# Contents

## Changes in This Edition ..................................................................................... xiii

## 1 Introduction ................................................................................................. 1
  1.1 About This Document ........................................................................................ 1
  1.2 The Board Support Package ............................................................................. 2
  1.3 The BSP Development Process ........................................................................ 2
  1.4 Terminology ....................................................................................................... 3

## 2 Overview of a BSP ...................................................................................... 5
  2.1 Introduction ......................................................................................................... 5
  2.2 Boot Sequence .................................................................................................... 7
    2.2.1 Sequence Overview .................................................................................. 7
    2.2.2 Boot Sequence Configurations ................................................................. 10
    2.2.3 Architecture Considerations ..................................................................... 13
    2.2.4 Detailed Boot Sequence ........................................................................... 14
  2.3 Components of a BSP ..................................................................................... 19
    2.3.1 Source and Include Files ......................................................................... 19
      README .......................................................................................................... 20
      Makefile ......................................................................................................... 21
      config.h ......................................................................................................... 21
      bspname.h ...................................................................................................... 21
      sysALib.s ....................................................................................................... 22
      romInit.s ....................................................................................................... 22
      sysLib.c ......................................................................................................... 23
      target.ref ...................................................................................................... 23
      board.jpg ...................................................................................................... 23
      sysDev.c ....................................................................................................... 23
      hwconf.c ...................................................................................................... 24
      configAll.h .................................................................................................... 24
bootInit.c ................................................................. 24
usrConfig.c ............................................................. 24

2.3.2 Derived Files .......................................................... 25

2.3.3 Required Data Variables ............................................. 25

sysPhysMemDesc[ ] (32-bit only) .................................................. 26
sysPhysMemDescNumEnt (32-bit only) ........................................... 27
sysPhysRamDesc[ ] (64-bit only) .................................................. 27
sysPhysRamDescCount (64-bit only) ............................................ 28
sysDma32MemDesc[ ] (64-bit only) .............................................. 28
sysDma32MemDescCount (64-bit only) ........................................ 29
sysUserMemDesc[ ] (64-bit only) ............................................... 29
sysUserMemDescCount (64-bit only) ........................................... 30
sysPmMemDesc[ ] (64-bit only) .................................................. 30
sysPmMemDescCount (64-bit only) ............................................. 31
sysBootLine ................................................................. 31
sysExcMsg ............................................................... 31
sysFlags ............................................................................. 31
hcfDeviceList[ ] ........................................................................ 31
hcfDeviceNum ........................................................................ 31
devUnitCfgData[ ] ........................................................... 32

2.3.4 Required Routines .......................................................... 32

sysBspRev( ) ........................................................................... 33
Timer Driver Routines ............................................................. 33
sysMemDescInit( ) (64-bit only) ................................................ 34
sysMemDescGet( ) (64-bit only) ............................................... 35
sysMemTop( ) (32-bit only) .................................................... 36
sysPhysMemTop( ) (32-bit only) ............................................. 37
sysModel( ) ........................................................................... 38
NVRAM Routines ................................................................. 38
Hardware Initialization Routines ............................................... 39
sysSerialChanGet( ) ........................................................... 39
sysToMonitor( ) .................................................................... 40

2.3.5 Required Macros ........................................................... 41

2.3.6 Optional Routines ........................................................ 44

2.3.7 Optional Macros .......................................................... 44

2.3.8 Hardware Considerations ................................................. 44

2.4 The Development Environment ........................................ 45

2.4.1 BSP Debugging Methods ............................................... 46

Primitive Tools ...................................................................... 46
Native Debug ROMs ............................................................. 46
ROM Emulator ..................................................................... 46
On-Chip Debugging (OCD) Devices ......................................... 47
Logic Analyzer ..................................................................... 47
In-Circuit Emulator ............................................................. 48

2.4.2 WDB Debugging Interface ............................................. 48

2.4.3 Workbench Libraries and Tools ..................................... 49

2.4.4 Compiler and Tool Choice ............................................. 49
## 2.5 Common Problems

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.1 Failing to Include LOCAL_MEM_LOCAL_ADRS</td>
<td>51</td>
</tr>
<tr>
<td>2.5.2 Providing Too Much Device Initialization in romInit.s</td>
<td>51</td>
</tr>
<tr>
<td>2.5.3 Providing Insufficient Initialization in sysALib.s</td>
<td>52</td>
</tr>
<tr>
<td>2.5.4 Including Confusing Configuration Options</td>
<td>52</td>
</tr>
<tr>
<td>2.5.5 Using Non-Maskable Interrupts</td>
<td>53</td>
</tr>
</tbody>
</table>

## 3 Porting a BSP to Custom Hardware

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>55</td>
</tr>
<tr>
<td>3.2 Getting a Default Kernel Running</td>
<td>56</td>
</tr>
<tr>
<td>3.2.1 Initializing the Board</td>
<td>56</td>
</tr>
<tr>
<td>3.2.2 Initializing Memory</td>
<td>57</td>
</tr>
<tr>
<td>3.2.3 Using Debug Routines in the Initialization Code</td>
<td>58</td>
</tr>
<tr>
<td>3.2.4 VxBus Initialization Sequence</td>
<td>60</td>
</tr>
<tr>
<td>3.2.5 Debugging the Initialization Code</td>
<td>61</td>
</tr>
<tr>
<td>Initializing ROM-Based Image Types</td>
<td>61</td>
</tr>
<tr>
<td>Creating Additional Diagnostic Routines</td>
<td>62</td>
</tr>
<tr>
<td>Copying Additional Code From the Reference BSP</td>
<td>63</td>
</tr>
<tr>
<td>Initializing RAM-Based Image Types</td>
<td>64</td>
</tr>
<tr>
<td>Initializing All Image Types</td>
<td>64</td>
</tr>
<tr>
<td>3.2.6 Starting the WDB Agent Before the Kernel</td>
<td>66</td>
</tr>
<tr>
<td>3.2.7 Building and Downloading VxWorks</td>
<td>67</td>
</tr>
<tr>
<td>3.2.8 Interrupt Controllers</td>
<td>68</td>
</tr>
<tr>
<td>3.2.9 DMA</td>
<td>68</td>
</tr>
<tr>
<td>3.2.10 Minimum Required Drivers</td>
<td>68</td>
</tr>
<tr>
<td>3.2.11 Serial Drivers</td>
<td>69</td>
</tr>
<tr>
<td>3.2.12 VxBus Initialization Routines</td>
<td>70</td>
</tr>
<tr>
<td>3.3 Finalizing Your BSP</td>
<td>71</td>
</tr>
<tr>
<td>3.3.1 Removing Development-Related Changes</td>
<td>71</td>
</tr>
<tr>
<td>3.3.2 Creating Workbench Projects</td>
<td>72</td>
</tr>
<tr>
<td>3.3.3 Adding Other Timers</td>
<td>72</td>
</tr>
<tr>
<td>3.3.4 Enabling NVRAM</td>
<td>72</td>
</tr>
<tr>
<td>3.3.5 Enabling Cache and MMU Devices</td>
<td>73</td>
</tr>
<tr>
<td>Cache and MMU BSP Development Overview</td>
<td>74</td>
</tr>
<tr>
<td>Managing Cache and MMU During Reset and Initialization</td>
<td>74</td>
</tr>
<tr>
<td>Initialize VxWorks Cache and MMU Support Layers</td>
<td>76</td>
</tr>
<tr>
<td>Providing Architecture-Specific Cache and MMU Support</td>
<td>82</td>
</tr>
<tr>
<td>Supporting L2 and L3 Cache (Optional)</td>
<td>83</td>
</tr>
</tbody>
</table>
# VxWorks BSP Developer's Guide, 6.9

