............................................................... 18
   2.8.5 Tasks and the TCB ...................................................................... 18
     TCB Access .................................................................................. 18
     Macros Changed .......................................................................... 19
     Task Self-destruction .................................................................. 19
   2.8.6 Task-Specific Variables: taskVarLib and tlsLib ....................... 20
   2.8.7 activeQhead ............................................................................ 20
   2.8.8 VxVMI ..................................................................................... 21
   2.8.9 VxFusion .................................................................................. 21
   2.8.10 Resource Reclamation ............................................................. 21
   2.8.11 Stricter Error Checking on Semaphores .................................. 22
   2.8.12 File System Changes ............................................................... 22
CONTENTS

CBIO and the Extended Block Device (XBD) Interface ...................... 22
File System Monitor ................................................................................. 23
Discontinued Features ............................................................................. 23
VxBus Introduced .................................................................................... 24

2.8.13 POSIX Support .......................................................................................... 24
Kernel and User Environments Decoupled ......................................... 24
VxWorks and POSIX Types .................................................................... 24
mq_attr Structure ..................................................................................... 24

2.8.14 Lazy Initialization Removed ............................................................... 25
2.8.15 Modified Routines .................................................................................... 26
2.8.16 Symbol Table and Module Changes ................................................. 28
2.8.17 Kernel Object Module Loader ............................................................ 28
    Loader errno Values ................................................................................ 28
    Loader Symbol Type Values ................................................................... 29
    Resolving Common Symbols ................................................................. 31
    SDA and Loading Kernel Object Modules ........................................... 31
2.8.18 Target Shell/Kernel Shell ................................................................... 32

2.9 Architecture-Specific Issues .............................................................. 33
    SDA Support for PowerPC ..................................................................... 33

3 Migrating Kernel Applications ........................................................ 37

3.1 Introduction ............................................................................................................. 37

3.2 Migration Checklist ............................................................................................... 38

3.3 Build Infrastructure ............................................................................................... 39
    3.3.1 Recompiling Source Code .................................................................. 39
    3.3.2 Header File Changes ........................................................................... 40
        Type Changes ........................................................................................... 40
        isascii(), toascii() .................................................................................. 40
        Private objLib Macro ............................................................................. 40
    3.3.3 Compiling for Both VxWorks 6.x and VxWorks 5.5 ............................... 40
### 3.4 Unsupported Facilities

- **3.4.1 VxVMI and Migration**
- **VxVMI and RTP Applications**
- **3.4.2 VxFusion and Migration**

### 3.5 System Changes

- **3.5.1 taskSwitchHookAdd( )**
- **3.5.2 taskCreat( )**
- **3.5.3 _func_excBaseHook Daisy Chaining**
- **3.5.4 cacheLib Routines**
- **3.5.5 Private HASH_TBL Structure**
- **3.5.6 vmBaseLib Parameter Change**
- **3.5.7 Changed Virtual Memory Routines**
- **3.5.8 Memory Partition Options**
- **3.5.9 Private Structures and Routines**
- **3.5.10 Deprecated Power Management APIs**
- **3.5.11 Removed APIs**

### 3.6 I/O System Changes

- **I/O Error Code Value Changes**

### 3.7 File System Changes

- **3.7.1 Extended Block Device (XBD) Support**
- **3.7.2 Disk Formatting**
- **3.7.3 ioctl( ) Commands Removed**
- **3.7.4 usrFsLib.c**
- **3.7.5 dosFS**
- **3.7.6 Modified I/O APIs**

### 3.8 POSIX Support Changes

- **POSIX Message Queues**
- **POSIX Thread Support**
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSIX Semaphores</td>
<td>61</td>
</tr>
<tr>
<td>POSIX Thread APIs</td>
<td>62</td>
</tr>
<tr>
<td>POSIX Signal APIs</td>
<td>63</td>
</tr>
<tr>
<td>I/O System Device Control APIs</td>
<td>64</td>
</tr>
<tr>
<td>POSIX-Related Changes in Libraries and APIs</td>
<td>64</td>
</tr>
<tr>
<td>WindView/System Viewer: wvLib</td>
<td>66</td>
</tr>
<tr>
<td>3.9 WindView/System Viewer: wvLib</td>
<td>66</td>
</tr>
<tr>
<td>4 Migrating BSPs and Drivers</td>
<td>67</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>67</td>
</tr>
<tr>
<td>4.2 Migrating BSPs</td>
<td>68</td>
</tr>
<tr>
<td>4.2.1 Planning for BSP Migration</td>
<td>68</td>
</tr>
<tr>
<td>4.2.2 BSP Migration Steps</td>
<td>69</td>
</tr>
<tr>
<td>4.2.3 PowerPC BSPs and SDA</td>
<td>74</td>
</tr>
<tr>
<td>4.2.4 Replacing VM_STATE_xxx Macros</td>
<td>74</td>
</tr>
<tr>
<td>4.2.5 Addressing Compiler Errors and Warnings</td>
<td>76</td>
</tr>
<tr>
<td>Compiler Error and Warning Levels</td>
<td>77</td>
</tr>
<tr>
<td>Example Problems and Solutions</td>
<td>77</td>
</tr>
<tr>
<td>4.2.6 Implementing the M Command</td>
<td>77</td>
</tr>
<tr>
<td>Background</td>
<td>77</td>
</tr>
<tr>
<td>Implementing the M Command</td>
<td>78</td>
</tr>
<tr>
<td>4.2.7 Converting BSP Documentation</td>
<td>79</td>
</tr>
<tr>
<td>Convert target.nr</td>
<td>79</td>
</tr>
<tr>
<td>Convert Other BSP Documentation</td>
<td>79</td>
</tr>
<tr>
<td>Test the BSP Documentation Build</td>
<td>80</td>
</tr>
<tr>
<td>Update the Infrastructure Files</td>
<td>80</td>
</tr>
<tr>
<td>4.3 Migrating Drivers</td>
<td>81</td>
</tr>
<tr>
<td>5 Converting to apigen</td>
<td>83</td>
</tr>
<tr>
<td>Index</td>
<td>85</td>
</tr>
</tbody>
</table>
1.1 Introduction

One of the original goals for VxWorks 6.0 was that existing VxWorks 5.5 kernel applications, BSPs, and drivers should be source-compatible. This goal has not changed for VxWorks 6.6. Most applications, BSPs, and drivers work once they are recompiled for VxWorks 6.6—as long as they do not rely on the few features that are not supported in VxWorks 6.6 (such as VxVMI), for which alternatives must be employed.

This guide focuses on the following topics related to VxWorks 5.5 to VxWorks 6.6 migration:

- Changes in the operating system and development environment.
- Configuring VxWorks 6.6 to provide an operating system platform similar to VxWorks 5.5.
- Migrating kernel applications.
- Migrating BSPs.
Migrating drivers.

For general information about VxWorks development, see 1.3 Related Documentation, p.4.

1.2 Terminology

The following terms describe features of VxWorks, Wind River Workbench, and related development tools.

**VxWorks Terms**

real-time process (RTP)
The user-mode application execution environment provided by VxWorks 6.x. Each real-time process has its own address space, containing the executable program, the program data, execution and exception stacks for each task, the heap, and resources associated with the management of the process itself.

project
A collection of source code, binary files, and build specifications within the Workbench development environment.

workspace
A collection of projects in the Workbench development environment.

component
A VxWorks facility that can be included or excluded, making VxWorks scalable.

error detection and reporting
The runtime facility for detecting errors, logging error information, and making that information available across a warm reboot.

**Project-related Terms**

VxWorks image project
The project type used to configure and build VxWorks images.
downloadable kernel module project
The project type used to manage and build dynamically linked application modules that run in kernel mode.

real-time process project
The project type used to manage and build statically or dynamically linked application modules into an executable that can be dynamically downloaded and run as a real-time process.

shared-library project
The project type used to manage and build dynamically linked shared libraries.

Tool-related Terms

toolchain
A collection of development tools for a specific target CPU; includes the compiler, linker, assembler, and so on.

build specification
User-specified settings and rules that are used to build a project.

graphical user interface
The interactive, graphical face (GUI) of Workbench. Workbench also offers a command-line interface. For more information on Workbench, see the Wind River Workbench User’s Guide. For more information on the command line, see the VxWorks Command-Line Tools User’s Guide.

Release-related Terms

deprecated
Wind River intends to discontinue an API in a future release; it currently works for the benefit of existing applications, but it should not be used in new applications and existing applications should be migrated away from it as soon as convenient.
1.3 Related Documentation

For general information about VxWorks kernel facilities, boot loaders, and kernel application development, see the *VxWorks Kernel Programmer’s Guide*.

For information about migrating kernel applications to real-time process (RTP) applications, as well as RTP application development in general, see the *VxWorks Application Programmer’s Guide*.

For information about migrating block-device drivers from the CBIO interface to the XBD interface, for information about migrating drivers to the current VxBus driver model, and for general information about device driver development, see the *VxWorks Device Driver Developer’s Guide*.

For information about BSP development, see the *VxWorks BSP Developer’s Guide*.

2.1 Introduction

VxWorks 6.6 both maintains backward compatibility with VxWorks 5.5, and also provides extensive new features. This chapter provides an overview of the changes as they affect basic migration. The focus is on differences between VxWorks 5.5 and VxWorks 6.6 as they generally relate to operating system configuration, application migration, BSP migration, and driver migration. For detailed information in migrating applications, see 3. Migrating Kernel Applications. For
detailed information about migrating BSPs and drivers, see 4. Migrating BSPs and Drivers.

In general, VxWorks 5.5 kernel applications, BSPs, and drivers will work with VxWorks 6.6—after recompilation—except in the following cases:

- You have made modifications to your BSP.
- Your application is written in C++.
- You use certain APIs that have changed.
- You use block device drivers that are not fully compatible with the new Extended Block Device facility (XBD) facility.
- You use the dosFs 1.0 API or Wind River’s proprietary VxWorks long name support for dosFs. You use tapeFs or rt11Fs, which are not supported.
- You use VxVMI, which is not supported.
- You use VxFusion, which is not supported.

WARNING: Apart from the exceptions listed above, VxWorks 6.x is source compatible with VxWorks 5.5. Your VxWorks 5.5 BSP, drivers, and applications must be recompiled against VxWorks 6.3.

2.2 Environment Variables and Development Shell

The wrenv script replaces the torVars script that was used to set up the working environment for VxWorks 5.x. For Workbench users, this change is transparent; as with Tornado 2.2, Workbench automatically sets the environment appropriately.

Command-line users should run the wrenv script at the root of their installation to set up the command-line environment and start the VxWorks command shell. For additional information on wrenv, see the VxWorks Command-Line Tools User’s Guide.
2 VxWorks and Development Environment Changes

2.3 Directory Structure

Windows

To start a command shell from Windows, use the new VxWorks 6.x command shell, available from the Start menu:

Start > Programs > Wind River > VxWorks 6.x and General Purpose Technologies > VxWorks Development Shell. The command shell is also available from the command prompt by entering:

C:\> installDir\wrenv.exe -p vxworks-6.x

Solaris/Linux

From Solaris or Linux, the new VxWorks 6.x command shell starts automatically when you type wrenv.sh on the command line.

% installDir/wrenv.sh -p vxworks-6.x

2.3 Directory Structure

The the target and host directory trees are now stored one sub-directory deeper in the install tree, under the vxworks-6.x directory. Also note that the Wind River Compiler (formerly known as Diab) and the Wind River GNU Compiler have moved from the installDirTornado/host directory tree and into their own separate subdirectory trees in VxWorks 6.x. Finally, in VxWorks 6.x, the header files for the network stack have been more finely divided.

These changes impact your makefiles in several ways. First, your binary executable path must include one of the following paths:

WIND_DIAB_PATH=$WIND_HOME/diab/relNum

WIND_GNU_PATH=$WIND_HOME/gnu/relNum-vxworks-6.x

It is important to note that common build utility binaries, some of them from the GNU distribution, are still located in:

$WIND_BASE/host/$WIND_HOST_TYPE/bin

However, the location pointed to by the above string has changed because the value of WIND_BASE is now defined as follows:

WIND_BASE=$WIND_HOME/vxworks-6.x
2.3.1 Location of Header Files

There are two sets of header files provided with VxWorks 6.x. One is for building the kernel, BSPs, and applications that are linked with the kernel. The other is for building VxWorks executables that run with memory protection in user mode, as real-time processes (RTPs). These header files should not be mixed. Each tree exposes the supported set of functionality in the two different operating modes.

The compilers and the Workbench tools know the difference between the two environments and use the appropriate header file tree. If the wrong set of header files is being used, check to be sure you are using the appropriate type of project.

The kernel-mode development header files can be found in:

\textit{installDir/vxworks-6.x/target/h}

The user-mode application development header files can be found in:

\textit{installDir/vxworks-6.x/target/usr/h}

2.4 Build Infrastructure

The build infrastructure has changed. If you use the default build method, these changes are transparent. If, however, you have a customized build system, some changes to your build mechanism may be necessary. See for information about changes in the product installation directory tree, see 2.3 Directory Structure, p.7. For general information, see the Wind River Workbench User’s Guide and the VxWorks Command-Line Tools User’s Guide.

2.4.1 Makefile and make Changes

No makefile changes are required to migrate any Wind River-supported VxWorks 5.5 BSP to VxWorks 6.x. However, if you have made custom modifications to the makefile or if there are differences between a third-party makefile and the standard Wind River makefile, some effort may be required to get a working VxWorks 6.x BSP makefile.

If you need to convert your makefile for this release, Wind River recommends that you start with a standard VxWorks 6.x makefile and get the BSP to build correctly.
When your BSP builds correctly using the standard makefile, apply non-standard additions and modifications from the VxWorks 5.5 version of the makefile one at a time. Be sure to test your modifications using both command-line builds and project builds.

2.5 Compilers

All VxWorks 5.5 user libraries are functionally compatible with VxWorks 6.x. However, your code, including your BSP, must be recompiled with either the Wind River Compiler or the Wind River GNU compiler.

2.5.1 Default Compiler Change

For VxWorks 6.x, the kernel libraries are built with the Wind River Compiler. However, Wind River continues to support both the Wind River Compiler and the Wind River GNU Compiler for building VxWorks applications. For general build information, see the Wind River Workbench User’s Guide and the VxWorks Command-Line Tools User’s Guide.

For information on compiling C++ code with the Wind River Compiler and the Dinkum standard template library, see 2.6 C and C++ Libraries, p.12 or the Wind River Compiler user’s guide for your architecture.

2.5.2 Stricter Syntax Checking

The Wind River-supplied makefiles for VxWorks 6.x implement stricter compiler error checking and issue more warnings and errors than in past releases.

You will probably need to modify some code because both the Wind River Compiler and the GNU compiler are stricter regarding syntax.
General Guidelines

The following issues apply to both compilers.

- Because both compilers are stricter about C++ syntax in particular, you must add "struct" in front of a C++ "enum" where the compiler asks for it.
- Multiline _asm("") macros now require a backslash \ continuation character at the end of the continued line.
- String literals " " spanning two lines are not allowed; use separate pairs of quotes on each line.
- The Dinkum STL has no assumed size for an enum type in a C++ class; unfortunately GNU 3.3 does not auto-convert enum types to the presumed width; thus, a cast is necessary in order to compile.

Wind River Compiler Warnings

The following compilation errors and warnings may occur when using the Wind River Compiler. The first example indicates incorrect code and requires modifying your code. The other warnings may not require modifying your code, but they could indicate problems and should be reviewed and understood.

**warning (etoa:4175): subscript out of range**
This warning flags the case when something like the following is done:

```c
char a[10];
...
a[60] = '\0';
```

**warning (etoa:1583): overflow in constant expression**
This warning points to the case where a variable is being assigned a value that does not legally fit into its type.

**warning (etoa:4167): argument of type foo is incompatible with parameter of type bar**
This warning requires type casting to make sure the types are as the function expects.

**warning (etoa:4513): a value of type foo cannot be assigned to an entity of type bar**
This warning also needs type casting.

**warning (etoa:4171): invalid type conversion**
Bad type conversions could cause silent failures.
warning (etoa:4167): argument of type foo is incompatible with parameter of type bar

warning (etoa:4513): a value of type foo cannot be assigned to an entity of type bar

warning (etoa:1047): trying to assign 'ptr to const' to 'ptr'
   If it is acceptable to use to different types, an explicit cast should be used, so
   the intent is clear. Passing around a const pointer as a non-const pointer is a
   particularly bad case.

warning (etoa:4111): statement is unreachable
   The code should be inspected to make sure the control flow is as you intend.

warning (etoa:4068): integer conversion resulted in a change of sign
   This warning usually results from assigning constants (which are by default of
   type int) to unsigned variables. It can be solved by explicitly casting the
   constant to the appropriate unsigned type.

warning (etoa:4940): missing return statement at end of non-void function bar

warning (etoa:4117): non-void function foo should return a value
   This example could result in erroneous values being returned by the function.

2.5.3 Extra Compiler Command-line Flags

When compiling from the command line in VxWorks 6.x, it is necessary to define
several macro values as command-line switches. For example:

Wind River Compiler:
-DCPU=PPC604 -DTOOL_FAMILY=diab -DTOOL=diab -D${BSPNAME}

Wind River GNU Compiler:
-DCPU=PPC604 -DTOOL_FAMILY=gnu -DTOOL=gnu -D${BSPNAME}

While some Wind River header files may not produce compilation errors without
these definitions, the resulting binary is suspect.