## Supporting Cache Locking (Optional)

- Defining the Cache and MMU Configuration .............................................. 85
- Additional Information .................................................................................... 93

## 3.3.6 Testing Boot ROMs  ........................................................................................... 94

## 3.3.7 Providing Bus Interface Support .............................................................. 94

- VxBus Support .................................................................................................. 95
- PCI Bus Interface Support .............................................................................. 95
- VME Bus Interface Support ....... ................................................................. 95
- USB Bus Interface Support ............................................................................ 96

## 3.3.8 Updating BSP-Specific Documentation ........................................................ 96

## 3.3.9 Providing Additional Optional Device Support ......................................... 96

- Adding VxBus Devices .................................................................................... 96
- Adding Ethernet Devices ................................................................................ 97
- Adding Other Devices ...................................................................................... 98

## 3.3.10 Troubleshooting and Debugging .............................................................. 99

- SCSI Cables and Termination ......................................................................... 99
- Data Coherency Problems ............................................................................... 99
- Data Address in Virtual Memory Environments ........................................ 99

### 4 Adding BSP Support for VxWorks Features ............................................ 101

#### 4.1 Introduction ................................................................................................. 101

#### 4.2 Power Management ....................................................................................... 101

#### 4.3 PROFILE_BOOTAPP ....................................................................................... 101

- Adding PROFILE_BOOTAPP Support to a BSP ........................................... 102
- Loading the ROM Compress Image to the Boot Device .............................. 104

#### 4.3.2 Migrating Boot Loader Customizations to PROFILE_BOOTAPP .......... 104

### 5 Debugging Your BSP .................................................................................. 107

#### 5.1 Introduction ................................................................................................. 107

#### 5.2 Applying Basic Debugging Techniques ......................................................... 108

- Using LEDs as a Diagnostic Tool ................................................................. 108
- Verifying the Image Location ......................................................................... 108
- Finding Addresses in the Image File .............................................................. 108
- Finding Addresses in the Flash image ........................................................... 109
- Finding Addresses in RAM ............................................................................. 111

#### 5.2.3 Verifying RAM ......................................................................................... 112

#### 5.2.4 Verifying the Image and OS Configuration .................................................. 113

- Post-Processed Compiler Output ................................................................. 113
- Operating System Components Built Into the Image ................................. 113

#### 5.3 Applying Advanced Debugging Techniques ................................................ 115
5.3.1 Symbols ............................................................................................................... 115
5.3.2 Breakpoints ....................................................................................................... 116
Types of Breakpoints ......................................................................................... 116
The Boot Procedure as It Relates to OCD ...................................................... 116
Initial Breakpoints in Flash Images ............................................................... 117
Initial Breakpoints in Downloaded Images ................................................... 117

6 Extending a BSP to an SMP System ................................................................. 119

6.1 Introduction ........................................................................................................... 119
6.2 About VxWorks SMP ............................................................................................ 120
6.3 Development Overview ....................................................................................... 120
6.4 Development Prerequisites .................................................................................. 121
6.4.1 Hardware Documentation ............................................................................... 121
6.4.2 Hardware Support ........................................................................................... 121
6.4.3 VxWorks SMP Architecture Support ............................................................ 122
6.4.4 Existing BSP Support ..................................................................................... 122
6.4.5 VxBus Support ............................................................................................... 123
6.5 Implementation Overview .................................................................................... 123
6.6 Implementation Details ....................................................................................... 125
6.6.1 Creating the Initial BSP .................................................................................. 125
Using Component and Configuration Preprocessor Macros ...................... 125
6.6.2 Updating and Adding CDF Code ................................................................... 126
Mandatory changes ........................................................................................... 127
Optional Changes .............................................................................................. 127
6.6.3 Implementing a Boot Loader Strategy ........................................................... 128
6.6.4 Implementing SMP Boot Logic ...................................................................... 129
Booting the Bootstrap CPU .............................................................................. 130
Booting the Secondary CPUs ........................................................................... 131
6.6.5 Holding a Secondary CPU in Startup or Reset ............................................... 131
6.6.6 Prioritizing Interprocessor Interrupts (IPIs) .................................................. 133
6.6.7 Modifying Cache Handling ......................................................................... 134
cacheEnable( ) and cacheDisable( ) ................................................................. 134
cacheClear( ), cacheFlush( ), and cacheInvalidate( ) .................................... 134
cacheLock( ) and cacheUnlock( ) ................................................................. 135
Modifying Cache Modes to Support Coherency ........................................ 135
6.6.8 Implementing SMP BSP Routines ................................................................ 135
CPU Index and CPU ID .................................................................................... 135
Logical and Physical CPU Index in VxWorks 6.8 Update Pack 1 and Beyond 136
vxCpuPhysIndexGet( ) ..................................................................................... 136
6.6.9 Configuring VxBus (hwconf.c) ................................................................. 141

Configuring Interrupt Controller Drivers .................................................. 141
I/O Device Configuration ............................................................................ 145
Configuring VxBus Memory Allocation ..................................................... 146

6.6.10 Debugging Techniques ........................................................................ 146

6.6.11 Building your BSP ............................................................................... 147

Using the vxprj Command ...................................................................... 147
Using Wind River Workbench ................................................................. 148

7 Extending a BSP to an AMP System ...................................................... 149

7.1 Introduction ........................................................................................... 149

7.2 Overview of BSP Development for AMP ............................................... 150

7.3 Development Prerequisites .................................................................... 151

7.3.1 Hardware Documentation ................................................................. 151

7.3.2 VxWorks AMP Architecture Support .............................................. 151

7.3.3 Existing BSP Support ......................................................................... 152

7.3.4 Custom Components for VxWorks AMP .......................................... 153

7.4 Implementation Overview ..................................................................... 154

7.4.1 Creating and Building a Baseline BSP .............................................. 155

7.4.2 Choosing a Boot Loader Strategy ..................................................... 155

7.4.3 Configuring Memory ......................................................................... 156

7.4.4 Configuring Exception Vectors ......................................................... 156

7.4.5 Adding General Exception and Interrupt Handling ......................... 157

7.4.6 Allocating Devices ............................................................................ 157

Device Driver Limitations and Design Considerations .............................. 157
Incremental Implementation ...................................................................... 158
Special Cases for Development and Debug .............................................. 158

7.4.7 Planning Secondary CPU Handling During Reboot and Restart .......... 158

7.4.8 Implementing Support for the wrload Utility ..................................... 158

7.4.9 Performing Miscellaneous Development Tasks ................................. 159
**A Resolving Common Development Issues** ................................................ 209

A.1 Introduction ............................................................................................................... 209

A.2 The Development Environment .................................................................................. 209
   A.2.1 Image Locations ................................................................................................. 210
   A.2.2 Position-Independent Code ............................................................................ 210

A.3 Exception Handling and Debug Tools ....................................................................... 211

A.4 Cache and MMU ............................................................................................................. 211
   A.4.1 Register Access .................................................................................................. 211
   A.4.2 Timing Issues .................................................................................................... 212

A.5 Non-portable Code ....................................................................................................... 212

A.6 Volatile Variables .......................................................................................................... 212

A.7 Conflicts Between Virtual and Physical Memory ..................................................... 213

**B Implementing Documentation Guidelines** ...................................................... 215

B.1 Introduction ............................................................................................................... 215

B.2 Written Style .............................................................................................................. 216
   Sentences .................................................................................................................. 216
   Punctuation .............................................................................................................. 216
   Word Usage .............................................................................................................. 216
   Spelling .................................................................................................................... 216
   Acronyms ............................................................................................................... 220
   Board Names .......................................................................................................... 222

B.3 Sections for Libraries and Subroutines ...................................................................... 222
   NAME Section ....................................................................................................... 223
   ROUTINES Section ............................................................................................... 223
   SYNOPSIS Section ............................................................................................... 223
   DESCRIPTION Section .......................................................................................... 224
   INCLUDE FILES Section ...................................................................................... 226
   RETURNS Section .................................................................................................. 226
   ERRNO or ERRORS Section .................................................................................. 227
   SEE ALSO Section ................................................................................................. 227

B.4 Sections for target.ref .................................................................................................... 228
   NAME Section ....................................................................................................... 228
   INTRODUCTION Section ....................................................................................... 228
   FEATURES Section ................................................................................................ 229
   HARDWARE DETAILS Section ............................................................................. 229
   SPECIAL CONSIDERATIONS Section ................................................................. 229
   BOARD LAYOUT Section ...................................................................................... 229
   SEE ALSO Section ................................................................................................. 229
   BIBLIOGRAPHY Section ....................................................................................... 230
Changes in This Edition

This edition of VxWorks BSP Developer’s Guide, 6.9 contains the following updates for VxWorks 6.9, Update Pack 2:

- Updates to the VxWorks AMP information. See 7. Extending a BSP to an AMP System.
- Defect fixes and other minor updates.
1 Introduction

1.1 About This Document  
This document describes, in general terms, the elements that make up a board support package (BSP), the requirements for a VxWorks BSP, and the general behavior of a BSP during the boot process. This document outlines the steps needed to port an existing BSP to a new hardware platform or to write a new VxWorks BSP for custom hardware using a reference BSP or template BSP as a starting point. It provides hints and tips for debugging a BSP and solving common BSP development problems. It also provides information on testing and validating your BSP.

The primary audience for this document is developers writing a custom BSP for a specific application platform, using an existing BSP as a starting point for development. In most cases, the document assumes that the developer has a reference BSP similar to the required new BSP. Although this document is also useful for developers writing a new BSP without a reference BSP, the document and the reference BSP are intended for use together. If you are writing a completely new BSP, there are template files available for use in lieu of the reference BSP. However, a template BSP is generally not as complete as a reference BSP.

During BSP development, you may want to consult the following VxWorks companion documents in addition to this manual:

- **VxWorks BSP Validation Test Suite User’s Guide**—This document provides detailed instructions for using the VxWorks BSP validation test suite (BSP VTS). This test suite helps you to assess your BSP functionality.
VxWorks device driver documentation

This includes:

- *VxBus Device Driver Developer’s Guide*—This document discusses the VxBus infrastructure and issues related to writing VxBus model device drivers for VxWorks, the customizations related to writing VxBus model device drivers for one of the supported VxBus device classes, and information on migrating legacy device drivers to the VxBus device driver model. It also includes guidelines for writing custom VxBus model device drivers.

- *VxWorks Legacy Driver Reference Guide*—This document provides the necessary information for maintaining legacy model (non-VxBus) device drivers.

### 1.2 The Board Support Package

A board support package (BSP) is typically composed of C and assembly source files, header files, a makefile, a *readme* file containing version numbers and high-level modification history, and a *target.ref* file containing documentation specific to the BSP.

The purpose of a BSP is to configure the VxWorks kernel for the specific hardware on your target board. In addition, the BSP provides an easy way to maintain portability across many different hardware configurations without having to customize VxWorks, the core operating system (OS). This portability is achieved by defining a boot procedure and a set of routines that are called during the boot process for configuration, and during normal operation for specific kinds of hardware access.