For more information about command-line compilation, see the VxWorks
2.5.4 Additional Build Flags

If you wish to set additional build flags, use the following macros:

Command line use: ADDED_CFLAGS, ADDED_C++FLAGS
Workbench use: CC_ARCH_SPEC

2.5.5 GNU Compiler Switches

The GNU compiler switch -fvec has become -maltivec.

When compiling the kernel with the Wind River GNU Compiler, the -nostdlib option should still be used when linking but note that many of the other compiler options used as workarounds are no longer necessary. Most notably, the GNU 3.3 compiler is capable of producing valid C++ code with automatic template instantiation without any special directives. However, C++ code that is manually instantiated should continue to be compiled with -fno-implicit-templates.

The GNU 3.3 C++ compiler does not support -fvolatile, though this capability is still available for C code. The Wind River Compiler supports the equivalent capability for C++ code and may be used instead.

⚠️ WARNING: Do not mix C++ object files compiled by different Wind River compilers. If you need to use -fvolatile in C++ code builds, you must use the Wind River Compiler to build all your C++ code.

For more information on this and other switches, see the Wind River GNU Compiler documentation.

2.5.6 objCopy Replaces Wind River Utilities

The objcopy utility, rather than the legacy Wind River utilities, is used to convert files to binary and Motorola hex format. (Some utilities may still be present in this release, but their use is deprecated. They have known bugs and have not been updated in several major releases.)
2.5.7 Unsupported Optimization Levels

Optimizing at the -O3 level is not supported in VxWorks 6.x, and was not supported in VxWorks 5.x. In some cases, it may be possible to compile with -O3 optimization and produce a valid object module but this is not supported.

⚠️ CAUTION: The fact that your application compiled successfully with -O3 optimization in the past is not a guarantee that it will do so with the new compiler. Optimizing at the -O3 level is not supported in VxWorks 6.x.

2.6 C and C++ Libraries

VxWorks 6.x is backward compatible for kernel applications written in C, within the limits set by stricter compiler syntax checking and the new C libraries. Applications written in C++ have additional considerations. For more information, see 2.5 Compilers, p.9.

2.6.1 Dinkum C and C++ Libraries

For VxWorks 6.x, Dinkum C and C++ libraries are provided. As shown in Table 2-1, Dinkum libraries are used in all cases except for C language applications in a kernel project, where the VxWorks native C libraries are used.

<table>
<thead>
<tr>
<th>Type of Application</th>
<th>C Language</th>
<th>C++ Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-mode RTP application</td>
<td>Dinkum C libraries</td>
<td>Dinkum C++ and embedded C++ libraries</td>
</tr>
<tr>
<td>Kernel-mode application</td>
<td>VxWorks native libraries</td>
<td>Dinkum C++ and embedded C++ libraries</td>
</tr>
</tbody>
</table>

The VxWorks native C libraries provide routines outside the ANSI specification and provide no support for wide or multibyte characters and no support for locales. For more information, see the various reference entries.
Because the Dinkum C++ libraries are different from the VxWorks 5.5 libraries, applications written in C++ require some migration due to differences in the STL implementation.

### 2.6.2 Standard Template Library

The C++ standard template library (STL) supplied with VxWorks 6.x is the Dinkum library, not the SGI library. Wind River provides no support for using the SGI STL with VxWorks 6.x. For information about the SGI STL, see [http://www.sgi.com/tech/stl/](http://www.sgi.com/tech/stl/).

There are certain differences that must be taken into account when you compile your C++ code with the Dinkum STL.

The Dinkum standard template library is more compliant with the ANSI standard than the SGI library it replaces. As a result, it provides fewer backward-compatible headers (for example, "template.h" versus `<template>`) and it does not include some STL extensions provided by the SGI library.

**Example 2-1 Namespace Information**

The standard used in the past:

```c
#include <map.h>
map<const int, int*, int*>::iterator pPtrIt;
```

becomes:

```c
#include <map>
std::map<int, int*, int*>::iterator pPtrIt;
```

The instructions for using the Dinkum STL with the Wind River Compiler are in the Wind River Compiler user's guide for your architecture.

To use the Dinkum embedded STL with the Wind River Compiler:

- Add `-fno-exceptions -fno-rtti` to your compiler line to disable exceptions.
- Modify `installDir/gnu/3.3.2-vxworks62/x86-win32/include/c++/3.3.2/yvals.h`:

  ```c
  #define __CONFIGURE_EXCEPTIONS 0.
  ```

There is no strict standard for the embedded STL; thus, there is no guarantee that what was included in the SGI embedded STL (no exceptions version) is the same as the Dinkum embedded STL. For more information, see the Dinkumware documentation.
2.7 IDE and CLI Configuration and Build Tools

In the past, VxWorks 5.5 could be configured either by using the Tornado project facility or by modifying \texttt{config.h}. For VxWorks 6.x, projects should be configured in Workbench or using \texttt{vxprj}. You can migrate your \texttt{config.h} changes to VxWorks 6.x by creating a VxWorks image project based on your BSP.

2.7.1 Configuration and Build Using \texttt{config.h}

Wind River recommends using either Workbench or the \texttt{vxprj} command-line facility for configuring and building VxWorks. The legacy method using \texttt{bspDir/config.h} and \texttt{bspDir/make} has been deprecated for most purposes since VxWorks 6.0, and it \emph{cannot} be used for multiprocessor (SMP and AMP) development.

However, the \texttt{config.h} method must still be used for the following:

- Some middleware products (consult the documentation in this regard).
- Boot loaders, when the BSP does not support the \texttt{PROFILE_BOOTAPP} configuration profile.

The \texttt{config.h} method can also be used for uniprocessor BSP development before CDFs have been implemented.

2.7.2 \texttt{vxWorks.st} Image Type

Building a \texttt{vxWorks.st} type of image is not supported by Workbench or the \texttt{vxprj} command-line tool, and the legacy \texttt{config.h} method has been deprecated (see 2.7.1 Configuration and Build Using \texttt{config.h}, p.14).

However, a comparable VxWorks image that includes a standalone symbol table can be configured and built with a Workbench VIP project or \texttt{vxprj}. To do so, configure VxWorks with the \texttt{INCLUDE_STANDALONE_SYM_TBL} and \texttt{INCLUDE_SHELL} components.

In order to prevent the network from starting automatically, add \texttt{INCLUDE_NET_INIT_SKIP} as well. The network can then be started from the shell as follows:

- \verb|-> sp usrNetworkInit|
Note however, that since \texttt{INCLUDE\_NET\_INIT\_SKIP} is incompatible with the WDB agent running \texttt{WDB\_COMM\_END} or \texttt{WDB\_COMM\_NETWORK}, the WDB agent must be excluded, or a non-network back end (such as \texttt{INCLUDE\_WDB\_COMM\_SERIAL}) must be used.

To get closer to the legacy configuration of a \texttt{vxWorks.st} image, components such as the following, can be added as well:

- \texttt{INCLUDE\_SHOW\_ROUTINES}
- \texttt{INCLUDE\_DEBUG}
- \texttt{INCLUDE\_UNLOADER}
- \texttt{INCLUDE\_DISK\_UTIL}

### 2.8 VxWorks 6.6 Facilities

Every effort has been made to minimize the migration effort between VxWorks 5.5 and VxWorks 6.6. To reproduce VxWorks 5.5 functionality with VxWorks 6.x, there are limited changes to system behavior and supported facilities. These changes are discussed in this section.

#### 2.8.1 Unsupported Facilities

In addition to changes to some individual components and parameters, the following VxWorks 5.5 facilities are not supported in VxWorks 6.6:

- dosFs 1.0
- tapeFs
- rt11Fs
- VxVMI
- VxFusion

In the case of dosFs 1.0, the current version of dosFs should be used. For the other file systems, an alternative file system must be selected. For VxVMI, VxWorks provides various memory protection features by default, and applications can be migrated to RTP applications (see \textit{3.4.1 VxVMI and Migration}, p.42).

For VxFusion, alternate communication mechanisms may be implemented (for more information, see \textit{3.4.2 VxFusion and Migration}, p.45).
2.8.2 Boot Loaders

Boot loaders created with VxWorks 5.5 are generally compatible with VxWorks 6.x. Wind River recommends that you try using your VxWorks 5.x boot loader, and if it does not work, then rebuild the boot loader for VxWorks 6.x.

Note the following with regard to using 5.5 boot loaders:

- You can use your VxWorks 5.5 boot loaders with kernel-mode applications that use only 5.5-compatible facilities. Your boot loader must have a 160-byte image path to be fully compatible.
- You can use a custom boot loader that works with VxWorks 5.5. However, the `bootConfig.c` file does not exist in the current release of VxWorks. If `bootConfig.c` was modified for custom features, and the custom features are needed for the current release, see Custom Commands for the Boot Loader Shell, p.17 for information about how to address this issue.

Many, but not all, VxWorks 5.4 boot loaders are compatible with VxWorks 5.5, and are therefore compatible with VxWorks 6.x, subject to the two conditions described previously.

Some, but not all, VxWorks 5.3 boot loaders are VxWorks 5.4- and 5.5-compatible. If the object module format (OMF) and the default load addresses are the same, the boot loader should be VxWorks 5.5 compatible, and therefore VxWorks 6.x compatible.

Boot Loader M and N Commands

The M command is a replacement for the N command, which is maintained for backward compatibility purposes. The commands are provided with the `INCLUDE_BOOT_ETH_MAC_HANDLER` and `INCLUDE_BOOT_ETH_ADR_SET`, respectively. Do not use both components in the same configuration of VxWorks. For information about which of the two commands is supported for a given BSP, consult the BSP reference.

In addition, do not use boot loaders configured with the `INCLUDE_BOOT_ETH_MAC_HANDLER` component to boot VxWorks images produced with any release prior to VxWorks 6.5.
Custom Commands for the Boot Loader Shell

If changes to `bootConfig.c` involved the addition of new boot loader shell commands, they can be ported to the current release with the `INCLUDE_USER_APPL` component.

To add custom commands to the boot loader shell, an initialization routine must be called to register the command processing function. Configure the boot loader with the `INCLUDE_USER_APPL` component, and call the initialization routine using the `usrAppInit()` routine stub, which is in `installDir/vxworks-6.x/target/proj/projDir/usrAppInit.c`.

For example, a command-registration call might look like the following:

```c
/* setup command handlers */
bootCommandHandlerAdd("f", bootAppMemFill, BOOT_CMD_MATCH_STRICT,
                     "f adrs, nbytes, value", "- fill memory");
```

For more information about the `bootCommandHandlerAdd()` routine, see the VxWorks API reference.

2.8.3 Default VxWorks Configuration

VxWorks 6.6 provides the `PROFILE_COMPATIBLE` configuration profile, which a minimal VxWorks 6.6 configuration that is compatible with VxWorks 5.5. For information about configuration profiles, see the VxWorks Kernel Programmer's Guide: Kernel.

2.8.4 Initialization Routines

Some of the `xxxLibInit()` library initialization routines have been published previously in the API reference manuals. These routines are now called automatically by the kernel initialization process and user code is not required to call them. The routines have been marked private and removed from the published documentation.

2.8.5 Tasks and the TCB

For VxWorks 6.x a new library replaces direct access to the TCB. Also, task self-destruction without a helper task is no longer supported.
TCB Access

Currently, direct access to the task control block (WIND_TCB) structure is permitted, but it is deprecated. For this release the taskUtilLib library provides controlled access to fields in the WIND_TCB structure. Wind River advises replacing any code that directly accesses the TCB with routines provided by this library and routines in the taskLib and taskInfo libraries.

The task name is now stored as part of the generic object structure, and can no longer be referenced directly from the TCB. For complete portability use taskName() instead.

The taskUtilLib library has been added to VxWorks and the following APIs are published. This library provides utility routines to access fields in the task control block (WIND_TCB) structure. Wind River advises using these routines when accessing fields in the WIND_TCB structure, because direct access to the structure will not be allowed in a future release. The new routines are the following:

- taskSpareNumAllot()
  This routine allocates the first available spare field in the TCB.

- taskSpareFieldGet()
  This routine gets the spare field of a TCB.

- taskSpareFieldSet()
  This routine sets the spare field of a TCB.

For more information, see the associated reference entries.

Macros Changed

The following macros have been introduced in taskUtilLib.h:

- TASK_SCHED_INFO_GET
  This macro gets the pSchedInfo field in the TCB.

- TASK_SCHED_INFO_SET
  This macro sets the pSchedInfo field in the TCB.

Task Self-destruction

Task self-destruction occurs when the entry point function specified to taskSpawn() returns, or when a task performs one of the following functions:
In VxWorks 5.5, a task self-destruct is handled by the exception task (tExcTask), if it exists. In other words, if the INCLUDE_EXC_TASK component is configured into the VxWorks image, the taskDelete() routine refers the self destruction to the tExcTask. If the tExcTask task does not exist, taskDelete() performs the self destruction in the context of the task itself. In order to support the task in destroying itself without a helper task, certain non-standard memory management techniques were introduced.

VxWorks 6.x introduces a memory partition and a heap instrumentation library (INCLUDE_MEM_EDR) to help detect common programming errors such as double-freeing an allocated block, freeing or reallocating an invalid pointer, writing into freed buffers, memory leaks, and so forth. In this new context, task self-destruction as practiced in VxWorks 5.5 results in false alarms from the memory partition and heap instrumentation library.

For these reasons, in VxWorks 6.x, task self-destruction without a helper task is no longer supported. In addition, tExcTask no longer supports task self-destruction. Instead, a new task named tJobTask is used; it executes at the priority of the task performing one of the self-destruct functions. To include support for task self-destruction, the INCLUDE_JOB_TASK component must be configured into the kernel; it is included by default.

If the INCLUDE_JOB_TASK component is not included in the kernel, any task that attempts to self-destruct is left in a suspended state. The task can then be deleted by another task using taskDelete().

An important consequence of this change is that the root task (tRootTask) self-destructs once it completes initialization of the kernel and applications. Thus, if the INCLUDE_JOB_TASK component is not included, the tRootTask remains in a suspended state. An application task spawned by USR_APPL_INIT (BSPs) or usrAppInit() (projects) is required to explicitly delete the root task:

```c
rootTid = taskNameToId ("tRootTask");
if (rootTid != ERROR)
    taskDelete (rootTid);
```

### 2.8.6 Task-Specific Variables: taskVarLib and tlsLib

The taskVarLib and tlsLib facilities are maintained primarily for backward-compatibility. They are not compatible with the SMP configuration of VxWorks, and their use is not recommended. In addition to being incompatible
with VxWorks SMP, the taskVarLib and tlsLib facilities increase task context-switch times.

Note that tlsLib is now named tlsOldLib.

The __thread storage class variables can be used for both uniprocessor and SMP configurations of VxWorks, and Wind River recommends its use in both cases as the best method of providing task-specific variables. For more information, see the VxWorks Kernel Programmer’s Guide: Multitasking.

2.8.7 activeQhead

activeQhead no longer exists and cannot be used to determine if tasking has started. Example 2-2 shows the previous and current methods:

Example 2-2 Determining if Tasking Has Started

The following method was used in previous releases:

```c
if (Q_FIRST (&activeQHead) == NULL) /* pre kernel */
    reboot (BOOT_WARM_AUTOBOOT); /* better reboot */
```

Because activeQHead no longer exists, instances where this method was used in the past must be replaced with the method shown below:

```c
if (taskIdCurrent == NULL) /* pre kernel */
    reboot (BOOT_WARM_AUTOBOOT); /* better reboot */
```

2.8.8 VxVMI

In VxWorks 5.x, VxVMI provided a form of MMU protection. This facility is not supported in VxWorks 6.x, which provides superior facilities with kernel hardening and real-time processes. For information on migrating from VxVMI to VxWorks 6.x, see 3.4.1 VxVMI and Migration, p.42.

For information about real-time processes, and about migrating kernel applications to RTP applications, see the VxWorks Application Programmer’s Guide: Applications and Processes and the VxWorks Application Programmer’s Guide: Kernel to RTP Migration.
2.8.9 VxFusion

VxFusion is not supported in VxWorks 6.0. Wind River advises customers to migrate their applications to message channels over TIPC, which provides superior functionality. For information about message channels, see the VxWorks Kernel Programmer’s Guide: Message Channels. For information about TIPC, see the Wind River TIPC Programmer’s Guide.

2.8.10 Resource Reclamation

The introduction of resource reclamation has minimal impact on drivers and BSPs. Objects such as semaphores, message queues, and tasks created by a process are owned by the process that created them. If a driver or a BSP creates objects, they are typically created during the kernel initialization phase, which occurs in the context of the kernel. Thus, there should be no concern that an object can be unexpectedly deleted, because the kernel is never deleted.

Resource reclamation can be an issue for any kernel routine that is executed as a result of a user task performing a system call. Any objects that are created are owned by the calling process, and are deleted when the process terminates. If any subsystem creates objects that must survive after the creating process terminates, the objOwnerSet() routine should be used to assign the object to the kernel, or to some other process that persists after the creating process terminates.