The BSP allows for a well-defined interface between your target hardware and the OS. During the boot process, the BSP routines must call core OS routines and device driver routines to configure a portion of the core OS as well as the device drivers. The OS and well-written device drivers then make calls to the BSP routines during system operation in order to make specific hardware requests.

Wind River provides processor-dependent software as part of each reference BSP. That is, the portions of the BSP that depend only on processor type are done for you. In addition, many hardware drivers are available for each processor type. You can often use these drivers without change or, in most other cases, you can easily modify the drivers to suit specific hardware.

### 1.3 The BSP Development Process

In order to create a functioning BSP, the BSP writer must pass through several stages of BSP development. Briefly, these stages include:
1 Introduction

1.4 Terminology

- configuring the development environment
- minimal hardware or hardware simulator configuration
- gaining a clear understanding of the hardware
- creating a minimal, functioning kernel
- BSP cleanup and the addition of device drivers

Configuration of the development environment includes choosing, installing, and configuring a compiler, debugger, and other tools. You must also determine what download mechanisms to use, including how to program ROM or flash devices, and which hardware debugger to use, if any.

The first step of actual development is writing software to initialize the hardware to a quiescent state. That is, bringing the hardware to a state where it does not generate interrupts that the processor is unable to handle early in the system initialization process. Quiescent initialization code is usually created in assembly language and is typically a short piece of code. Additional code is necessary to reach the point where VxWorks is running. Most of this additional code is written in a higher-level language, such as C.

The amount of time required to bring a BSP to the point that VxWorks is running with just clock, serial, and Ethernet drivers varies greatly. In part, the amount of time depends on how close the reference BSP is to the target hardware, as well as the choice of development environment. It may also depend on what boot configurations are supported. If a hardware debugger, such as the Wind River ICE 2 (in-circuit emulator), is available for debugging and flash programming, if a relatively similar reference BSP is available, and if all normal boot configurations are supported, the BSP development process can take as little as five to six weeks for an experienced developer. However, a typical development process is more likely to take several months.

Once the default kernel is running, additional drivers may be required for the intended application. In most cases, these drivers can be added at a later date, concurrent with application development.

For more information on developing and using drivers with your BSP, see the VxBus Device Driver Developer’s Guide.

1.4 Terminology

The following terminology is used in this document:

 installsDir
Within this document, file paths are typically expressed as a full path; this practice maintains consistency between this and other Wind River documentation.

 bspname
In several places within this document, there are references to file names that are based on the BSP. These filenames have the string bspname substituted. For example, if you are working on a BSP called acmeBSP, change any reference bspname to acmeBSP. For example, bspname.h would become acmeBSP.h.
Where this document refers to devices that the BSP might support, these devices are generically referred to as `dev`. In such cases, substitute the name of each device or device type for `dev`. For example, if your BSP supports both ATA and SCSI devices, change `sysDev.c` to the pair of files `sysAta.c` and `sysScsi.c`.

`projName`

Each project must be given a name. When the project is created, several files are created based on the name given to the newly created project. These files are referred to as `projName`. 
2.1 Introduction

This chapter begins by introducing the BSP routines in the context of the VxWorks boot sequence. Later sections provide descriptions of each of the files containing the BSP routines and the standard preprocessor macros used to configure VxWorks. The chapter finishes with an overview of the BSP development environment and some brief insight into common mistakes.

Before you begin, note that most of the work involved in developing a BSP is accomplished by writing the following routines:

- **romInit() in romInit.s**
  - initializes the CPU and memory

- **sysHwInit() in sysLib.c**
  - ensures that all board hardware is initialized to a quiescent state

- **sysHwInit0() in sysLib.c**
  - performs early BSP-specific hardware initialization. This routine is present only in some BSPs. Refer to your reference BSP to determine if `sysHwInit0()` is required.

- **sysHwInit1() in sysLib.c (64-bit only)**
  - initializes all board hardware and virtual memory

---

NOTE: Some of the information in the following sections is specific to 32-bit systems or 64-bit systems as indicated. For this release, 64-bit VxWorks is only available for the Intel Architecture.
sysHwInit2() in sysLib.c
    further prepares the board hardware for use with VxWorks applications

A comprehensive list of the routines required in every BSP is provided in
2.3.4 Required Routines, p.32.

The following BSP files, device driver directories, and configuration directories are
common to most BSPs:

Required BSP Files:

- installDir/vxworks-6.x/target/config/bspname/bspname.h
- installDir/vxworks-6.x/target/config/bspname/config.h
- installDir/vxworks-6.x/target/config/bspname/Makefile
- installDir/vxworks-6.x/target/config/bspname/README
- installDir/vxworks-6.x/target/config/bspname/romInit.s
- installDir/vxworks-6.x/target/config/bspname/sysALib.s
- installDir/vxworks-6.x/target/config/bspname/sysLib.c
- installDir/vxworks-6.x/target/config/bspname/target.ref

Optional BSP Files:

- installDir/vxworks-6.x/target/config/bspname/sysSerial.c
- installDir/vxworks-6.x/target/config/bspname/configNet.h
- installDir/vxworks-6.x/target/config/bspname/sysEnd.c

VxBus Device Driver Directories:

- installDir/vxworks-6.x/target/src/hwif/busCtrlr
- installDir/vxworks-6.x/target/src/hwif/console
- installDir/vxworks-6.x/target/src/hwif/cpu
- installDir/vxworks-6.x/target/src/hwif/demo
- installDir/vxworks-6.x/target/src/hwif/dmaCtrlr
- installDir/vxworks-6.x/target/src/hwif/end
- installDir/vxworks-6.x/target/src/hwif/end2
- installDir/vxworks-6.x/target/src/hwif/fw
- installDir/vxworks-6.x/target/src/hwif/h
- installDir/vxworks-6.x/target/src/hwif/hEnd
- installDir/vxworks-6.x/target/src/hwif/i2c
- installDir/vxworks-6.x/target/src/hwif/intCtrlr
- installDir/vxworks-6.x/target/src/hwif/mf
- installDir/vxworks-6.x/target/src/hwif/mii
- installDir/vxworks-6.x/target/src/hwif/nvram
- installDir/vxworks-6.x/target/src/hwif/resource
- installDir/vxworks-6.x/target/src/hwif/sio
2.2 Boot Sequence

This section describes the steps in a typical VxWorks boot scenario and identifies which routines implement each step. All processors execute the same logical process when initializing and loading VxWorks, although some may require an extra step or two and others may skip certain steps.