Resource reclamation is also an issue for kernel task create hooks. For most scenarios, a task create hook routine need not worry about the ownership of objects. The only case that presents a problem is when a process (for example, applA running in process A) performs an rtpSpawn(). The rtpSpawn() system call creates an initial task (iApplA) in the newly created process. However, the taskSpawn() of the iApplA occurs in the context of process A; thus, the task create hooks also execute in the context of process A. Any objects created in a task create hook for the iApplA are owned by process A. If process A terminates, the objects will be deleted. Use objOwnerSet() to set ownership of newly created objects to the new process as recommended in the reference entry for the kernel version of taskCreateHookAdd().

A similar issue exists for process post-create hooks. Use objOwnerSet() to set ownership of newly created objects to the new process as recommended in the reference entry for rtpPostCreateHook().
2.8.11 Stricter Error Checking on Semaphores

The semaphore create routines for all types of semaphores, for example `semBCreate()`, have been updated to perform stricter error checking on the options passed to the routine. In VxWorks 5.x, reserved bits are not checked. For VxWorks 6.x, all options passed are checked against allowed options. Therefore, code that worked in the past may nevertheless have passed disallowed options; now, when checking identifies the disallowed options, such code generates an `errno` with the value `S_semLib_INVALID_OPTION`.

2.8.12 File System Changes

CBIO and the Extended Block Device (XBD) Interface

In previous systems, a stack consisting of some number of CBIO modules with a file system component on top of it was created at initialization time. This stack monitored insertion and removal of disks such as floppies or CD-ROMs, mounting and unmounting the file system as disks were inserted and removed. The CBIO stack has been replaced by the extended block device (XBD) which serves many of the purposes of CBIO, but which also provides for insertion and removal as well as safe dynamic creation and deletion.

VxWorks must be configured with the special components for any drivers that were designed to work with the 5.5 CBIO interface. For more information in this regard, see 3.7.1 Extended Block Device (XBD) Support, p.53 the VxWorks Kernel Programmer’s Guide: I/O System.

File System Monitor

When a device insertion is detected, a new component, the file system monitor, automatically detects the file system type of the inserted device and creates a file system stack with the appropriate file system at the top. There is no longer the requirement to explicitly create file systems by invoking a creation routine; this is handled by the file system monitor.
Discontinued Features

**dosFs 1.0 Support**

dosFs 1.0 is no longer supported in this release. For information on migrating from dosFs 1.0 to dosFs 2.0, see **3.7.5 dosFS**, p.58.

**dosFs Long Filename Support**

Wind River’s proprietary long name support for dosFs is not fully supported in this release. Wind River advises customers to migrate to the Microsoft standard of long names.

**tapeFs**

The tapeFs file system is no longer supported in this release.

Deprecated Features

**RAM Disk**

The `BLK_DEV`-based RAM disk is deprecated. The XBD-based RAM disk should be used instead. For more information, see **3.7.1 Extended Block Device (XBD) Support**, p.53.

**VxWorks CBIO Interface**

The VxWorks CBIO interface is replaced by the XBD facility. For information on migrating to XBD, see **3.7.1 Extended Block Device (XBD) Support**, p.53.

**VxBus Introduced**

VxWorks 6.2 introduces a new bus and device model. For more information, see **4.3 Migrating Drivers**, p.81.

2.8.13 **POSIX Support**

**Kernel and User Environments Decoupled**

The header files `unistd.h` and `limits.h` are now different in the two environments. In the user environment, changes reflect PSE52 compliance and involve changes to
VxWorks and POSIX Types

In order to prevent the native VxWorks symbols from polluting the namespace of a conforming POSIX application, the VxWorks types used in POSIX header files have been prepended with either _Vx_ or _VX_. The original VxWorks types are still defined and used by the native VxWorks header files. For example, the new type is _VX_SEM_ID_, but _SEM_ID_ is also still available for non-POSIX applications. This change means that a VxWorks application that uses POSIX header files may not get some of the VxWorks types and macros it used to get indirectly by including the POSIX header files. You may have to explicitly include the VxWorks header files that define these types and macros.

mq_attr Structure

The mq_attr structure has been updated in both the kernel and application spaces to use type long to conform to PSE52. Previously the type for the fields in the structure was size_t.

In previous releases the structure was defined as follows:

```c
struct mq_attr
{
    size_t    mq_maxmsg;
    size_t    mq_msgsize;
    unsigned  mq_flags;
    size_t    mq_curmsgs;
};
```

In VxWorks 6.3 the structure is defined as follows:

```c
struct mq_attr
{
    long      mq_maxmsg;
    long      mq_msgsize;
    long      mq_flags;
    long    mq_curmsgs;
};
```

The size of the old and new types is the same, so the change should be transparent.
2.8.14 Lazy Initialization Removed

In the past, it was possible for some kernel facilities (message queues, semaphores, watchdogs, tasks) to be used in an application without either including the facility components (such as INCLUDE_SEM_MUTEX) in your project or initializing the library in your code. This was possible because the compiler pulled in the library when it found the call in the code and the routines for these kernel facilities auto-initialized the associated libraries when they were called. In other words, if \texttt{semMCreate()} was called on a mutex semaphore, the \texttt{semMCreate()} API checked and initialized the necessary semaphore library, before making the actual \texttt{semMCreate()} call.

Checking initialization status before creating every object was expensive, and also introduced the possibility that the library could be initialized at any time, leading to indeterminacy and sometimes unacceptable latency.

This scenario is no longer supported. If you use message queues, semaphores, watchdogs, or tasks, you must include the appropriate components in your project. Once you do this, VxWorks automatically initializes the libraries at startup.

In most cases, you will already have the appropriate components included. However, you should check to confirm that this is the case. Table 2-2 lists the libraries and their associated components.

### Table 2-2 Libraries and Associated Components

<table>
<thead>
<tr>
<th>Library</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>taskLib</td>
<td>INCLUDE_KERNEL (always present)</td>
</tr>
<tr>
<td>semBLib</td>
<td>INCLUDE_SEM_BINARY,</td>
</tr>
<tr>
<td></td>
<td>INCLUDE_SEM_BINARY_CREATE</td>
</tr>
<tr>
<td>semCLib</td>
<td>INCLUDE_SEM_COUNTING,</td>
</tr>
<tr>
<td></td>
<td>INCLUDE_SEM_COUNTING_CREATE</td>
</tr>
<tr>
<td>semMLib</td>
<td>INCLUDE_SEM_MUTEX,</td>
</tr>
<tr>
<td></td>
<td>INCLUDE_SEM_MUTEX_CREATE</td>
</tr>
<tr>
<td>msgQLib</td>
<td>INCLUDE_MSG_Q,</td>
</tr>
<tr>
<td></td>
<td>INCLUDE_MSG_Q_DELETE</td>
</tr>
<tr>
<td>wdLib</td>
<td>INCLUDE_WATCHDOGS</td>
</tr>
</tbody>
</table>
2.8.15 Modified Routines

In addition to routines discussed in the context of specific technologies elsewhere in this guide, the following routines have changed.

Private Routines

The following routines are no longer public:

- sigeventCreate()
- sigeventInit()
- sigeventNotify()
- sigeventSigOverrunGet()

They should not have been published in the past, and are not required by users in order to use the POSIX signal events facilities.

Modified Kernel Routines

The following routines have been moved from memLib (INCLUDE_MEM_MGR_FULL) to memInfo (new component INCLUDE_MEM_MGR_INFO):

- memPartInfoGet()
- memPartFindMax()
- memInfoGet()
- memFindMax()

symLib APIs

The symLib routines symFindByValue() and symFindByValueAndType() have been deprecated since the VxWorks 6.0 release and will be removed in a future release. The following alternatives should be used:

- For symFindByValue(), use symByValueFind() instead.
- For symFindByValueAndType(), use symByValueAndTypeFind() instead.

Semaphore APIs

The non-public semCCoreInit(), semBCoreInit(), and semMCoreInit() APIs are no longer available. Use the following alternatives:

- For semCCoreInit(), use semCInit() instead.
- For semBCoreInit(), use semBInit() instead.
- For semMCoreInit(), use semMInit() instead.
The `semOLib` library has been removed. It provided the following routines:

- `semOLibInit()`
- `semCreate()`
- `semInit()`
- `semOTake()`
- `semClear()`

**Shell APIs**

The shell routines `shellInit()`, `shell()`, and `shellOrigStdSet()` have been deprecated since the VxWorks 6.0 release, and are no longer available. Use the following alternatives:

- For `shellInit()`, use `shellGenericInit()` instead.
- For `shell()`, use `shellGenericInit()` instead.
- For `shellOrigStdSet()`, use `shellInOutSet()` instead.

**Other APIs**

The `pipe()` routine has been removed. It was implemented previously as a stub routine that returned `ERROR`.

In addition, the following individual routines have also been removed:

- `shellInit()`
- `shell()`
- `shellOrigStdSet()`

### 2.8.16 Symbol Table and Module Changes

Symbol tables and modules are no longer objects.

- The `objShow()` and `show()` APIs no longer work with a module ID or a symbol table ID. For modules, use `moduleShow()`. For symbol tables, use `lkup()` with no arguments; it prints the contents of the symbol table. The new routine `symShow()` can be used to display general information about a symbol table.

- If an invalid ID is provided to symbol table or module APIs, the `errno` is no longer set to `S_objLib_OBJ_ID_ERROR`. Instead, it is now set to either `S_symLib_INVALID_SYMTAB_ID` or `S_moduleLib_INVALID_MODULE_ID`. 
2.8.17 Kernel Object Module Loader

The kernel object module loader and its supporting libraries (including loadLib, unldLib, symLib, and moduleLib) have undergone an internal overhaul for VxWorks 6.x. For the most part, changes are transparent to users:

- APIs have the same signatures and the same options (or additional ones). All API behavior remains the same except where it changed due to a bug fix. For more information, see the online support Web page at http://www.windriver.com/support.
- The same errno is returned by the VxWorks 6.x loader for most error conditions as was returned by the VxWorks 5.5 loader.

Changes to errno values returned and to the loader symbol values are discussed in the following sections. However, the loader, unloader, module, and symbol libraries are essentially backward compatible, after you re-compile with new headers.

Loader errno Values

Certain errno values returned have changed in order to provide better information about the error.

- In VxWorks 6.x the errno that is set when there is a relocation overflow is now S_loadElfLib_RELOCATION_OFFSET_TOO_LARGE. In all other relocation error cases, the errno value is S_loadElfLib_RELOC, which is the same as was set by the VxWorks 5.5 loader.
- In certain cases S_loadElfLib_SCN_READ, S_loadElfLib_SHDR_READ, and S_loadElfLib_READ_SECTIONS are set by the VxWorks 5.5 loader when there is an error reading the header of the module to download. The VxWorks 6.x loader now keeps the errno that is set by underlying routines while executing the module to provide more information about the cause of the error.

Loader Symbol Type Values

The single notable exception to full backward compatibility of the loader libraries is that the values used to represent symbol types have changed. In general, changes to values associated with macros do not violate backward compatibility,
because VxWorks is not binary backward compatible. Code that uses only the symbol names of the macros should not encounter any compatibility problems.

However, the previous set of values used for representing symbol types were almost, but not quite, usable as a bit map. The result was that sometimes code could be forced to look at the actual numeric values contained in a symbol type variable to try to deduce the real meaning of the variable.

Any code that used the numeric values of these macros must be modified to use only the symbolic names. The new set of values makes it possible to always work only with the symbolic names; thus, this problem no longer occurs.

The previous (VxWorks 5.5) values are as follows:

```c
#define SYM_UNDF        0x0     /* undefined */
#define SYM_LOCAL       0x0     /* local */
#define SYM_GLOBAL      0x1     /* global (external) */
#define SYM_ABS         0x2     /* absolute */
#define SYM_TEXT        0x4     /* text */
#define SYM_DATA        0x6     /* data */
#define SYM_BSS         0x8     /* bss */
#define SYM_COMM        0x12    /* common symbol */
#define SYM_SDA         0x40    /* symbols related to a */
#define SYM_SDA2        0x80    /* symbols related to a */
#define SYM_THUMB       0x40    /* Thumb function */
```

The new (VxWorks 6.x) values are as follows:

```c
#define SYM_UNDF        0x0     /* undefined (lowest 8 bits only) */
#define SYM_GLOBAL      0x1     /* global */
#define SYM_ABS         0x2     /* absolute */
#define SYM_TEXT        0x4     /* text */
#define SYM_DATA        0x8     /* data */
#define SYM_BSS         0x10    /* bss */
#define SYM_COMM        0x20    /* common symbol */
#define SYM_LOCAL       0x40    /* local */
#define SYM_THUMB       0x80    /* Thumb function */
```

With the VxWorks 5.5 values, some bits could be meaningfully OR’d together (for instance, the `global` symbol type could be meaningfully OR’d with any other symbol type). However, if certain symbol types were OR’d together, the original meaning could be lost. For example, with the old values:

```
SYM_ABS | SYM_TEXT = 0x6 = SYM_DATA.
```

The VxWorks 6.x values work as a true bit field. Each bit carries one (and only one) possible meaning. The symbol masks should be used to avoid these bit-field and
compatibility problems. You should test for symbol types with the following macros, defined in symbol.h:

```c
#define SYM_IS_UNDF(symType)
#define SYM_IS_GLOBAL(symType)
#define SYM_IS_LOCAL(symType)
#define SYM_IS_TEXT(symType)
#define SYM_IS_DATA(symType)
#define SYM_IS_BSS(symType)
#define SYM_IS_ABS(symType)
#define SYM_IS_COMMON(symType)
```

⚠️ **WARNING:** Code that only uses the symbolic names of these macros should not encounter any problems. However, any code that uses the numeric values of these macros must be modified to use only the symbolic names.

### Resolving Common Symbols

Resolving common symbols has changed between VxWorks 5.5 and VxWorks 6.x. In the past, if you used the `LOAD_COMMON_MATCH_USER` or `LOAD_COMMON_MATCH_ALL` loader options and there were several matches, bss symbols would be picked first, then data symbols. Starting with VxWorks 6.1, the matching order is data symbols, then bss symbols.

### SDA and Loading Kernel Object Modules

Small data area (SDA) support is provided for the VxWorks kernel programming environment with the PowerPC architecture (it is also supported in the user-space programming environment). The SDA construct is defined by the PowerPC Embedded Application Binary Interface (EABI) specification. SDA is designed to take advantage of base plus displacement addressing mode, which provides a more memory-efficient way of accessing a variable and better performance.

If a kernel module is built with SDA, the loader will not load it, but generates an error message. The error messages for the kernel object module loader and the host loader (respectively), are as follows:
S_loadLib_SDA_NOT_SUPPORTED

WTX_ERR_LOADER_SDA_NOT_SUPPORTED

Workbench also displays the following error message: “WTX Loader Error: dynamic loading of modules with SDA (Small Data Area) sections is not supported; check your build rules and make sure your module does not contain any SDA section or relocation. The loader cannot perform SDA relocation.”

Modules with SDA should be statically linked with the kernel. For more information about VxWorks SDA support, see SDA Support for PowerPC, p.33.

The Wind River Compiler (diab) assembler flag -Xwarn-use-greg can be used to generate the following warning if code accesses the SDA reserved registers:

Xwarn-use-greg=0x2004

In addition, the SDA_DISABLE makefile variable can be used to disable SDA, as follows:

SDA_DISABLE=TRUE

2.8.18 Target Shell/Kernel Shell

With VxWorks 6.0, the name of the target shell was changed to the kernel shell. Several changes in the kernel shell reflect process support. The behavior of multiple sessions can be restored to the VxWorks 5.5 behavior. unld() and reld() are now supported in the shell only.

Multiple Sessions

It is possible to have several kernel shell sessions running at the same time on different terminals (such as the console, VIO, telnet, rlogin, and so on). Each session has a different environment (user ID, current path, prompt) and does not affect the global standard I/O.

The initial shell script is launched on a different shell session from the initial kernel shell session on the console. A side effect of this behavior is that when shellPromptSet() is called in the shell script, the scope is limited to the script shell session and does not affect the prompt of other sessions. The default prompt can be changed with shellPromptFmtDftSet().

To restore VxWorks 5.5 behavior, set the shell component parameter SHELL_COMPATIBLE to TRUE when creating a project.
unld( ) and reld( )

The unld( ) and reld( ) routines have been moved from the file unldLib.c to usrLib.c. To include these routines in your image, use the component INCLUDE_SHELL.

These routines were deprecated in VxWorks 5.5 and can no longer be called directly from programs; they are now for use from the shell only. For unloading from within a program, use either unldByModuleId() or unldByNameAndPath(). For reloading from within a program, you must first unload the module (using unldByModuleId() or unldByNameAndPath()) and then load it again using one of the load routines: loadModule() or loadModuleAt().

2.9 Architecture-Specific Issues

Architecture-specific migration issues, as well as all other architecture-specific issues, are covered in the VxWorks Architecture Supplement. Some concerns include the following:

PowerPC:
- PHYS_ADDR is now an unsigned long long type.
- PPC603 now uses a two-level translation table instead of a hash table. (The PPC604 family still uses both a hash table and translation tables.)