2.2.1 Sequence Overview

At a minimum, initializing a processor consists of providing a portion of code, and possibly some tables, that are located at the specific location in memory that the processor jumps to upon reset or power-up of the target system. This code sets the processor to a specific state, initializes memory and memory addressing, disables interrupts, and then passes control to additional bootstrapping code.

Upon power-up or reset, the processor first jumps to the entry point in ROM, _romInit(). The assembly code at the jump destination sets up memory, initializes the processor status word, and creates a temporary stack. The processor then jumps to a C routine, romStart() which is located in:

installDir/vxworks-6.x/target/config/all/bootInit.c

A parameter in a register or on a temporary stack determines whether memory must be cleared (cold start) and then copies the appropriate sections of ROM into RAM. If the code in ROM is compressed, it is decompressed during the copy. Next, the processor jumps to the VxWorks entry point in RAM.
The VxWorks entry point is `_sysInit()` in `sysALib.s`. This assembly routine sets the initial hardware state (much as `_romInit()` does) and then jumps to `usrInit()` in `usrConfig.c` for BSP directory builds and `prjConfig.c` for project builds. `usrInit()` is the first C routine that runs from a VxWorks image. This routine initializes the cache, clears the block storage segment (bss) to zero, initializes the vector table, performs board-specific initialization, and then starts the multitasking kernel with a user-booting task.

Figure 2-1 provides an overview of the boot sequence when 32-bit VxWorks is booted from an image.

![Boot Sequence Using a 32-Bit VxWorks Image](image-url)}
Figure 2-2 provides an overview of the boot sequence when 64-bit VxWorks is booted from an image.

**Figure 2-2  Boot Sequence Using a 64-Bit VxWorks Image**

- **sysInit**
  - Iniitializes CPU.
  - Initializes RAM (controller).
  - Initializes stack, quiets CPU (disables interrupts).

- **usrInit**
  - Provided in target/config/all/usrConfig.c for BSP directory builds or target/proj/bsp_tool/prjConfig.c for project builds.
  - Performs minimal kernel initialization.

- **usrKernelInit**
  - Performed any remaining CPU/BSP initialization.
  - sysHwInit0 - Provides early BSP-specific initialization.

- **kernelInit**
  - Initializes and starts kernel.
  - Defines system memory partition.
  - Activates task and usrRoot.

- **usrRoot**
  - Initializes the memory partition library and MMU.
  - Initializes the system clock.
  - Activates the application and spawns a task.

- **sysHwInit1**
  - Quiets devices (disables interrupts).

- **intStartupUnlock**
  - Unlocks interrupts.

- **usrSysHwInit2**
  - Completes device initialization.

**NOTE:** For this release, 64-bit VxWorks is only available for the Intel Architecture.
Figure 2-3 illustrates the boot sequence used when VxWorks is booted from a ROM-based loader. Note that the boot sequence after `usrInit()` is the same as that shown in Figure 2-1.

### 2.2.2 Boot Sequence Configurations

There are several boot sequence configurations that are commonly supported. For BSPs intended for specific applications only, at least one of the methods is supported. General-purpose BSPs support all boot methods, unless specified in the BSP documentation.

For development, the most common boot method is through the presence of a ROM-based boot loader. The purpose of this boot loader is to load the final VxWorks image from a remote development host (or possibly from a local file system) and then to start running the newly downloaded image. The boot image is a specially-crafted version of VxWorks with a boot loader as its only application. Because this boot image is largely independent of development changes, it is seldom necessary to re-program the flash, which saves development time. The boot image is referred to as `bootrom`; the OS image that is loaded is referred to as `vxWorks`. These images are described later in the document.

Another alternative boot method is to put the VxWorks image into flash directly. Typically, both the core OS and the application code reside in the same image. In this case, there is no intermediate step between loading the boot loader and loading the application image. Also, no file system is needed to hold the final VxWorks image file. Although this boot method can be used for development, progress can be hindered by the need to re-program the flash often. However, this is a very common boot mechanism in final product delivery.
Whether loading a VxWorks image from flash or loading a boot loader from flash, there are special requirements. The processor’s reset vector causes it to start execution of the code in flash memory. This flash-resident code can do one of several things. It can:

- Continue running from flash (that is, fetch instructions from the flash memory and execute them).
- Copy itself from flash to RAM and branch to an appropriate place in the RAM copy.
- Decompress a compressed image contained in flash and put the image into RAM, then branch to an appropriate place in the RAM copy.

The names of the different image types are determined both by the build method used and by the behavior of the image.

An image that continues running from flash and is built from a project is normally called \texttt{vxWorks\_romResident}. Such an image built from the command line is typically called one of the following image types:

- \texttt{vxWorks\_res\_rom}
- \texttt{vxWorks\_res\_rom\_res\_low}
- \texttt{vxWorks\_res\_rom\_nosym}
- \texttt{vxWorks\_res\_rom\_nosym\_res\_low}
- \texttt{bootrom\_res}
- \texttt{bootrom\_res\_high}

An image that copies itself from flash to RAM and is built from a project is normally called \texttt{vxWorks\_rom}. Such an image built from the command line is normally called \texttt{vxWorks\_st} or \texttt{bootrom\_uncmp}.

An image that decompresses itself and puts the results in RAM, and is built from a project, is generally called \texttt{vxWorks\_romCompress}. Such an image built from the command line is called \texttt{vxWorks\_st\_rom} or \texttt{bootrom}.

If a boot loader loads a VxWorks image, whether from a local file system or from a remote host, the image is referred to as \texttt{vxWorks}, regardless of how it is built.

A more detailed description of each image type is listed below. Much of this information can be found in the following file:

\texttt{installDir/vxworks-6.x/target/h/make/rules.bsp}

\texttt{vxWorks}

The primary VxWorks image that is loaded by a boot loader from a local file system or from a remote host. If a downloaded symbol table is configured, \texttt{vxWorks} requires the \texttt{vxWorks\_sym} file.

\texttt{vxWorks\_sym}

A symbol table that is loaded from the same file system and directory as the \texttt{vxWorks} image itself.

\texttt{vxWorks\_rom}

A VxWorks standalone ROM image programmed into flash that copies itself to RAM for execution. This format is typically used when making an application in ROM that does not include the shell or the symbol table. Because these images are usually smaller, this version does not use ROM compression.
**vxWorks.st**
A VxWorks standalone image that is loaded by a boot loader from a local file system or from a remote host. This VxWorks image has a symbol table already linked in, so it does not need to load `vxWorks.sym` from a local file system or over the network. This image requires a large ROM space and a large RAM space. This image can not be run from ROM.

**vxWorks.st_rom**
A VxWorks standalone image programmed into flash that decompresses itself to RAM for execution. This image includes a linked-in symbol table so that a complete VxWorks image with shell and symbol table is put into ROM. Because these systems tend to be large, ROM compression is used. This rule also creates `vxImage.o` for use as a “core” file (to provide a symbol table) for the target server or other host-based debugger. This image uses less ROM than a `vxWorks` image and requires a large RAM space.

**NOTE:** This image format may require larger EEPROMs. The user is advised to check the macros for ROM sizes and offsets for compatibility.

**vxWorks.res_rom**
A VxWorks image programmed into flash. This image copies the data segment from flash into RAM, but continues to fetch instructions from flash. This image includes a linked-in symbol table so that a complete VxWorks image with shell and symbol table is put into ROM. This type of image uses less RAM than a `vxWorks` image and requires a large ROM space. In general, execution is slow for all ROM-resident images, as compared to RAM-resident images.

**vxWorks.res_rom_res_low**
This image is similar to `vxWorks.res_rom`, but sometimes starts the data segment in RAM closer to `RAM_LOW_ADRS` on some architectures.

**vxWorks.res_rom_nosym**
A VxWorks image programmed into flash. This image copies the data segment from flash into RAM, but continues to fetch instructions from flash. This image does not include a symbol table. This image uses a small amount of RAM and requires a large ROM space. This type of image has a quick start time but executes more slowly than a RAM image.

**vxWorks.res_rom_nosym_res_low**
This image is similar to `vxWorks.res_rom_nosym`, but sometimes starts the data segment in RAM closer to `RAM_LOW_ADRS` for some architectures.

**bootrom**
A VxWorks image with a boot loader application that is programmed into flash. This image decompresses itself into RAM for execution. This image requires a minimal amount of ROM space and a large RAM space.

**bootrom_uncmp**
A VxWorks image with a boot loader application that is programmed into flash. This image copies itself into RAM for execution. This image requires a large amount of both RAM and ROM but executes quickly.

**bootrom_res**
A VxWorks image with a boot loader application that is programmed into flash. This image copies its data segment into RAM for execution, but continues fetching instructions from flash. This image requires a large ROM space and little RAM space.
bootrom_res_high
A VxWorks image with a boot loader application that is programmed into flash. This image copies its data segment into RAM for execution, but continues fetching instructions from flash. This image loads the VxWorks image into a higher location in RAM.

In the information presented above, references are made to bootrom, which is a VxWorks image configured with a boot loader as its application. While this is typically how the image is loaded during development, and in released products as well, it is also possible for VxWorks to be loaded by an external agent of some kind. For example, the image can be loaded by a boot loader not based on VxWorks, such as a ROM monitor, or the image can be loaded directly into RAM by a hardware debugger such as the Wind River ICE 2. For more information, see 2.4 The Development Environment, p.45. Also, consult the documents listed in 1. Introduction.

2.2.3 Architecture Considerations

The VxWorks BSP design was developed to be architecture-independent. However, many architectures and boards have special requirements. This section briefly discusses some of these special considerations. This discussion is not intended to be complete or comprehensive, but is intended to give an idea of the kinds of customizations that may affect a BSP port.

NOTE: The architectures discussed in this section are examples only. Your VxWorks installation may not include support for all architectures described here. For a list of supported architectures, see the Wind River Online Support Web site or your product’s release notes. In addition, architectures not discussed in this section may have special BSP requirements as well. For more information regarding your target architecture, see the appropriate chapter of the VxWorks Architecture Supplement.

The most common memory configuration is for RAM to be located at addresses beginning with 0, and for flash to be addressed in regions of upper memory. However, on some architectures and for some types of applications, it is customary to design the board so that flash is located at address 0 and RAM is in upper memory. In addition, many processors locate their interrupt vectors at address 0 by default. For boards designed in such a way that the interrupt vectors are located in flash, there are two requirements. First, a set of default interrupt vectors must be located at the beginning of the flash image. Second, some mechanism must be made available for the BSP to change the contents of the vectors when VxWorks boots.

Some processors allow both big- and little-endian configurations. Typically, a BSP supports only one byte order type. If both configurations are supported, a second BSP is typically created and named with _le or _be suffix. For example, the bspname BSP is configured for little-endian mode and the bspname_be BSP is configured for big-endian mode. For MIPS, BSP names are followed by a tool specifier. Little-endian MIPS BSPs end with le and big-endian BSPs have no endian suffix. In these situations, the boot ROMs cannot typically boot images from the other byte order.
### 2.2.4 Detailed Boot Sequence

The following is a step-by-step description of a generic boot sequence.

**NOTE:** For 64-bit VxWorks, only project builds are supported (with `vxprj` and Workbench). You cannot use the legacy method using `bspDir/config.h` to configure VxWorks and `bspDir/make` to build it (this method is sometimes referred to simply as a BSP build).

**Step 1: Execute `romInit()`**

**32-Bit**

At power-up (cold start), the processor begins execution at the `romInit()` entry point, located in `romInit.s`. For resets (warm starts), the processor begins execution at `romInit()` plus a small offset (see `sysToMonitor()` in `installDir/vxworks-6.x/target/config/bspname/sysLib.c`). The `romInit()` routine must be written in assembly language.

The purpose of this routine is to initialize the CPU and some portion of memory. It does the absolute minimum amount of initialization—that is, the initialization of essential hardware only—before jumping to `romStart()`. If `romInit()` is working correctly, the memory from `LOCAL_MEM_LOCAL_ADRS` through `(LOCAL_MEM_LOCAL_ADRS + LOCAL_MEM_SIZE)` should be readable and writable. If this is not the case, `romInit()` is not working properly.