Intel Architecture
- Cacheability and type of cacheability are now treated as one entity; thus VM_STATE_WBACK and VM_STATE_WBACK_NOT cannot be combined with VM_STATE_CACHEABLE or VM_STATE_CACHEABLE_NOT.

MIPS
- PHYS_ADDR is now an unsigned long long type.

ARM
- Boot offsets must move in order to support kernel hardening. By default, BSPs are built with T2_BOOTROM_COMPATIBILITY. Enable kernel hardening by defining INCLUDE_KERNEL_HARDENING and un-defining T2_BOOTROM_COMPATIBILITY. For additional information, see the architecture supplement.
XScale
Boot offsets must move in order to support kernel hardening. By default, BSPs are built with T2_BOOTROM_COMPATIBILITY. Enable kernel hardening by defining INCLUDE_KERNEL_HARDENING and un-defining T2_BOOTROM_COMPATIBILITY. For additional information, see the architecture supplement.

ColdFire
Support for the ColdFire architecture is added in VxWorks 6.3. Note that none of the boards supported was supported in Tornado 2.2.x. You must use a VxWorks 6.3 BSP and create a VxWorks 6.3 boot loader.

SDA Support for PowerPC
Small data area (SDA) support has been implemented for the VxWorks kernel programming environment with the PowerPC architecture (it is also currently supported in the user-space programming environment and has been in the past). The SDA construct is defined by the PowerPC Embedded Application Binary Interface (EABI) specification. SDA is designed to take advantage of base plus displacement addressing mode, which provides a more memory-efficient way of accessing a variable and better performance. For information about the impact of SDA support on other features, see SDA and Loading Kernel Object Modules, p.34 and SDA and Custom PowerPC BSPs, p.34.

SDA and Loading Kernel Object Modules
If a kernel module is built with SDA, the loader will not load it, but generates an error message. The error messages for the kernel object module loader and the host loader (respectively), are as follows:

- S_loadLib_SDA_NOT_SUPPORTED
- WTX_ERR_LOADER_SDA_NOT_SUPPORTED

Workbench also displays the following error message: “WTX Loader Error: dynamic loading of modules with SDA (Small Data Area) sections is not supported; check your build rules and make sure your module does not contain any SDA section or relocation. The loader cannot perform SDA relocation.”

Modules with SDA should be statically linked with the kernel.

The the Wind River Compiler (diab) assembler flag -Xwarn-use-greg can be used to generate the following warning if code accesses the SDA reserved registers:

-Xwarn-use-greg=0x2004
In addition, the SDA_DISABLE makefile variable can be used to disable SDA, as follows:

```
SDA_DISABLE=TRUE
```

**SDA and Custom PowerPC BSPs**

The VxWorks kernel initialization process now initializes PowerPC SDA/SDA2 base registers. If a custom BSP invokes a C function from within _sysInit( )—that is, before the invocation of _usrInit( )—the SDA/SDA2 base registers need to be initialized prior to calling the C function. They should be initialized as follows:

```
lis     r2, HI(_SDA2_BASE_)
ori     r2, r2, LO(_SDA2_BASE_)
lis     r13, HI(_SDA_BASE_)
ori     r13, r13, LO(_SDA_BASE_)
```

Note that the Wind River Compiler (diab) assembler flag -Xwarn-use-greg can be used to generate the following warning if code accesses the SDA reserved registers:

```
Xwarn-use-greg=0x2004
```
3.1 Introduction

VxWorks 5.5 application software can be migrated to the VxWorks 6.x kernel without modification, provided it uses standard features of the 5.5 release and does not include components or facilities that are obsolete. For information about individual components and facilities that are not supported see 2.8 VxWorks 6.6 Facilities, p.15.
This chapter provides information specific to migrating VxWorks 5.5 kernel applications to the VxWorks 6.6 kernel. You should also consult 2. VxWorks and Development Environment Changes for general information about 5.5-to-6.6 migration, much of which is relevant to application migration as well.

For information about migrating kernel applications to user-mode, real-time process (RTP) applications, see the VxWorks Application Programmer’s Guide: Kernel to RTP Migration.

For information on using new features have been introduced with the VxWorks 6.x releases, see the VxWorks Kernel Programmer’s Guide and the VxWorks Application Programmer’s Guide.

3.2 Migration Checklist

Once recompiled, kernel applications from VxWorks 5.5 should run in the VxWorks 6.x kernel, unless the application uses non-standard libraries or compiler options or the WDB API.

⚠️ WARNING: The checklist below assumes a Tornado 2.2 and VxWorks 5.5 baseline. If your applications are based on VxWorks 5.4, you must first migrate to VxWorks 5.5 using the Tornado Migration Guide, 2.2.

The checklist that follows provides guidance for assessing what aspects of existing applications might require additional effort to migrate to kernel projects. If the answer is no to all questions, there should not be any migration issues.

- Do your applications utilize Wind River private APIs (libraries that are not documented as part of the VxWorks kernel)? Examples are aioSysDrv, avlLib, avlUintLib, cbioLib, classLib, dcacheCbio, dpartCbio, hashLib, inflateLib, objLib, passFsLib, poolLib, and ramDiskCbio. Not all libraries defined in installDirTornado/target/h in earlier releases are considered public APIs.

⚠️ NOTE: As in earlier releases, private APIs continue to be undocumented. These APIs can change without notification as they are internal. If you have used private APIs in the past, you should migrate to public APIs, which are documented in the VxWorks reference entries.
• Do you utilize any special Wind River Compiler options, pragmas, or in-line assembly code?
  If so, use standard Wind River macros for portability. For more information, see the *Wind River Compiler User’s Guide*.

• Do you utilize any special GNU compiler options, pragmas, or in-line assembly code?
  If so, use standard Wind River macros for portability. For more information, see the Wind River Compiler documentation.

• Does your product use any WTX tool interface APIs such as *wtxtcl* or the Tornado C API?
  Changes have occurred in these areas. For more information, see the *Wind River Workbench Migration Guide*.

• Does your application include any make rule changes?
  Changes to make rules must be migrated manually.

• Does your application use the following POSIX thread APIs:
  
  ```
  pthread_attr_setscope()
  pthread_cond_init()
  pthread_create()
  pthread_mutex_init()
  pthread_setschedparam()
  ```

### 3.3 Build Infrastructure

Most build changes are transparent if you use Workbench. This section highlights some exceptions. For more build information, see [2. VxWorks and Development Environment Changes](#).

#### 3.3.1 Recompiling Source Code

If you need to recompile the VxWorks kernel libraries from source code, or if you wish to create additional kernel libraries from your own source code, use the *wrconfig* utility to set up a build environment. *wrconfig* generates a makefile and
subdirectory structure to support the build; the resulting archive (.a) files can later be linked into VxWorks image projects. For more information, see the *VxWorks Command-Line Tools User’s Guide*.

**NOTE:** The VxWorks 5.5 kernel library build model is still supported, in order to maintain backward compatibility.

### 3.3.2 Header File Changes

#### Type Changes

The `pathLib.h` header file now uses `const char *`.

`isascii()`, `toastcii()`

These routines have been moved. They are no longer defined in `vxWorks.h`; instead they now reside in `ctypes.h` to parallel the user-mode library API. If you rebuild an application and see undefined references to these routines, include `ctypes.h`.

#### Private objLib Macro

The `OBJ_VERIFY` macro has moved into the private header file `objLibP.h`, which is located in `installDir/vxworks-6.x/target/h/private`.

### 3.3.3 Compiling for Both VxWorks 6.x and VxWorks 5.5

Because VxWorks 6.x is highly backward compatible, the same code can be compiled for both VxWorks 6.x and VxWorks 5.5. Provided you are aware of the differences, it is straightforward to design code that can be moved between the two environments.

In the short run, the following macros set in `version.h` can assist you in compiling for multiple versions. However, you should be aware that these macros are intended for internal Wind River use. Their definition will change with each release, according to the release level, making any code that uses them potentially obsolete.
For code intended to run in processes, it is more appropriate to use the `uname()` API, which is provided for this purpose.

**Table 3-1 Macros For Specifying the VxWorks Version**

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>_WRS_VXWORKS_MAJOR</td>
<td>6</td>
</tr>
<tr>
<td>_WRS_VXWORKS_MINOR</td>
<td>3(^a)</td>
</tr>
<tr>
<td>_WRS_VXWORKS_MAINT</td>
<td>0</td>
</tr>
<tr>
<td>_WRS_VXWORKS_5_X</td>
<td>N/A</td>
</tr>
</tbody>
</table>

\(^a\) Updated in VxWorks 6.3.

### 3.4 Unsupported Facilities

The following VxWorks 5.5 facilities are not supported in VxWorks 6.6:

- dosFs 1.0
- tapeFs
- rt11Fs
- VxVMI
- VxFusion

In the case of dosFs 1.0, the current version of dosFs should be used. For the other file systems, an alternative file system must be selected.

For VxVMI, VxWorks provides various memory protection features by default, and applications can be migrated to RTP applications (see **3.4.1 VxVMI and Migration**, p.42).

For VxFusion, alternate communication mechanisms may be implemented (for more information, see **3.4.2 VxFusion and Migration**, p.45).
3.4.1 VxVMI and Migration

VxVMI is no longer supported in VxWorks 6.x. It is replaced by real-time process (RTP) support. Table 3-2 compares VxVMI features to those provided in VxWorks 6.x.

<table>
<thead>
<tr>
<th>VxVMI Feature</th>
<th>Supported in VxWorks 6.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>text write protection</td>
<td>Yes</td>
</tr>
<tr>
<td>kernel vector table write protection</td>
<td>Yes</td>
</tr>
<tr>
<td>API to map physical to virtual memory</td>
<td>Yes</td>
</tr>
<tr>
<td>API to modify and examine state of virtual memory</td>
<td>Yes</td>
</tr>
<tr>
<td>API to generate report on state of virtual memory</td>
<td>Yes</td>
</tr>
<tr>
<td>creation of virtual memory contexts</td>
<td>No</td>
</tr>
</tbody>
</table>

In VxWorks 6.x, all of the features shown in Table 3-2, except creation of virtual memory contexts, are available as part of the basic virtual memory library, INCLUDE_MMU_BASIC.

Creation of virtual memory contexts as implemented for VxVMI is no longer available. This functionality is replaced by real-time processes. In VxWorks 6.x, processes execute in private virtual memory contexts.

The behavior of vmBaseStateSet() and vmStateSet() has changed with regard to the cache attributes, projection attributes, and validity attribute, as described below.

**Cache Attributes**

When changing cache attributes, the user must always specify the cacheability and, if supported by the architecture, the guarded and coherency bits together. These are changed using a single mask, MMU_ATTR_CACHE_MSK. The cacheability must be specified with one (and only one) of the following: MMU_ATTR_CACHE_OFF, MMU_ATTR_CACHE_COPYBACK, or MMU_ATTR_CACHE_WRITETHRU.
Protection Attributes

In previous releases, the protection setting of `VM_STATE_WRITABLE` changed both supervisor- and user-mode protection. In VxWorks 6.x, supervisor and user protection attributes are set with distinct macros, that is, `MMU_ATTR_SUP_RWX` and `MMU_ATTR_USR_RWX`.

Validity Attribute

There is no change in behavior for the validity attribute.

In VxWorks 6.x, these states must be changed by first calling `vmStateGet()` to get the current state for the page and then calling `vmStateSet()` or `vmBaseStateSet()` to set the new state.

Table 3-3 illustrates the VxVMI APIs and their VxWorks 6.x replacements, where replacements are available.
The type of the parameter specifying the virtual address defined for the routines `vmMap()`, `vmStateSet()`, `vmBaseStateSet()`, `vmStateGet()`, and `vmTranslate()` is changed from `void *` in VxWorks 5.5 to `VIRT_ADDR` in VxWorks 6.x. `VIRT_ADDR` is declared as an unsigned integer (UINT32). Depending on the level of compiler warnings and error checking selected, as well as the toolchain used (the Wind River Compiler or the Wind River GNU Compiler), this change may generate compiler warnings or errors. You must either change the type of variables used in your application to represent virtual addresses or cast them to `VIRT_ADDR` where required.

---

<table>
<thead>
<tr>
<th>VxVMI</th>
<th>VxWorks 6.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmMap()</td>
<td>vmMap()</td>
</tr>
<tr>
<td>vmTextProtect( )</td>
<td>vmTextProtect( )</td>
</tr>
<tr>
<td>vmStateSet()</td>
<td>vmStateSet()</td>
</tr>
<tr>
<td>vmStateGet()</td>
<td>vmStateGet()</td>
</tr>
<tr>
<td>vmTranslate()</td>
<td>vmTranslate()</td>
</tr>
<tr>
<td>vmPageSizeGet()</td>
<td>vmPageSizeGet()</td>
</tr>
<tr>
<td>vmContextShow()</td>
<td>vmContextShow()</td>
</tr>
<tr>
<td>vmShowInit()</td>
<td>not available</td>
</tr>
<tr>
<td>vmContextCreate()</td>
<td>not available</td>
</tr>
<tr>
<td>vmContextDelete()</td>
<td>not available</td>
</tr>
<tr>
<td>vmCurrentGet()</td>
<td>not available</td>
</tr>
<tr>
<td>vmCurrentSet()</td>
<td>not available</td>
</tr>
<tr>
<td>vmGlobalInfoGet()</td>
<td>not available</td>
</tr>
<tr>
<td>vmGlobalMap()</td>
<td>not available</td>
</tr>
<tr>
<td>vmPageBlockSizeGet()</td>
<td>not available</td>
</tr>
</tbody>
</table>

a. `vmTextProtect()` now takes an argument (BOOL setState)
The type of the parameter specifying the physical address defined for the routines `vmMap()` and `vmTranslate()` is changed respectively from `void *` and `void **` in VxWorks 5.5 to `PHYS_ADDR`, and `PHYS_ADDR *` in VxWorks 6.x. On some architectures (PowerPC and MIPS), `PHYS_ADDR` is defined as an unsigned `long long` (64-bit unsigned integer, `UINT64`), while on the other architectures it is defined as a unsigned integer (32-bit unsigned integer, `UINT32`). Again, depending on the level of compiler warnings and error checking selected, as well as the toolchain used (the Wind River Compiler or the Wind River GNU Compiler), this change may generate compiler warnings or errors. You must either change the type of variables used in your application to represent physical addresses, or cast them to `PHYS_ADDR` where required.

**VxVMI and RTP Applications**

In some situations, it may be appropriate to migrate your VxWorks 5.5 application to a VxWorks 6.x process instead of a VxWorks 6.x kernel application. This is usually the case when the application requires memory protection or if it makes use of the VxVMI library routines which are no longer supported. It may also be desirable when the product contains multiple disparate applications that are largely independent of each other, or if the application relies on POSIX behaviors and interfaces that may not be provided in the kernel space.

Note that processes are built differently from kernel application legacy code. When examining the build support for VxWorks 6.x, you must not confuse the new support included for building processes and shared libraries with that provided for building the kernel and dynamically linked objects. For more information, see the *VxWorks Application Programmer’s Guide*.

**3.4.2 VxFusion and Migration**

For information about VxFusion and migration, see *2.8.9 VxFusion*, p. 21.
3.5 System Changes

Changes have occurred in several areas, including tasks, caching, and memory partition options. For additional changes, see *VxWorks 6.6 Facilities*, p.15.

3.5.1 taskSwitchHookAdd()

The general behavior of this routine has not changed, but a subtle change in the VxWorks 6.x scheduler may affect customer task switch hooks.

In the VxWorks kernel, there is a global variable called taskIdCurrent that typically contains the task ID of the currently executing task (or, in an ISR context, the task ID of the task that was interrupted). In the VxWorks 5.5 scheduler, the value of taskIdCurrent was updated to contain the task ID of the task to be scheduled in *before* invoking the task switch (and swap) hooks. In the VxWorks 6.x scheduler, the value of taskIdCurrent is updated to contain the task ID of the task to be scheduled in *after* invoking the task switch (and swap) hooks.

3.5.2 taskCreat()

The unpublished VxWorks 5.5 routine taskCreat(), defined in taskLibP.h in the installDir/vxworks-6.x/target/h/private directory, is deprecated. The new taskCreate() routine has the same behavior and API as taskCreat() and should be used in its place.

3.5.3 _func_excBaseHook Daisy Chaining

The _func_excBaseHook function pointer is private in VxWorks 5.5 and remains so in VxWorks 6.x. If you continue to use this function pointer, you must follow the daisy chaining policy described in this section.

The _func_excBaseHook is provided so that Wind River components can use the exception mechanism to handle exceptions in their own way. Currently, the only user of this feature is objVerify(). The object management system installs a hook during system initialization; the hook is always present to trap accesses to non-existing or protected memory when an application supplies a bad object identifier.

The functions hooked into _func_excBaseHook must return a non-zero value to indicate that the exception has been handled, which allows excExcHandle() to
return without taking any action. A zero return value indicates that normal exception handling should continue.

If an additional Wind River subsystem wishes to hook into the exception handling path, the _func_excBaseHook can be daisy-chained. When the subsystem initialization function executes, the existing FUNCPTR value of _func_excBaseHook must be cached. Then, during exception handling, the cached FUNCPTR must be called if the exception is not to be handled by the current hook.