**64-Bit**

In 64-bit, `romInit()` sets up the static MMU map. When a 64-bit BSP boots, it will initially be running with the MMU disabled. This creates a very tricky initial runtime environment, because the VxWorks image is loaded very low in the 64-bit physical address space. However, it is linked as if it was running in the upper 2 G of the address space. While the MMU is disabled, the BSP cannot perform any CPU instructions that refer to the absolute (virtual) address of any code or data. This restriction is in place because the virtual addresses of the text and data are not accessible by the CPU until an MMU mapping is created. This maps the virtual address space for the kernel onto the physical address where the VxWorks image was loaded.

To address this limitation, one of the first tasks that the BSP must perform is to turn on the MMU so that the VxWorks image becomes “visible” at the correct virtual address in the upper 2 G of the virtual address space. To accomplish this, the BSP must create the appropriate page table structures to support the initial bootstrap of the kernel. At minimum, this page table structure must contain at least two distinct map areas:

- The static map must provide a mapping of at least all of the RAM described in the 0th `MEM_DESC_RAM` descriptor returned by `sysMemDescGet()`, mapped at the address `LOCAL_MEM_LOCAL_ADRS`. For example, if the 0th descriptor returned by `sysMemDescGet()` describes a 256-MB region of RAM, then the static map must map at least this 256 MB of RAM at `LOCAL_MEM_LOCAL_ADRS`. The downloaded VxWorks image is linked to run within this memory area, so this part of the memory map is the part that is used to execute kernel code once the MMU is enabled.
The static map must also map a very small portion of the VxWorks image with an identity mapping, to avoid causing a page fault right after the MMU is enabled.

32-Bit and 64-Bit

In addition to initializing memory as described above, the `romInit()` routine must also disable interrupts and clear caches. `romInit()` then configures the boot type (cold or warm) to be a subroutine argument and branches to `romStart()` in `bootInit.c`. For more information on configuring the boot type, see 2.3.8 Hardware Considerations, p. 44.

`romInit()` must do only as much device setup as is required to start executing C code. Hardware initialization is the responsibility of the `sysHwInit()` (for 32-bit) or `sysHwInit1()` (for 64-bit) routine which is called later during the boot sequence. These routines are located in:

```
installDir/vxworks-6.x/target/config/bspname/sysALib.s
installDir/vxworks-6.x/target/config/bspname/sysLib.c
```

**Step 2: Execute `romStart()`**

The purpose of the `romStart()` routine is to move all further bootstrap code from ROM into RAM and then, if necessary, jump to the VxWorks image. Because this implementation depends only on the CPU architecture, the `romStart()` routine is provided by Wind River and is located in the file `bootInit.c`. Typically, `romStart()` jumps to the `sysInit()` routine in RAM.

The required execution steps are as follows:

1. Copy the data segment from flash to ROM. Depending on the image type, you may also need to copy the text segment. If necessary, decompress the data during the copy.
2. Clear unused RAM.
3. The `romStart()` routine then jumps to the RAM entry point, `sysInit()`, which is located in `sysALib.s`. The boot type (cold or warm) is passed as an argument to `sysInit()`. Note that BSP directory built ROM images jump to `usrInit()` from `romStart()`. Project builds jump to `sysInit()`.

**Step 3: Execute `sysInit()`**

The `sysInit()` routine is the RAM entry point. `sysInit()`—which should be the first routine defined in `sysALib.s”—invalidates caches if applicable, initializes the system interrupt tables with default stubs, initializes the system fault tables with default stubs, and initializes all processor registers to known default values. The routine also enables tracing, clears all pending interrupts, and finally invokes `usrInit()` with the argument `bootType`.

This routine must duplicate much of the hardware initialization done by `romInit()` in order to set the run-time state rather than the boot state. Keep in mind that the board may have been booted using a ROM monitor or hardware debugger. In this case, the VxWorks boot ROM code, where `romInit()` is located, is not executed and VxWorks system initialization is not performed. Therefore, failure to duplicate the initialization code from `romInit()` may result in BSP failure.
Step 4: Execute **usrInit()**

The purpose of the **usrInit()** routine is to completely initialize the CPU and shut down any other hardware, thus preparing the way for the kernel to initialize and start itself. This routine is located in **usrConfig.c** for BSP directory builds and **prjConfig.c** for project builds, but calls routines in several other files, some of which you must provide and some of which are provided by Wind River. Normally, it is not necessary to modify the **usrInit()** routine provided by Wind River. However, the **sysHwInit()** routine that must be called from **usrInit()** typically does require modification. The **sysHwInit()** routine ensures that the board-dependent hardware is quiescent (for 32-bit). **sysHwInit()** is provided in the reference BSP or template and is located in **sysLib.c**.

The **usrInit()** routine (in **usrConfig.c/prjConfig.c**) saves information about the boot type, handles all initialization that must be performed before the kernel is actually started, and then starts the kernel execution. It is the first C code to run in VxWorks. This routine is invoked in supervisor mode with all hardware interrupts locked out.

Many facilities cannot be invoked from this routine. There is no task context yet—that is, no task control block (TCB) and no thread stack—therefore, facilities that require a task context cannot be invoked. This includes any facility that can cause the caller to be preempted (such as semaphores) or any facility that itself uses a facility of this type, such as **printf()**. Instead, the **usrInit()** routine does only what is necessary to create the initial thread, **usrRoot()**. **usrRoot()** then completes the startup.

The initialization operations performed in **usrInit()** include the following:

- Initializing cache.
  
  The code at the beginning of **usrInit()** initializes the caches, sets the mode of the caches, and puts the caches in a safe state. At the end of **usrInit()**, the instruction and data caches are enabled by default.

- Zeroing out the system BSS segment.
  
  C language specifies that all uninitialized variables (stored in bss) must have initial values of 0. Because **usrInit()** is the first C code to execute, it clears the section of memory containing bss.

- Initializing interrupt vectors.
  
  The exception vectors must be initialized before enabling interrupts and starting the kernel. First, **intVecBaseSet()** is called to establish the vector table base address.

  **NOTE:** The **intVecBaseSet()** routine is not called first on all architectures. For more information, see the appropriate chapter of the *VxWorks Architecture Supplement*.

After **intVecBaseSet()** is called, the routine **excVecInit()** initializes all exception vectors to default handlers. These handlers safely trap and report exceptions caused by program errors or unexpected hardware interrupts.
Overview of a BSP

2.2 Boot Sequence

- Initializing system hardware to a quiescent state.

Calling the system-dependent routine \texttt{sysHwInit() } initializes the system hardware. Initialization mainly consists of resetting and disabling hardware devices. Otherwise, when the kernel is started and interrupts are enabled, these devices can cause unexpected interrupts.