**NOTE:** The VxWorks simulator temporarily overwrites the _func_excBaseHook hook (and does not perform daisy chaining) in vxMemProbeArch(). However, the entire sequence of operations in vxMemProbeArch() where the _func_excBaseHook hook has been used in a non-standard manner is protected with intLock()/intUnlock().

### 3.5.4 cacheLib Routines

In previous versions of VxWorks, two non-published routines, cacheArchFlush() and cacheArchInvalidate(), have occasionally been used to manipulate the PowerPC caches. These routines were part of the PowerPC cache library implementation. Occasionally, BSPs and device drivers made direct use of these routines instead of calling the standard library entry points for these operations.

In VxWorks 6.x, cacheArchFlush() and cacheArchInvalidate() have been removed from the VxWorks libraries. Any source file that uses either of these two function calls can instead use the following architecture-independent cache library routines:

<table>
<thead>
<tr>
<th>Obsolete Routines</th>
<th>Replacement Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>cacheArchFlush()</td>
<td>cacheFlush()</td>
</tr>
<tr>
<td>cacheArchInvalidate()</td>
<td>cacheInvalidate()</td>
</tr>
</tbody>
</table>

The cacheFlush() and cacheInvalidate() routines accept the same parameters as the cacheArchFlush() and cacheArchInvalidate() routines they replace.
3.5.5 Private HASH_TBL Structure

The definition of the HASH_TBL structure has been moved into the private header file installDir/vxworks-6.x/target/h/private/hashLibP.h.

3.5.6 vmBaseLib Parameter Change

The VxWorks 6.x version of the vmBaseLibInit() routine takes the parameter cacheDefault. The VxWorks 5.5 version of the routine takes the parameter pageSize.

NOTE: In general, library initialization routines should not be called by user code; they should only be called by the operating system.

3.5.7 Changed Virtual Memory Routines

The vmBaseStateSet() and vmStateSet() routines are not fully backward compatible with the VxWorks 5.5 versions. For more information, see 3.4.1 VxVMI and Migration, p.42.

3.5.8 Memory Partition Options

This section provides a summary of the new and changed memory partition options introduced in VxWorks 6.2. For more information about the memory partition error handling options see the reference entry for the memLib kernel library and the memPartLib application library, as well as the kernel and application versions of memPartOptionsSet() and memOptionsSet(). For more information about the error detection and reporting facility and policy handlers see the VxWorks Kernel Programmer’s Guide: Error Detection and Reporting.

New Options

In VxWorks 6.2 the following new memory partition options have been added to memPartLib.c and memLib.c:

MEM_ALLOC_ERROR_EDR_FATAL_FLAG
   Inject a fatal event when there is an error in allocating memory.

MEM_ALLOC_ERROR_EDR_WARN_FLAG
   Inject a warning when there is an error in allocating memory.
MEM_BLOCK_ERROR_EDR_FATAL_FLAG
Inject a fatal event when there is an error in freeing or reallocating memory.

MEM_BLOCK_ERROR_EDR_WARN_FLAG
Inject a non-fatal event when there is an error in freeing or reallocating memory.

Enabling the error detection and reporting-specific options does not require the infrastructure to be enabled. However, when error detection and reporting is enabled, these flags provide additional debug capability, such as call stack trace information.

Deprecated Options

The following options are deprecated; for alternatives, see Replacement Options, p.49.

MEM_ALLOC_ERROR_SUSPEND_FLAG
Suspend the task when there is an error in allocating memory unless the task was spawned with the VX_UNBREAKABLE option.

MEM_BLOCK_ERROR_SUSPEND_FLAG
Suspend the task when there is an error in freeing or reallocating memory, unless the task was spawned with the VX_UNBREAKABLE option.

Replacement Options

MEMALLOC_ERROR_EDR_FATAL_FLAG
This flag replaces MEM_ALLOC_ERROR_SUSPEND_FLAG. It differs in that it suspends all tasks, including unbreakable ones.

MEM_BLOCK_ERROR_EDR_FATAL_FLAG
This flag replaces MEM_BLOCK_ERROR_SUSPEND_FLAG. It differs in that it suspends all tasks, including unbreakable ones.

For information on modifying the behavior of the fatal error flags, see the VxWorks Kernel Programmer's Guide: Error Detection and Reporting.

Changed Default Options

In the kernel, the default memory partition options have been changed as follows:
The addition of the error detection and reporting warning flags in kernel space does not have backward compatibility consequences.

In future releases the MEM_BLOCK_ERROR_SUSPEND_FLAG flag will be removed from the default options.

Restoring Prior Options

If you prefer to deploy the default memory partition options of previous releases, memOptionsSet() can be used for the heap memory partition. For example:

```c
memOptionsSet(MEM_ALLOC_ERROR_LOG_FLAG | MEM_BLOCK_CHECK | MEM_BLOCK_ERROR_LOG_FLAG | MEM_BLOCK_ERROR_SUSPEND_FLAG);
```

### 3.5.9 Private Structures and Routines

The excLib library documentation has changed. The excTask() routine, which was not intended to be public, is no longer published. The excJobAdd() routine is now provided.
3.5.10 Deprecated Power Management APIs

With the introduction of the new power management facility, the `vxPowerModeSet()` and `vxPowerModeGet()` routines are deprecated. The routines still exist but have no effect on power management. Applications making use of these routines should migrate to APIs provided by the light CPU power manager.

To migrate your kernel applications to the new facility, do the following:

- Replace calls to `vxPowerModeSet(VX_POWER_MODE_DISABLE)` with `cpuPwrMgrEnable(FALSE)`.
- Replace calls to `vxPowerModeSet(VX_POWER_MODE_AUTOHALT)` with `cpuPwrMgrEnable(TRUE)`.
- Replace calls to `vxPowerModeGet()` with `cpuPwrMgrIsEnabled()`.

The `INCLUDE_CPU_LIGHT_PWR_MGR` component is used to include or exclude this module from VxWorks. It is included in the default VxWorks configuration.

3.5.11 Removed APIs

The following unpublished routines are not available in this release:

```c
  semBCoreInit()
  semCCoreInit()
  semMCoreInit()
  semQInit()
```

If you have used these unpublished APIs with a past release, you should modify your code to use `semBInitialize()`, `semCInitialize()`, and `semMInitialize()` routines (or their associated macros) instead.

3.6 I/O System Changes

The I/O system has been updated to support POSIX compliance as well as the new file system architecture.
I/O Error Code Value Changes

In order for the VxWorks I/O system to be more in-line with POSIX I/O system error codes, the numeric values for the old VxWorks I/O-related errors have been changed. The macro names have not been changed; only the numeric values that represent them have changed.

Because VxWorks had a richer set of error codes than POSIX, some of the VxWorks error codes are no longer numerically unique. This could cause application code that tries to decode the numeric values to fail with compiler errors. For example, a switch statement that tries to decode `S_iosLib_NO_DEVICE_FOUND` and `S_ioLib_NO_DEVICE_NAME_IN_PATH` as different cases now generates a compiler error because the two are not unique.

Table 3-5 provides a mapping of VxWorks error codes to their POSIX equivalents. This table contains those error codes that are no longer numerically unique.

Table 3-5  VxWorks I/O Errors with Non-Unique Numeric Error Codes

<table>
<thead>
<tr>
<th>VxWorks Name</th>
<th>POSIX Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_iosLib_NODEVICE_FOUND</td>
<td>ENODEV</td>
</tr>
<tr>
<td>S_ioLib_NODEVICE_NAME_IN_PATH</td>
<td>ENODEV</td>
</tr>
<tr>
<td>S_iosLib_CONTROLLER_NOT_PRESENT</td>
<td>ENXIO</td>
</tr>
<tr>
<td>S_ioLib_DISK_NOT_PRESENT</td>
<td>ENXIO</td>
</tr>
<tr>
<td>S_ioLib_NO_DRIVER</td>
<td>ENXIO</td>
</tr>
<tr>
<td>S_iosLib_DUPLICATEDEVICE_NAME</td>
<td>EINVAL</td>
</tr>
<tr>
<td>S_iosLib_INVALIDETHERNETADDRESS</td>
<td>EINVAL</td>
</tr>
<tr>
<td>S_ioLib_NOFILENAME</td>
<td>EINVAL</td>
</tr>
<tr>
<td>S_ioLib_DEVICE_ERROR</td>
<td>EIO</td>
</tr>
<tr>
<td>S_ioLib_DEVICE_TIMEOUT</td>
<td>EIO</td>
</tr>
<tr>
<td>S_ioLib_UNFORMATTED</td>
<td>EIO</td>
</tr>
</tbody>
</table>

Table 3-6 provides a list of error codes that are still numerically unique.
3 Migrating Kernel Applications

3.7 File System Changes

If an application does not use a custom block device driver or file system, the changes to the changes to the file system should have no impact on your application, other than minor changes to initialization APIs.

VxWorks 6.2 introduces extensive changes to file system support. This section contains information about migrating your kernel applications. For an overview of XBD support, the new highly reliable file system (HRFS), and other changes, see 2.8.12 File System Changes, p. 22. For more information, see the VxWorks Kernel Programmer’s Guide: Local File Systems.

3.7.1 Extended Block Device (XBD) Support

Under the new XBD facility, only file system code should directly access XBDs. If it is necessary to access the underlying device, rawFs is available.

<table>
<thead>
<tr>
<th>VxWorks Name</th>
<th>POSIX Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_iosLib_DRIVER_GLUT</td>
<td>ENOMEM</td>
</tr>
<tr>
<td>S_iosLib_INVALID_FILE_DESCRIPTOR</td>
<td>EBADF</td>
</tr>
<tr>
<td>S_iosLib_TOO_MANY_OPEN_FILES</td>
<td>EMFILE</td>
</tr>
<tr>
<td>S_ioLib_UNKNOWN_REQUEST</td>
<td>ENOSYS</td>
</tr>
<tr>
<td>S_ioLib_WRITE_PROTECTED</td>
<td>EACCES</td>
</tr>
<tr>
<td>S_ioLib_CANCELLED</td>
<td>ECANCELED</td>
</tr>
<tr>
<td>S_ioLib_NAME_TOO_LONG</td>
<td>ENAMETOOLONG</td>
</tr>
<tr>
<td>S_ioLib_CANT_OVERWRITE_DIR</td>
<td>EISDIR</td>
</tr>
</tbody>
</table>

Table 3-6 VxWorks I/O Errors with Unique Numeric Error Codes
The XBD facility resides between the file system and the driver, replacing CBIO. In most cases, migration is straightforward.

- The Wind River device drivers for USB block storage, ATA, and RAM disk devices have been updated to be compliant with the XBD driver interface. The only migration steps are:
  - Include the `INCLUDE_XBD` component in your VxWorks project.
  - Remove any code that directly initializes a file system. For example, a call to `dosFsDevCreate()` must be removed.

- The `BLK_DEV`-based device drivers for floppy drives, SCSI, and TrueFFS (the disk-access emulator for flash) have not been updated to be compliant with the XBD driver interface. They require the XBD wrapper component in order to work with the XBD facility.
  - In addition to `INCLUDE_XBD`, add the `INCLUDE_XBD_BLK_DEV` component in your VxWorks project.
  - Remove any code that directly initializes a file system.

- Custom drivers that were compliant with the `BLK_DEV` interface can be used with XBD by using `INCLUDE_XBD_BLK_DEV`.

- Custom drivers that were not `BLK_DEV`-compliant must be migrated to be either `BLK_DEV`-compliant or XBD-compliant. XBD is the preferred route.

<table>
<thead>
<tr>
<th>XBD-Compliant Drivers</th>
<th>Drivers Requiring XBD Wrapper Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB block storage</td>
<td>Floppy devices</td>
</tr>
<tr>
<td>ATA</td>
<td>SCSI</td>
</tr>
<tr>
<td>RAM disk</td>
<td>TrueFFS</td>
</tr>
</tbody>
</table>

**xbdBlkDev.c**

XBDs replace CBIO and block device drivers. These are soft or logical extended block devices. The `xbdBlkDev.c` library provides the XBD block wrapper for `BLK_DEV` drivers.
3 Migrating Kernel Applications
3.7 File System Changes

xbdBlkDevLibInit()
This routine initializes the XBD block wrapper library.

xbdBlkDevCreate()
This routine creates an XBD block device wrapper on top of a BLK_DEV device.

xbdBlkDevDelete()
This routine destroys a previously created XBD block device wrapper.

cdromFsLib.c

cdromFsDevCreate()
This routine now takes a device_t instead of a BLK_DEV *. However, it is not necessary to call this routine in VxWorks 6.x as the new file system framework calls it automatically when the CD-ROM device is detected.

cdromFsVersionDisplay()
cdromFsVersionNumGet()
These routines are deprecated.

usrFdiskPartLib.c
While still supported, this component and all CBIO-based components are deprecated.

partLib.c
partLib.c is the XBD version of the usrFdisk component. Include it with INCLUDE_XBD_PART_LIB.

xbdCreatePartition()
This routine creates an FDIDK-like partition table on a disk.

xbdRamDisk.c
The following routines are provided by xbdRamDisk.c, the XBD version of the BLK_DEV and CBIO RAM disk components.

xbdRamDiskDevCreate()
This routine creates an XBD RAM disk. It replaces ramDiskDevCreate().

xbdRamDiskDevDelete()
This routine deletes a previously created XBD RAM disk.
Disk Partitioning

The `usrFdiskPartCreate()` routine is no longer used for disk partitioning. It is replaced by the `xbdCreatePartition()` routine.

```c
STATUS xbdCreatePartition
{
    char * pathName,
    int nPart,
    int size1,
    int size2,
    int size3
}
```

This routine creates a partition table using `pathName` to specify the appropriate XBD. The `nPart` parameter indicates the number of partitions to create (up to 4). The `size1`, `size2`, and `size3` parameters indicate the percentage of the disk to use for the first, second and third partitions respectively.

This routine performs the following steps:

- Remove all of the file systems and intermediate XBDs from the XBD stacks based on a single XBD.
- Remove the partition manager.
- Place a partition table on the XBD which actually accesses the media.
- Recreate the XBD stacks by creating a new partition manager based on the newly created table.

Fallback to rawFs

In situations in which a file system cannot be detected on an XBD, or where unformatted access to the media is required (as when formatting a file system or partitioning a disk) rawFs is used as a file system on the XBD. The top of every XBD stack is accessible in core I/O as a pathname; if no other file system exists, then that XBD stack is accessed by rawFs.

3.7.2 Disk Formatting

Disk formatting routines, such as `dosFsVolFormat()` and `hrfsFormat()`, take a pathname argument that specifies what to format. That pathname must refer to an entry in the core I/O device table (accessible by the `devs` command). Once formatting is complete, the path refers to the newly formatted file system. This differs from pre-VxWorks 6.2 versions, when you created the file system by calling `xxxDevCreate()`. Now, once you format the disk, file system creation is automatic.
3.7.3 ioctl() Commands Removed

Several ioctl() commands are no longer supported by file systems.

**FIODISKCHANGE**
This command is no longer supported. XBD-based devices determine their status either automatically or by calling the XBD_TEST ioctl() command. XBD_TEST causes the devices to test for status and insert or remove a new file system as appropriate.

**FIOFORMAT**
This command is no longer supported. There are now multiple, general purpose file systems, which cannot be specified using FIOFORMAT ioctl().

3.7.4 usrFsLib.c

There have been some changes to usrFsLib.c. Most routines have stayed the same, but the following APIs have changed:

**diskFormat()**
This routine is deprecated and prints a warning, but proceeds with formatting the device for dosFs.

**diskInit()**
This routine is deprecated and prints an error when used.

**dosfsDiskFormat()**
This routine is new in VxWorks 6.2; it replaces diskInit() and diskFormat().
3.7.5 dosFS

VxWorks no longer supports dosFs 1.0, which was deprecated in the VxWorks 5.x time frame.

**dosFs 2.0 Migration APIs Removed**

This VxWorks release does not provide backward compatibility with the dosFs 1.0 API. The following migration routines have been replaced with empty stub routines:

- `dosFsInit()`
- `dosFsDevInit()`
- `dosFsDevInitOptionsSet()`
- `dosFsMkOptionsSet()`
- `dosFsConfigInit()`
- `dosFsConfigGet()`
- `dosFsConfigShow()`
- `dosFsModeChange()`
- `dosFsReadyChange()`
- `dosFsVolOptionsGet()`
- `dosFsVolOptionsSet()`
- `dosFsDateTimeInstall()`

**Caches and Cache Tuning**

The cache has been modified so that the FAT, directory entries, and the data each have their own cache. This allows for finely tuning the cache and improved performance. The following parameters to `INCLUDE_DOSFS_CACHE` allow for cache configuration:

- **DOSFS_DEFAULT_FAT_CACHE_SIZE** (default 16 KB)
- **DOSFS_DEFAULT_DATA_CACHE_SIZE** (default 128 KB)
- **DOSFS_DEFAULT_DIR_CACHE_SIZE** (default 64 KB)

These parameters replace **DOSFS_DEFAULT_CACHE_SIZE** (which does exist in this release). The settings for a particular cache can be retrieved using `dosFsCacheInfo()` and `dosFsCacheTune()`.
The `dosFsCacheCreate()` routine now takes additional parameters for all three cache types.

**Other API Changes**

**dosFsLib.c**

`dosFsDevCreate()`  
This routine now takes a `device_t` instead of a `CBIO_DEVI_ID`. However, it is not necessary to call this routine in VxWorks 6.2 as the new file system framework calls it automatically when the CD-ROM device is detected.

**dosFsFmtLib.c**

`dosFsVolFormat()`  
This routine now exclusively takes the name of the device to format:

Previous:  
```
void *device
```

Current:  
```
char *device
```

### 3.7.6 Modified I/O APIs

The following file system routines have changed:

**creat()**  
This routine now accommodates HRFS permission bits. This means that it is unchanged for dosFs, but for HRFS, you change the open mode flags to permission bits. For example:

dosFs:
```
creat("/mydisk/myfile", O_RDWR);
```

HRFS:
```
creat("/mydisk/myfile", 0666);
```

**fcntl()**  
This routine now supports advisory file locking. The following values for the `command` argument are now supported:

**F_GETLK**  
Find out if a lock already exists.

**F_SETLK**  
Set a shared or exclusive lock, or return -1.
F_SETLKW
Set a shared or exclusive lock, waiting until the resource is available if necessary. Return -1 if interrupted before the lock is set.

rmdir()
This routine now removes, and.. entries on HRFS if the directory being removed is open; this conforms to the POSIX standard.

3.8 POSIX Support Changes

The changes in POSIX support in the kernel are discussed in this section.

POSIX Message Queues

Several changes have been made to the mq_des structure.
In both VxWorks 5.5 and VxWorks 6.x, the POSIX message queue (mqPxLib) structure type mqd_t is defined as follows in mqueue.h:

```c
struct mq_des;
typedef struct mq_des * mqd_t;
```

The internals of the mq_des structure have changed between VxWorks 5.5 and VxWorks 6.x. Because this structure is defined in a private header (installDir/vxworks-6.x/target/h/private/mqPxLib.h), the possibility of applications accessing mqd_t internals is much less likely than in the POSIX semaphore situation described below. However, the impact of the change to the mq_des structure is that an mqd_t value is no longer a VxWorks kernel object ID. In concrete terms, performing the following no longer works:

```c
mqd_t mq_id = mq_open ("test", 0x202, 0, 0);
show (mq_id);
```

Instead, the mqPxShow() command must be substituted for show() :

```c
mqd_t mq_id = mq_open ("test", 0x202, 0, 0);
mqPxShow (mq_id);
```
POSIX Thread Support

Several changes have been made to pthreadLib, as follows:

pthread_attr_setstacksize()

This routine now returns the EINVAL status if the stack size is smaller than PTHREAD_STACK_MIN. Previously (in VxWorks 5.5), stack size was not checked, even though this check is required by the POSIX standard.

pthread_create()

This routine now returns the EINVAL status if the a user-supplied stack area is provided but its size is not valid. Previously (in VxWorks 5.5), the stack size, if invalid, was forced to the default stack size. Then, because no stack area of this default size was actually provided by user code, thread creation would (at best) fail with an EAGAIN status. According to the POSIX standard, EINVAL is the status that should be returned when the thread attributes are invalid. The EAGAIN error status should be returned when the system does not have the necessary resources to create a new thread, which is not the case in this situation.

POSIX Semaphores

Several changes have been made to semPxLib.

The sem_t type is defined as follows in the VxWorks 5.5 semaphore.h file:

```c
typedef struct sem_des /* sem_t */
{
    OBJ_CORE objCore;     /* semaphore object core */
    SEM_ID semId;         /* semaphore identifier */
    int refCnt;           /* number of attachments */
    char * sem_name;      /* name of semaphore */
} sem_t;
```

In VxWorks 6.x, the definition of the structure has been made private (private/semPxLibP.h) and the definition of sem_t appears as follows in semaphore.h:

```c
typedef void * sem_t;
```

It is non-standard for POSIX applications to access the internals of the sem_t structure. Any application that accesses the internals of the sem_t structure must be modified to execute in VxWorks 6.x, preferably by eliminating such references.
The following pthread routines have been modified:

**pthread_attr_getdetachstate()**
- This routine now returns the **EINVAL** error code if either the `pAttr` parameter or the `pDetachState` parameter is not valid.

**pthread_attr_getname()**
- This routine now returns the **EINVAL** error code if either the `pAttr` parameter or the `name` parameter is not valid.

**pthread_attr_getopt()**
- This routine now returns the **EINVAL** error code if either the `pAttr` parameter or the `pOptions` parameter is not valid.

**pthread_attr_getschedpolicy()**
- `SCHED_OTHER` is now described as the equivalent of the active native VxWorks scheduling policy.

**pthread_attr_getstackaddr()**
- This routine now returns the **EINVAL** error code if either the `pAttr` parameter or the `ppStackAddr` parameter is not valid.

**pthread_attr_getstacksize()**
- This routine now returns the **EINVAL** error code if either the `pAttr` parameter or the `pStackSize` parameter is not valid.

**pthread_attr_init()**
- This routine now sets the default scheduling policy to be `SCHED_OTHER`, the active VxWorks native scheduling policy, instead of `SCHED_RR`.

**pthread_attr_setdetachstate()**
- This routine now returns the **EINVAL** error code if the `pAttr` parameter is not valid.

**pthread_attr_setname()**
- This routine now returns the **EINVAL** error code if the `pAttr` parameter is not valid.

**pthread_attr_setschedpolicy()**
- `SCHED_OTHER` is now described as the equivalent of the active native VxWorks scheduling policy.
3 Migrating Kernel Applications
3.8 POSIX Support Changes

pthread_attr_setscope()
This routine now returns the error code ENOTSUP, instead of indicating success, for the specific case when the requested contention scope is PTHREAD_SCOPE_PROCESS.

pthread_attr_setopt()
This routine now returns the EINVAL error code if the pAttr parameter is not valid.

pthread_attr_setstackaddr()
This routine now returns the EINVAL error code if the pAttr parameter is not valid.

pthread_attr_setstacksize()
This routine now returns the EINVAL error code if the pAttr parameter is not valid.

pthread_create()
This routine now returns the EPERM error code instead of ENOTTY when the requested scheduling policy is not the current system one. It no longer fails when the requested scheduling policy is SCHED_OTHER because this now defaults to using the active native scheduling policy.

pthread_getschedparam()
The reference entry for this routine has been updated.

pthread_setschedparam()
This routine now returns the EPERM error code, instead of EINVAL, when the scheduling policy is not the same as the active native VxWorks scheduling policy.

POSIX Signal APIs

The following signal routines have been modified:

sigtimedwait()
The API for this routine has changed from:

```c
int sigtimedwait (const sigset_t, struct siginfo, const struct timespec);
```

to:

```c
int sigtimedwait (const sigset_t, siginfo_t, const struct timespec);
```
This change is to conform with POSIX but should be transparent to existing application code.
sigwaitinfo()  
The API for this routine has changed from:

```c
int sigwaitinfo (const sigset_t, struct siginfo);
```

to:

```c
int sigwaitinfo (const sigset_t, siginfo_t);
```

This change is to conform with POSIX but should be transparent to existing application code. This routine may set `errno` to `ESRCH`. For more information, see the reference entry.

I/O System Device Control APIs

The following I/O system device control routines have been modified:

- `iosDevDelete()`  
- `iosDrvRemove()`  

The `iosDevDelete()` and `iosDrvRemove()` APIs now support the device delete callback feature. This is transparent to existing applications. For more information on these routines, see the reference entries and the VxWorks Kernel Programmer’s Guide: I/O System.

POSIX-Related Changes in Libraries and APIs

API changes support the POSIX clock and timer, and POSIX threads.

**POSIX Clock and Timer**

The following routines have been modified:

- `timer_settime()`  

This routine now rounds up time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer to the larger multiple of the resolution.

**POSIX Threads**

The following routines have been modified:

- `pthread_cancel()`  

This routine no longer discards cancellation requests against threads in the `PTHREAD_CANCEL_DISABLE` or `PTHREAD_CANCEL_DEFERRED` states.
3 Migrating Kernel Applications
3.8 POSIX Support Changes

pthread_cond_init( )
This routine no longer checks whether the mutex or condition variable object it initializes is already initialized, and potentially, in use.

pthread_cond_timedwait( )
This routine now correctly handles its timeout value and can return the ETIMEDOUT error.

pthread_cond_wait( )
This routine now checks that its pthread_mutex_t parameter is a valid object and returns EINVAL if it is not.

pthread_exit( )
This routine now sets the exiting thread cancellation type to PTHREAD_CANCEL_DEFERRED and its cancellation state to PTHREAD_CANCEL_DISABLE in order to prevent cancellation loops if a cleanup handler is a cancellation point.

pthread_getspecific( )
This routine now returns NULL when called with a valid key parameter to which no value is associated. (This occurs when no corresponding call to pthread_setspecific() has been made.)

pthread_mutexattr_init( )
This routine now initializes all the fields of its pthread_mutexattr_t argument.

pthread_mutex_init( )
This routine no longer checks whether the mutex or condition variable object it initializes is already initialized, and potentially, in use.

pthread_mutex_lock( )
This routine now correctly handles the PTHREAD_PRIO_PROTECT protocol and no longer risks triggering a priority inversion situation.

pthread_mutex_setprioceiling( )
This routine now checks whether the new priority ceiling value is within the supported range of priorities (that is, 0 to 255).

pthread_mutex_trylock( )
This routine now correctly handles the PTHREAD_PRIO_PROTECT protocol and no longer risks triggering a priority inversion situation.

pthread_mutex_unlock( )
This routine now checks whether the calling thread owns the semaphore before it changes the priority of the thread when the PTHREAD_PRIO_PROTECT protocol applies.
3.9 WindView/System Viewer: wvLib

Note that WindView was renamed System Viewer with VxWorks 6.0.

The wvLib library has been updated for VxWorks 6.x in order to simplify creation of System Viewer logs. It is no longer necessary to create and manage the log header, task name buffer, and ring buffer as separate entities. Instead, a ring buffer is created and added to a System Viewer log; the log is managed as a single item. Examples are provided in the wvLib documentation, and also in the supplied wvOn() and wvOff() routines.

APIs Changed

The following System Viewer routine has changed:

wvUploadStart()

This routine, used to upload System Viewer log data to a host, now takes a pointer to a WV_LOG structure instead of a pointer to a ring buffer.

APIs Removed

The following System Viewer routines have changed:

wvEvtLogInit()

Event logging is now initialized when the System Viewer log is created, as part of wvLogCreate().

wvLogHeaderCreate()

The log header is now created when the System Viewer log is created, as part of wvLogCreate().

wvLogHeaderUpload()

The entire log is now uploaded as a single entity as part of wvUploadStart(), rather than in parts.

wvTaskNamesPreserve()

Task name preservation is now done as part of System Viewer log creation, in wvLogCreate().

wvTaskNamesUpload()

Task names are now uploaded as part of the System Viewer log upload, by wvUploadStart().
4  Migrating BSPs and Drivers

4.1  Introduction  67
4.2  Migrating BSPs  68
4.3  Migrating Drivers  81

4.1  Introduction

The goal of this chapter is to provide an example of how to convert VxWorks 5.5 BSPs and drivers to VxWorks 6.6. The Wind River BSPs shipped with VxWorks 6.x can be used for reference purposes.

This chapter provides information specific to migrating VxWorks 5.5 BSPs and drivers to VxWorks 6.6. You should also consult 2. VxWorks and Development Environment Changes for general information about 5.5-to-6.6 migration, much of which is relevant to BSP and driver migration as well.

For a complete discussion of BSPs, including new features provided with VxWorks 6.x, see the VxWorks BSP Developer’s Guide. For information about drivers, including the 6.x XBD driver interface, see the VxWorks Device Driver Developer’s Guide.

Note that VxWorks 6.x BSPs do not work with VxWorks 5.5, meaning you cannot build and run a VxWorks 6.x BSP in a VxWorks 5.5 environment.
4.2 Migrating BSPs

This section covers the process of migrating a VxWorks 5.5 BSP to VxWorks 6.6, including the step-by-step procedure and additional detail on several topics.

4.2.1 Planning for BSP Migration

Wind River recommends that you study VxWorks 6.6 BSPs to see what changes might be required for your VxWorks 5.5 BSP. A few changes (such as the need to recompile) affect your BSP even if you choose not to use new VxWorks 6.x capabilities. Other changes need only concern you when you incorporate new functionality. Note that the sets of BSPs supported for 5.5 and 6.6 are different.

The following checklist highlights the areas that you may have customized in your BSP; the steps you must take if you have done so are detailed in 4. Migrating BSPs and Drivers.

1. Is your architecture supported?
   See your product release notes for a complete list of supported architectures.

2. Is your host supported, and is your preferred compiler supported on that host?
   See your product release notes for a complete list of supported hosts and compilers.

3. Is your BSP based on VxWorks 5.5?

   ! WARNING: If your BSP is based on a release older than VxWorks 5.5, you must first migrate your BSP to VxWorks 5.5. For instructions on migrating a BSP to VxWorks 5.5, see the Tornado Migration Guide, 2.2.

4. Do you have existing VxWorks 5.5-based code that must be migrated?

5. Does your BSP include or link with any third-party binary libraries or objects?
   Any binary objects provided by customers or third-parties must be recompiled to be compatible with VxWorks 6.x data structures and headers.

6. Does your BSP contain modified versions of any standard files from the installDirTornado/target/config/all directory? If you have modified any of the standard BSP files, then you must migrate those changes to the latest versions of those files.
7. Does the BSP contain modified versions of any standard configlette files from `installDir/Tornado/target/src/config` or `installDir/Tornado/target/config/comps/src`?

If you have modified any of the standard configlette files, then you must migrate those changes to the latest versions of those files.

8. Does the BSP try to patch any architecture code through means other than a published hook or call-out function?

9. Does the BSP have customized make targets or rules?

### 4.2.2 BSP Migration Steps

The following steps explain how to port a BSP so that it is compatible with VxWorks 6.6.

**NOTE:** These steps are cumulative. You must follow the steps in the order shown in order for the later ones to work properly.

**Step 1: Copy your BSP.**

Copy your BSP to the VxWorks 6.x directory structure.

**Step 2: Establish your environment.**

If you are using Workbench, this is automatic. From the command line, start `wrenv` as described in 2.2 Environment Variables and Development Shell, p. 6.

**Step 3: Make local copies and customize non-standard files.**

If your BSP contains a modified version of any of the standard files from the `installDir/Tornado/target/config/all` directory, the corresponding files provided in `installDir/vxworks-6.x/target/config/all` must be patched with your changes. Make a local copy of the common file in your BSP directory before patching it. Use the filename `build` macro in `Makefile` under your BSP.

**WARNING:** This procedure may not work, depending on what you have modified and how you have modified it. Many of the standard files have been extensively modified for VxWorks 6.x, so that it may not be obvious how to reapply the changes. You should expect the effort to modify the files again to be equivalent to the effort required to make the modifications to the previous version of the standard file.
Step 4: Create custom configlettes.

If your BSP contains a modified version of any of the standard configlette files in `installDirTornado/target/src/config` or `installDirTornado/target/config/comps/src`, the modifications must be re-applied to the VxWorks 6.x configlette files in `installDir/vxworks-6.0/target/src/config` or `installDir/vxworks-6.0/target/config/comps/src` and added to your BSP. Do not expect to copy the working VxWorks 5.5 or 5.5.1 files; Wind River has made extensive changes in many configlettes. Instead, use the following procedure:

1. Copy the files from `target/src/config` to your BSP directory and link them using `MACH_EXTRA`. For example:

   ```
   MACH_EXTRA = usrNetwork.o
   ```

2. Copy the modified file from `target/comps/src` to your BSP directory and rename the file.

3. Make modifications to `bsp.cdf` in your BSP directory to include your file for a component name. Decide on a component name for your changed file.

4. Merge changes in the modified VxWorks 5.5 or 5.5.1 configlettes with the Wind River changes that were applied to VxWorks 6.x. If you choose a merge method that requires a common ancestor, the as-shipped files from 5.5 or 5.5.1 (as applicable) can be used for this purpose.

   **NOTE:** Depending on the nature of your changes, some amount of redesign may be needed. In some cases, it may be easier to reapply your changes to the VxWorks 6.x files.

Step 5: Copy and link any custom files.

If your BSP patches any architecture code through means other than a published hook or call-out function, copy the custom architecture file to the BSP and link it using `MACH_EXTRA`.

Step 6: Optionally Update your cache library APIs.

Optionally, update your BSP to use the architecture-independent cache library API; replace routines named `cacheArch*` or `cachePpc*` with their associated architecture-independent cache routines.
Step 7: Add support for the new shared-memory networking driver (if used).

The old shared-memory driver, if_sm, is no longer supported. If your BSP will support shared memory, make the following changes to support the new smEnd driver.

- Change config.h.

  If INCLUDE_SM_NET is defined in the original BSP, conditionally include the INCLUDE_SMEND component. Remove the INCLUDE_BSD component if it is defined.

- Change configNet.h.

  Add the shared memory entry in the endDevTbl[] before the last NULL entry.

  For example:

  ```c
  #ifdef INCLUDE_SMEND
  #   define SMEND_LOAD_STRING    ""
  #   define SMEND_LOAD_FUNC      sysSmEndLoad
  IMPORT END_OBJ* SMEND_LOAD_FUNC (char*, void*);
  #endif /* INCLUDE_SMEND */
  
  #ifdef INCLUDE_SMEND
  { 0, SMEND_LOAD_FUNC, SMEND_LOAD_STRING, 0, NULL, FALSE},
  #endif
  ```

- Create a new routine, sysSmEndLoad().

  Although this routine can be placed in any appropriate file (such as sysNet.c), some BSPs include a placeholder for sysSmEndLoad(). (This is the case, for example, with mv5100/sysSmEnd.c.)
The routine `sysSmEndLoad()` converts all shared memory configuration parameters to a load string. The load string format is as follows:

"unit:pAnchor:smAddr:memSize:tasType:maxCpus:masterCpu:localCpu:
maxPktBytes:maxInputPkt:intType:intArg1:intArg2:intArg3:mbNum:
mbNum:configFlg:pBootParams"

- If you create a new file for the routine described in the previous step, modify `sysLib.c` or `Makefile` to include this file.

In `sysLib.c`:

```c
#ifdef INCLUDE_SMEND
    #include "sysSmEnd.c"
#endif /* INCLUDE_SMEND */
```

In `Makefile`:

```
MACH_EXTRA    = sysSmEnd.o
```

**NOTE:** `usrNetwork.c` and `bootConfig.c` have been modified to support the shared memory END driver.

**Step 8:** Update any custom make rules.

Make sure any custom rules are in `Makefile` under the BSP, and that they are documented in the `target.ref` file. This requires case-by-case evaluation. For more information, see 2.4.1 Makefile and make Changes, p.8.

**Step 9:** Build your BSP.

The level of compiler warning and error checking in Wind River's BSP makefiles has increased. This results in higher quality BSPs. However, some BSPs that compiled silently using the VxWorks 5.5 and 5.5.1 makefiles require modification to compile without errors or warnings in VxWorks 6.x.

In order to avoid compiler warnings and errors for your BSP, you can increase the level of checking in your BSP makefiles and consider rewriting your code if necessary (Wind River recommends this approach) or you may choose to reduce the level of checking. For information on how to do this, see 4.2.5 Addressing Compiler Errors and Warnings, p.76 and the documentation for your compiler.

**Step 10:** Change the version number.

In `config.h`, update the BSP version to correspond to the operating system version, and also increment the BSP revision. For example:

```c
#define BSP_VERSION   "2.0"
#define BSP_REV       "/4"  <- increased by 1
```
The **BSP_VERSION** definition should be **2.0** for all VxWorks 6.x BSPs.

In the **README**, add release notes for the VxWorks 6.x version. An example of a Wind River BSP entry is:

```
RELEASE 2.0/4
Released by Wind River for VxWorks 6.6.
```

**Step 11: Migrate the Ethernet MAC address boot loader command.**

In providing boot loader support that works easily, it helps if a BSP is modified to use the **M** interface to MAC addresses. There are a number of problems with the existing **N** command. For this reason, you may wish to add **M** command support, while keeping the **N** command for backward compatibility. For more information and directions, see 4.2.6 Implementing the M Command, p.77.

**Step 12: Convert your BSP documentation.**

Many BSP developers want to provide documentation similar to that provided for Wind River BSPs. For this reason, the tools that Wind River uses internally to generate BSP documentation are provided with this release.

The following is a summary of the required steps:

1. Convert `target.nr` to the current format.
2. Convert other BSP documentation.
3. Test the BSP documentation build.
4. Update the infrastructure files.

For detailed instructions, see 4.2.7 Converting BSP Documentation, p.79.

**Step 13: Compile your BSP and documentation.**

Build both your VxWorks image and your BSP documentation. See 2.5 Compilers, p.9 and your product getting started.
4.2.3 PowerPC BSPs and SDA

The VxWorks kernel initialization process now initializes PowerPC SDA/SDA2 base registers. If a custom BSP invokes a C function from within _sysInit()—that is, before the invocation of _usrInit()—the SDA/SDA2 base registers need to be initialized prior to calling the C function. They should be initialized as follows:

\begin{verbatim}
lis  r2, HI(_SDA2_BASE_)
ori  r2, r2, LO(_SDA2_BASE_)

lis  r13, HI(_SDA_BASE_)
ori  r13, r13, LO(_SDA_BASE_)
\end{verbatim}

4.2.4 Replacing VM_STATE_xxx Macros

The VM_STATE_xxx macros may be deprecated in a future release. You may, therefore, want to replace the VM_STATE_xxx macros in sysPhysMemDesc[] (located in sysLib.c) with the MMU_ATTR_xxx macros as shown in Table 4-2.

In VxWorks 6.x, the VM_STATE_xxx macros are mapped to the appropriate MMU_ATTR_xxx macros. However, for definitions of VM_STATE_xxx and MMU_ATTR_xxx, see the vmLib.h and vmLibCommon.h header files.

<table>
<thead>
<tr>
<th>VxWorks 5.5 Macros</th>
<th>VxWorks 6.x Macros</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM_STATE_MASK_VALID</td>
<td>MMU_ATTR_VALID_MSK</td>
</tr>
<tr>
<td>VM_STATE_MASK_WRITABLE</td>
<td>MMU_ATTR_PROT_MSK</td>
</tr>
<tr>
<td>VM_STATE_MASK_CACHEABLE</td>
<td>MMU_ATTR_CACHE_MSK</td>
</tr>
<tr>
<td>VM_STATE_MASK_MEM_COHERENCY</td>
<td>MMU_ATTR_CACHE_MSK</td>
</tr>
<tr>
<td>VM_STATE_MASK_GUARDED</td>
<td>MMU_ATTR_CACHE_MSK</td>
</tr>
<tr>
<td>VM_STATE_VALID</td>
<td>MMU_ATTR_VALID</td>
</tr>
<tr>
<td>VM_STATE_VALID_NOT</td>
<td>MMU_ATTR_VALID_NOT</td>
</tr>
<tr>
<td>VM_STATE_WRITABLE</td>
<td>MMU_ATTR_SUP_RWX</td>
</tr>
</tbody>
</table>
| VM_STATE_WRITABLE_NOT       | (MMU_ATTR_PROT_SUP_READ | MMU_ATTR_PROT_SUP_EXE)
For those architectures where `PHYS_ADDR` is defined as a 64-bit type, such as PowerPC and MIPS, physical addresses in `sysPhysMemDesc[]` should be changed from 32-bit values to 64-bit values in `sysLib.c`. This prevents type mismatch warnings when compiling. You may need to change explicit castings in your BSP if they conflict with the new definitions of `PHYS_ADDR` or `VIRT_ADDR`.

- **Pentium BSPs Only**: In VxWorks 5.5, Pentium MMU support is implemented in a slightly different manner than other architectures with regard to cache states. This is no longer the case in VxWorks 6.x; therefore Pentium BSPs must change the cache state definitions as shown in Table 4-3.

### Table 4-2 Old and New Memory Protection Macros (cont’d)

<table>
<thead>
<tr>
<th>VxWorks 5.5 Macros</th>
<th>VxWorks 6.x Macros</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VM_STATE_CACHEABLE</code></td>
<td><code>MMU_ATTR_CACHE_DEFAULT</code></td>
</tr>
<tr>
<td><code>VM_STATE_CACHEABLE_NOT</code></td>
<td><code>MMU_ATTR_CACHE_OFF</code></td>
</tr>
<tr>
<td><code>VM_STATE_MEM_COHERENCY</code></td>
<td><code>MMU_ATTR_CACHE_COHERENCY</code></td>
</tr>
<tr>
<td><code>VM_STATE_MEM_COHERENCY_NOT</code></td>
<td>0 (set this macro to 0 for VxWorks 6.x)</td>
</tr>
<tr>
<td><code>VM_STATE_GUARDED</code></td>
<td><code>MMU_ATTR_CACHE_GUARDED</code></td>
</tr>
<tr>
<td><code>VM_STATE_GUARDED_NOT</code></td>
<td>0 (set this macro to 0 for VxWorks 6.x)</td>
</tr>
</tbody>
</table>

### Table 4-3 New Cache State Definitions for Pentium BSPs

<table>
<thead>
<tr>
<th>VxWorks 5.5 Definition</th>
<th>Required VxWorks 6.x Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VM_STATE_MASK_CACHEABLE</code></td>
<td><code>VM_STATE_MASK_CACHEABLE</code> or <code>MMU_ATTR_CACHE_MSK</code></td>
</tr>
<tr>
<td><code>VM_STATE_MASK_WBACK</code></td>
<td><code>VM_STATE_MASK_WBACK</code> or <code>MMU_ATTR_CACHE_MSK</code></td>
</tr>
<tr>
<td><code>VM_STATE.Cacheable</code></td>
<td><code>VM_STATE_MEM.COHERENCY</code> or <code>MMU_ATTR_CACHE_OFF</code></td>
</tr>
<tr>
<td><code>VM_STATE_MEM.COHERENCY</code></td>
<td><code>VM_STATE_MEM.COHERENCY</code> or <code>MMU_ATTR_CACHE_OFF</code></td>
</tr>
<tr>
<td><code>VM_STATE_MEM.COHERENCY</code></td>
<td><code>VM_STATE_MEM.COHERENCY</code> or <code>MMU_ATTR_CACHE_OFF</code></td>
</tr>
<tr>
<td><code>VM_STATE_GUARDED</code></td>
<td><code>VM_STATE_GUARDED</code> or <code>MMU_ATTR_CACHE_COPYBACK</code></td>
</tr>
<tr>
<td><code>VM_STATE_GUARDED</code></td>
<td><code>VM_STATE_GUARDED</code> or <code>MMU_ATTR_CACHE_COPYBACK</code></td>
</tr>
</tbody>
</table>
Failure to make these changes results in an error similar to the following:

```
invalid combination of MMU attributes for sysPhysMemDesc[] entry
```

Example 4-1  Replacing Cache States in Pentium BSPs

The following example illustrates how to replace the cache states in Pentium BSPs while maintaining backward compatibility. In `config.h`:

```
#ifdef MMU_ATTR_SUP_RO /* VxWorks 6.x settings */
# define VM_STATE_MASK_FOR_ALL \ 
  VM_STATE_VALID | VM_STATE_WRITABLE | \ 
  VM_STATE_CACHEABLE | VM_STATE_GLOBAL
# define VM_STATE_FOR_IO \ 
  VM_STATE_VALID | VM_STATE_WRITABLE | \ 
  VM_STATE_CACHEABLE_NOT | VM_STATE_GLOBAL_NOT
# define VM_STATE_FOR_MEM_OS \ 
  VM_STATE_VALID | VM_STATE_WRITABLE | \ 
  VM_STATE_WBACK | VM_STATE_GLOBAL_NOT
# define VM_STATE_FOR_MEM_APPLICATION \ 
  VM_STATE_VALID | VM_STATE_WRITABLE | \ 
  VM_STATE_WBACK | VM_STATE_GLOBAL_NOT
# define VM_STATE_FOR_PCI \ 
  VM_STATE_VALID | VM_STATE_WRITABLE | \ 
  VM_STATE_CACHEABLE_NOT | VM_STATE_WBACK_NOT | 
  VM_STATE_GLOBAL_NOT
#else /* VxWorks 5.x settings */
# define VM_STATE_MASK_FOR_ALL \ 
  VM_STATE_VALID | VM_STATE_WRITABLE | \ 
  VM_STATE_CACHEABLE | VM_STATE_WBACK | VM_STATE_GLOBAL
# define VM_STATE_FOR_IO \ 
  VM_STATE_VALID | VM_STATE_WRITABLE | \ 
  VM_STATE_CACHEABLE_NOT | VM_STATE_WBACK_NOT | 
  VM_STATE_GLOBAL_NOT
# define VM_STATE_FOR_MEM_OS \ 
  VM_STATE_VALID | VM_STATE_WRITABLE | \ 
  VM_STATE_CACHEABLE | VM_STATE_WBACK | VM_STATE_GLOBAL_NOT
# define VM_STATE_FOR_MEM_APPLICATION \ 
  VM_STATE_VALID | VM_STATE_WRITABLE | \ 
  VM_STATE_CACHEABLE | VM_STATE_WBACK | VM_STATE_GLOBAL_NOT
# define VM_STATE_FOR_PCI \ 
  VM_STATE_VALID | VM_STATE_WRITABLE | \ 
  VM_STATE_CACHEABLE_NOT | VM_STATE_WBACK_NOT | 
  VM_STATE_GLOBAL_NOT
#endif /* MMU_ATTR_SUP_RO */
```

4.2.5  Addressing Compiler Errors and Warnings

This section summarizes warning levels and some common reported problems. For more information, see *Wind River Compiler Warnings*, p.10 and your compiler documentation.
Compiler Error and Warning Levels

In addition to the obsolete (but still present) CC_WARNINGS_NONE and CC_WARNINGS_ALL, VxWorks 6.x introduces three new warning levels. The new warning level settings are:

- CC_WARNINGS_LOW
- CC_WARNINGS_MED
- CC_WARNINGS_HIGH

For library builds, the default setting is now CC_WARNINGS_LOW. For BSP and application builds, the default setting is now CC_WARNINGS_MED. The CC_WARNINGS_HIGH level is provided for the purpose of detailed code inspections. The warning level setting can be changed from the command-line make as follows (bash shell on Solaris):

```bash
% make CPU=PPC32 TOOL=diab CC_WARNINGS=$(CC_WARNINGS_HIGH)
```

Example Problems and Solutions

The following problems are commonly encountered due to the increased compiler warning level:

- signed/unsigned value mismatch
  A signed constant or variable may be assigned to an unsigned variable.
  Re-defining the variable’s value (for example, -1 to 0xFFFFFFFF) or using a more suitable typedef may fix this problem.

- condition always true or always false
  The use of a constant as the expression in a condition such as while (TRUE) can be replaced with for (;

4.2.6 Implementing the M Command

This section provides background and details on the process of providing M interface support for your BSP.

Background

There are a number of problems with the N command. These relate to multiple interfaces and to a lack of definition of the byte order of the data. The original N
command did not provide a mechanism for the user to set the MAC address of any interface except the one designated at BSP creation time.

Although some workarounds exist in some BSPs, they are not consistently applied, well documented, or obvious to use. In addition, the design of the N command does not define the order of bytes within the MAC address for various functions. This means that on some architectures and configurations, the MAC address is reversed from the intended order, or other odd behavior results.

The design of the M command defines the byte order for the MAC address at every level in order to eliminate confusion of the MAC address at BSP development time. It also requires the use of a mechanism to allow multiple interfaces to be handled without special consideration.

Keep code supporting the N command for backward compatibility, unless code size or other considerations forbid it. However, where problems occur, the new M command is available.

Implementing the M Command

To provide the option of the M command for a BSP, do the following:

1. From the console, identify whether or not the BSP supports the N command.
   a. Check to see if ETHERNET_ADR_SET is defined in config.h in the BSP directory. If so, support for M command must be added.
   b. Keep code supporting the N command for backward compatibility.

2. Add M support. The following is a synopsis of the steps required:
   a. Replace ETHERNET_ADR_SET with ETHERNET_MAC_HANDLER in config.h.
   b. Replace sysEnetAddrGet() with sysNetMacNVRamAddrGet() in sys{IF}End.c.
   c. Modify sysNet.c or sysLib.c to add the routines sysNetMacNVRamAddrGet(), sysNetMacAddrGet(), and sysNetMacAddrSet() to sysNet.c or sysLib.c, depending on which file currently contains sysEnetAddrGet().

For examples of these changes, see the appropriate files in the wrSbc8260Atm and wrSbcPowerQuiccII BSPs.
4.2.7 Converting BSP Documentation

For VxWorks 6.x, the tool that generates reference documentation is a Perl script, `apigen`. `apigen` uses a simpler syntax than the previous `nroff`-based markup (found in `target.nr` files).

The `apigen` and other documentation tools are located in `installDir/vxworks-6.x/host/resource/doctools`. For syntax information see 5Converting to apigen, p.83.

Convert target.nr

Start by editing the target-specific documentation file to generate clean output using `apigen`.

1. If you have a `target.nr` file, convert it to `target.ref` using the `mg2ref` tool.
   
   ```
   mg2ref target.nr
   ```
   
   If you already have a `target.ref` file, proceed to the next step.

2. Edit the `target.ref` file to correct any markup errors. This effort can vary significantly, depending on the compliance of the original file to the Wind River Coding Conventions. Your goal is to eliminate warning and error messages generated by `apigen`.
   
   a. Change the name in the first line to `target.ref`.
   
   b. Align all table columns and diagrammatic representations if they are not already aligned. Blank characters at the beginning of a line cause the line to be unformatted or document generation to fail.
   
   c. Eliminate all remaining `nroff` formatting; for a table of correspondences, see 5.Converting to apigen. In particular, remove any remaining `nroff` formatting at the beginning of tables; it is no longer required or allowed.
   
   d. Correct grammar and spelling.
   
   e. Edit content for completeness.

3. Run `apigen` manually to create the `bsp.html` file. Edit the `target.ref` until the `bsp.html` is acceptable.
   
   ```
   apigen target.ref
   ```
Convert Other BSP Documentation

Convert other BSP API documentation, such as `sysLib.c`.
1. Follow the same procedure on your remaining files as you did for `target.nr`.
2. Test your conversion by running `apigen` manually.

```%
apigen -missingok sysLib.c
```

The `-missingok` flag allows you to convert your document without warnings for failure to comply with the Wind River coding conventions. For example, failure to include an ERRNO section in an unpublished routine does not generate a warning when you use `-missingok`.

Test the BSP Documentation Build

Test the BSP documentation build.
1. Execute `make man` for your BSP.

```%
make man
```
2. Check the output in the `$DOCS_ROOT/com.windriver.ide.doc.bsp/bspName` directory.
   a. To correct markup and formatting errors that generate error messages in the build logs, be sure the files conform to the coding conventions described in *Wind River Coding Conventions*, available from Online Support.
   b. If the `sysLib.html` has `unknown()` routines listed, it is possible that files with third-party copyright information are not tagged correctly. To fix this problem:

```
/*******
Third-party copyright and license blurb...
\NOMANUAL <<ADD THIS
*******
```

Update the Infrastructure Files

The `bspinstall` script dynamically updates the online help table of contents with new BSPs. The script is located in `installDir/setup`. In Wind River products, this
runs as a post-installation step. In order for your BSP to appear in the Workbench table of contents, you need to run it manually or package it into your own installer.

4.3 Migrating Drivers

VxWorks 5.5 drivers are not binary compatible with VxWorks 6.x drivers. You must install your driver source and recompile the drivers. For general information about 5.5-to-6.6 migration—some of which is relevant to driver migration as well—see 2. VxWorks and Development Environment Changes.

As part of migrating your drivers from 5.5 to 6.6, Wind River recommends migrating to the VxBus driver framework. For information in this regard, see the VxWorks Device Driver Developer’s Guide.

If you choose not to migrate to the VxBus framework, note that VxWorks must be configured with the INCLUDE_XBD_BLK_DEV component for any drivers that were designed to work with the 5.5 CBIO interface. The CBIO interface is superseded by the XBD (extended block device) facility. For more information in this regard, see the VxWorks Kernel Programmer’s Guide: I/O System.
Summary of apigen Markup

Table 5-1 is a summary of apigen markup. It also shows the corresponding mangen markup for those more familiar with that style. For details, see the reference entry for apigen and the conversion steps provided in 4.2.7 Converting BSP Documentation, p.79.

<table>
<thead>
<tr>
<th>Column</th>
<th>apigen Markup</th>
<th>Mangen Equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>'text ...' or 'text ...'</td>
<td>same</td>
<td>boldface words</td>
</tr>
<tr>
<td>any</td>
<td>&lt;text&gt;</td>
<td>same</td>
<td>italicized words (text variables, emphasis)</td>
</tr>
<tr>
<td>any</td>
<td>\ or /</td>
<td>\e</td>
<td>the character \</td>
</tr>
<tr>
<td>any</td>
<td>&lt; &gt; \ ' \ '</td>
<td>N/A</td>
<td>the characters &lt; &gt; \ '</td>
</tr>
<tr>
<td>any</td>
<td>|</td>
<td>N/A</td>
<td>the character</td>
</tr>
<tr>
<td>1</td>
<td>\ss</td>
<td>.tS</td>
<td>preformatted text (syntax)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>\se</td>
<td>.tE</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>\cs</td>
<td>.CS</td>
<td>literal text (code)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>\ce</td>
<td>.CE</td>
<td></td>
</tr>
</tbody>
</table>
Table 5-1  apigen Markup (cont’d)

<table>
<thead>
<tr>
<th>Column</th>
<th>apigen Markup</th>
<th>Mangen Equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\bs</td>
<td>.bS</td>
<td>literal text, smaller (board layout)</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>\be</td>
<td>.BE</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>\is</td>
<td>.IP or .IP</td>
<td>itemized list (terms list)</td>
</tr>
<tr>
<td></td>
<td>\i item</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>\ie</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>\ms</td>
<td>.IP or .IP</td>
<td>marker list (numbered or dashed)</td>
</tr>
<tr>
<td></td>
<td>\m mark</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>\me</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>\ts</td>
<td>.TS</td>
<td>table</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>\te</td>
<td>.TE</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>\sh text</td>
<td>.SS text</td>
<td>subheading</td>
</tr>
<tr>
<td>1</td>
<td>\tb reference</td>
<td>.I, .pG, .tG</td>
<td>cross-reference to a publication</td>
</tr>
</tbody>
</table>

**NOTE:** In Table 5-1, “any” denotes inline markup (it can appear anywhere in text); “1” denotes a tag that must start in effective column 1.
Symbols

_func_excBaseHook  46

dosFsDevCreate( )  59
dosFsVolFormat( )  59
fcntl( )  59
ioctl( )  57
iosDevDelete( )  64
iosDevRemove( )  64
pthread_attr_getdetachstate( )  62
pthread_attr_getname( )  62
pthread_attr_getopt( )  62
pthread_attr_getschedpolicy( )  62
pthread_attr_getstackaddr( )  62
pthread_attr_getstacksize( )  62, 63
pthread_attr_init( )  62
pthread_attr_setdetachstate( )  62
pthread_attr_setname( )  62
pthread_attr_setopt( )  63
pthread_attr_setschedpolicy( )  62
pthread_attr_setscope( )  63
pthread_attr_setstackaddr( )  63
pthread_attr_setstacksize( )  62, 63
pthread_attr_setname( )  62
pthread_attr_setsockopt( )  63
pthread_attr_setschedparam( )  62
pthread_attr_setschedparam( )  63
pthread_atexit( )  61
pthread_create( ), VxWorks 6.0  61
pthread_create( ), VxWorks 6.2  63
pthread_exit( )  65
pthread_getspecific( )  65
pthread_mutex_init( )  65
pthread_mutex_lock( )  65

A

activeQhead  20
apigen
BSP conversion  79
markup summary  83

APIs added

kernel

  cacheFlush( )  47
  cacheInvalidate( )  47
  cdromFsDevCreate( )  55
  cdromFsVersionDisplay( )  55
  cdromFsVersionNumGet( )  55
  dosfsDiskFormat( )  57
  partLib.c  55
  xbdBlkDevLibCreate( )  55
  xbdBlkDevLibDelete( )  55
  xbdBlkDevLibInit( )  55
  xbdCreatePartition( )  55
  xbdRamDiskDevCreate( )  55
  xbdRamDiskDevDelete( )  55

APIs changed

kernel

  creat( )  59
VxWorks
5.5 Migration Guide, 6.6

backward compatibility
kernel mode 37
vmBaseLib 48
boot loaders
backward compatibility 16
VxWorks 5.5 16
BSPs
apigen 79
compatibility checklist 68
compatibility with VxWorks 6.x 6
compiler warnings 72
custom 8
porting
cache library APIs 70
custom configlettes 70
documentation 79
Ethernet MAC address 77
ETHERNET_ADR_SET 78
ETHERNET_MAC_HANDLER 78
introduction 67
version number 72
VM_STATE_xxx macros 74
source compatible 6
building
build specification, glossary definition 3
infrastructure changes 8
setting flags 11
user applications 45
wrconfig utility 39

C
C and C++ libraries, Dinkum 13
cache attributes
MMU_ATTR_CACHE_MSK 42
MMU_ATTR_CACHE_OFF 42
MMU_ATTR_CACHE_WRITE_THRU 42
MMU_ATTR_SUP_RWX 43
MMU_ATTR_USR_RWX 43
MMU_ATTR_xxx 74
cache state definitions 75
cacheArchFlush()  47
cacheArchInvalidate()  47
cacheFlush()  47
cacheInvalidate()  47
cacheLib  47
cdromFsDevCreate()  55
cdromFsVersionDisplay()  55
cdromFsVersionNumGet()  55
changes
build
  kernel  39
file system
  kernel  53
I/O system
  kernel  51
POSIX
  kernel  60
system support
  kernel  46
VxWorks components
  kernel  41
command line environment setup  6
compiler
  default changed  9
  extra command-line flags  11
  new warning and error levels
    GNU, VxWorks 6.x  9
    Wind River Compiler  9
options, GNU  11
path, specifying  7
stricter syntax  9
warnings
  building BSPs  72
  levels  77
compiling
  VxWorks 5.5  40
  VxWorks 6.x  40
components
glossary definition  2
INCLUDE_BSD  71
INCLUDE_EXC_TASK  19
INCLUDE_JOB_TASK  20
INCLUDE_MEM_EDR  19
INCLUDE_MMU_BASIC  42
INCLUDE_SM_NET  71
INCLUDE_SMEND  71
config.h
  changing version number, BSPs  72
  replaced for projects  14
creat()
  kernel  59
create hooks
  kernel task  22
  process, post-create  22
D
definitions
  build specification  3
  component  2
  deprecated  3
downloadable kernel module project  3
error detection and reporting  2
  graphical user interface  3
  project  2
  real-time process  2
  real-time process project  3
  shared-library project  3
  toolchain  3
  VxWorks image project  2
  workspace  2
deprecated, glossary definition  3
Dinkum C and C++ libraries  13
Dinkum standard template library  13
directory structure, changed  7
diskFormat()
  kernel  57
diskInit()
  kernel  57
dosFs 1.0  23
dosFsDevCreate()  59
dosFsDiskFormat()  57
dosFsVolFormat()
  device name  59
  pathname argument  56
downloadable kernel module project  3
driver
  source-compatible with BSPs  81
E

ED&R, see error detection and reporting environment setup, command line 6
ernos
   I/O
duplicate values 52
unique values 52
error detection and reporting glossary definition 2
error-checking, semaphores 22
Ethernet MAC address 77
ETHERNET_ADR_SET 78
ETHERNET_MAC_HANDLER 78
excJobAdd( ) 50
excTask( ) 50
extended block device 22

F

fcntl( )
   kernel 59
file system
   removability support 23
FIODISKCHANGE 57
FIOFORMAT 57
flags
   build, setting 11
GNU
   -fno-exception 14
   -fno-implicit-templates 11
   -fno-rtti 14
   -fvec 11
   -fvolatile 11
   -maltivec 11
   -nostdlib 11

G

graphical user interface, glossary definition 3
GUI, see graphical user interface
Index

linking custom architecture code 70
makefiles, BSP 8
MEMALLOCERROR_EDR_FATAL_FLAG
kernel 48
MEMALLOCERROR_EDR_WARN_FLAG
kernel 48
MEMALLOCERROR_SUSPEND_FLAG
kernel 49
MEMBLOCKERROR_EDR_FATAL_FLAG
kernel 49
MEMBLOCKERROR_EDR_WARN_FLAG
kernel 49
MEMBLOCKERROR_SUSPEND_FLAG
kernel 49
memory partition options
kernel 48
memory protection macros 74
migrating
BSPs
   Ethernet MAC address 77
   version number 72
migration
architecture-specific issues 33
BSPs
   cache library APIs 70
   custom configlettes 70
documentation 79
   sysEnetAddrGet( ) 78
   sysNetMacNVRamAddrGet( ) 78
   VM_STATE_xxx macros 74
checklist, kernel
   VxWorks 5.5 to VxWorks 6.x 38
   custom BSP makefiles 8
MMU_ATTR_CACHE_MSK 42
MMU_ATTR_CACHE_OFF 42
MMU_ATTR_CACHE_WRITETHRU 42
MMU_ATTR_SUP_RWX 43
MMU_ATTR_USR_RWX 43
MMU_ATTR_xxx 74
mq_attr 24
mq_des 60

N
new
   configuration process 14

O
OBJ_VERIFY( ) 40
objcCopy 12
objLib 40
optimization
   compiler 12

P
parameter type change
   physical address 45
   virtual address 44
partLib.c 55
pathLib.h 40
Pentium BSPs, new cache state definitions 75
PHYS_ADDR
   parameter 75
   type changed 45
porting a BSP
   introduction 67
   VxWorks 6.x, to 68
POSIX
   I/O, kernel
      iosDevDelete( ) 64
      iosDevRemove( ) 64
   message queues, kernel
      mq_des 60
   semaphores, kernel
      sem_t moved 61
   signals, kernel
      sigtimedwait( ) 63
      sigwaitinfo( ) 64
   threads, kernel
      pthread_attr_getdetachstate( ) 62
      pthread_attr_getname( ) 62
      pthread_attr_getopt( ) 62
VxWorks
5.5 Migration Guide, 6.6

pthread_attr_getschedpolicy( ) 62
pthread_attr_getstackaddr( ) 62
pthread_attr_getstacksize( ) 62
pthread_attr_init( ) 62
pthread_attr_setdetachstate( ) 62
pthread_attr_setname( ) 62
pthread_attr_setopt( ) 63
pthread_attr_setschedpolicy( ) 62
pthread_attr_setscope( ) 63
pthread_attr_setstackaddr( ) 63
pthread_attr_setstacksize( ) 63
pthread_attr_setstackaddr( )
kernel 62
 pthread_attr_getstacksize( )
kernel 62
 pthread_attr_init( )
kernel 62
 pthread_attr_setdetachstate( )
kernel 62
 pthread_attr_setname( )
kernel 62
 pthread_attr_setopt( )
kernel 62
 pthread_attr_setschedpolicy( )
kernel 62
 pthread_attr_setscope( )
kernel 62
 pthread_attr_setstackaddr( )
kernel 62
 pthread_attr_setstacksize( )
kernel 62
 pthread_cancel( )
kernel 64
 pthread_cond_init( )
kernel 65
 pthread_cond_timedwait( )
kernel 65
 pthread_cond_wait( )
kernel 65
 pthread_create( ), VxWorks 6.0 61
 pthread_create( ), VxWorks 6.2 63
 pthread_exit( )
kernel 65
 pthread_getschedparam( ) 63
 pthread_getspecific( ) 65
 pthread_mutex_init( ) 65
 pthread_mutex_lock( ) 65
 pthread_mutex_setprioceiling( ) 65
 pthread_mutex_trylock( ) 65
 pthread_mutex_unlock( ) 65
 pthread_mutexatt_init( )
kernel 65
 pthread_setschedparam( ) 63
 timers, kernel
timer_settime( ) 64
 private headers
cacheLib 47
hashLib 48
objLib 40
 processes
glossary definition 2
post-create hooks 22
project, glossary definition 2
pthread_attr_getdetachstate( )
kernl 62
 pthread_attr_getname( )
kernl 62
 pthread_attr_getopt( )
kernl 62
 pthread_attr_getschedpolicy( )
kernl 62

90
Index
VxWorks
5.5 Migration Guide, 6.6

target.ref 79
task self-destruction 19
TASK_SCHED_INFO_GET 19
TASK_SCHED_INFO_SET 19
taskCreate() 46
taskDelete() 19
taskName() 18
taskSpareFieldGet() 19
taskSpareFieldSet() 19
taskSpareNumAllot() 18
taskSwitchHookAdd() 18
scheduling change 46
TCB, accessing 18
timer_settime() 18
kernel, VxWorks 6.3 64
toascii() 40
toolchain, glossary definition 3
torVars 6

U

unld() 32
unldByModuleId() 32
unldByNameAndPath() 32
unloading
unldByModuleId() 32
unldByNameAndPath() 32
user applications, building 45
usrFdiskPartCreate() 56
usrFdiskPartLib 55
usrFsLib.c 57
utilities, compiler 12

V

VIRT_ADDR
BSP macro 75
parameter type change 44
VM_STATE_xxx 74
vmBaseLib
not backward compatible 48
vmBaseLibInit() 48

vmContextCreate() 44
vmContextDelete() 44
vmContextShow() 44
vmCurrentGet() 44
vmCurrentSet() 44
vmGlobalInfoGet() 44
vmGlobalMap() 44
vmMap() 44
vmPageSizeGet() 44
vmShowInit() 44
vmStateGet() 44
vmStateSet()
kernel 44
vmTextProtect() 44
vmTranslate() 44
VxFusion, replaced 21
vxMemProbeArch() 47
vxPowerModeGet() 51
vxPowerModeSet() 51
VxVMI
removed 42
routines, removed
vmContextCreate() 44
vmContextDelete() 44
vmCurrentGet() 44
vmCurrentSet() 44
vmGlobalInfoGet() 44
vmGlobalMap() 44
vmPageSizeGet() 44
vmShowInit() 44
routines, still present in kernel
vmContextShow() 44
vmMap() 44
vmTextProtect() 44
vmTranslate() 44
VxWorks 5.x boot loaders 16
VxWorks image project, glossary definition 2
W

warnings, Wind River Compiler 10
Wind River Compiler 9
Wind River Compiler warnings 10
Wind River GNU compiler 9
WIND_BASE 7
WIND_TCB structure 18
workspace, glossary definition 2
wrconfig build utility 39
wrenv 6
wvEvtLogInit( ) 66
wvLib 66
wvLogHeaderCreate( ) 66
wvLogHeaderUpload( ) 66
wvTaskNamesPreserve( ) 66
wvTaskNamesUpload( ) 66
wvUploadStart( ) 66

X

XBD facility 54
xbdBlkDevLibCreat( ) 55
xbdBlkDevLibDelete( ) 55
xbdBlkDevLibInit( ) 55
xbdCreatePartition( ) 55
xbdRamDiskDevCreat( ) 55
xbdRamDiskDevDelete( ) 55