In VxWorks, ISRs (for I/O devices, system clocks, and so on) are not connected to their interrupt vectors until system initialization is completed by the \texttt{usrRoot() } task. This is a requirement because the memory pool is not yet initialized. You must not connect an interrupt handler to an interrupt during the \texttt{sysHwInit()} call; doing so requires memory allocation, which is not available at this time. Most interrupt connection occurs later in the \texttt{sysHwInit2()} routine located in \texttt{sysLib.c}.

\begin{itemize}
  \item Calling \texttt{kernelInit() }. \end{itemize}

The VxWorks libraries contain the code for \texttt{kernelInit()} . Therefore, it is not normally available to BSP developers in source form. The \texttt{kernelInit()} routine initiates the multitasking environment and never returns. It takes the following parameters:

- address of the routine to be spawned as the root task, typically \texttt{usrRoot()} 
- stack size 
- start of usable memory; that is, the memory after the main text, data, and bss segments of the VxWorks image. All memory after this area is allocated to the system memory pool, which is managed by \texttt{memPartLib}. All cached dynamic allocations are derived from this memory pool.
- top of cached memory as indicated by \texttt{sysMemTop()} (32-bit only)
- interrupt stack size. The interrupt stack corresponds to the largest amount of stack space that can be used by any interrupt-level routine that may be called, plus a safety margin for the nesting of interrupts.
- interrupt lockout level. For architectures that have a level concept, it is the maximum level. For architectures that do not have a level concept, it is a mask to disable interrupts. For more details, see the appropriate chapter of the \texttt{VxWorks Architecture Supplement}.

\textbf{Step 5: Execute kernelInit()}

The \texttt{kernelInit()} routine is provided by Wind River in a VxWorks library archive file. The purpose of this routine is to get the kernel up and running so that all further initialization can be done as a task running under the kernel. The name of this task is \texttt{tRootTask}, and the routine it executes is typically \texttt{usrRoot()}.
The kernelInit() routine calls intLockLevelSet(), disables round-robin scheduling mode, and creates an interrupt stack (if supported by the architecture). The routine then creates a root stack and a task control block (TCB) from the top of the memory pool, spawns the root thread usrRoot(), and terminates the usrInit() thread of execution. At this time, interrupts are enabled. It is critical that all interrupt sources be disabled by usrInit(), and that all pending interrupts be cleared. Failure to do so causes system failure as described in Creating Additional Diagnostic Routines, p.62.

Step 6: Execute usrRoot() as a task

The purpose of the usrRoot() routine is to complete the initialization of the kernel and all hardware, then launch any application code. This routine is supplied by Wind River in usrConfig.c for BSP directory builds and prjConfig.c for project builds. The original copy should not be changed. During development, usrConfig.c (for BSP directory builds only) is often temporarily copied to the BSP directory for BSP directory builds, and debugging changes are made to the temporary copy. usrConfig.c is also configurable with the macros defined in config.h for BSP directory builds.

NOTE: Do not use a custom version of usrConfig.c/prjConfig.c in your final BSP.

The usrRoot() routine calls the memInit() routine. Optionally, usrRoot() can call the memShowInit() and usrMmuInit() routines. Once the system is multitasking, the BSP calls its first routine, sysClkConnect(). sysClkConnect() immediately calls the sysHwInit2() routine. sysHwInit2() is responsible for any board initialization not completed in sysHwInit(), such as the connection of interrupt sources using intConnect().

32-Bit

Next, the usrRoot() routine continues the clock initialization. It sets the default clock rate to the value of the macro SYS_CLK_RATE, typically 60 Hz. usrRoot() then enables the system clock with a call to sysClkEnable().

NOTE: The system clock can be dynamically changed from the shell or from an application. However, facilities that take a “snapshot” of the clock rate (for example, spyLib) can be broken by an unexpected rate change.

Once the clock is initialized and running, several kernel modules, such as selectLib, the I/O subsystem, and the console, are initialized. See the source code in usrConfig.c, which uses configAll.h and config.h for BSP directory builds, and prjConfig.c, which uses 20bsp.cdf (and other CDF files) and config.h for project builds.

If INCLUDE_WDB is defined, wdbConfig() is called. The wdbConfig() routine is located in:

installDir/vxworks-6.x/target/src/config/usrwdb.c

This routine initializes the agent’s communication interface, and then starts the debug agent. For information on configuring the agent, see the VxWorks Kernel Programmer’s Guide. The debug agent is the portion of VxWorks that connects to and serves the Workbench tools such as the shell and the debugger.
If the `INCLUDE_USR_APPL` macro is defined, the default `usrRoot()` code executes the `USER_APPL_INIT` macro. This macro allows you to start your application automatically at boot time. The BSP assumes that the `USER_APPL_INIT` macro is a valid C statement. For more information on `USER_APPL_INIT`, see the VxWorks Kernel Programmer's Guide: Kernel Applications.

2.3 Components of a BSP

The BSP directory contains source files, header files, a makefile (Makefile), one or more documentation files (target.ref), and possibly other files such as derived files and object modules distributed in object format only.

From the BSP directory, other derivative files may be generated; for example, there may be files generated for a project. Some derived files are not put into the BSP directory itself. These include the WPJ project file, additional source and header files, a custom `usrAppInit.c` file, another makefile for use with the project facility, and directories containing object modules.

In addition to the BSP directory, there are several other directories that contain files related to the BSP.

The following directory contains some source and header files that are used by the BSPs for default configuration:

`installDir/vxworks-6.x/target/config/all`

The files in this directory should never be changed. If changes to a file are necessary, copy the file to the BSP directory and make changes to the local copy.

The makefile provides a mechanism that allows you to use the local copy instead of the common version. The following macros designate a private copy of the related files:

- USRCONFIG = `usrConfig.c`
- BOOTINIT = `bootInit.c`
- DATASEGPad = `dataSegPad.c`

For example, to use a private version of `usrConfig.c`, set the macro `USRCONFIG` to `usrConfig.c` in Makefile as follows (for BSP directory builds only):

`USRCONFIG = usrConfig.c`

2.3.1 Source and Include Files

BSP routines are contained in a number of C and assembly files that you may need to modify or create. These routines are located in a relatively small number of source files. This section provides a list of these source files as well as a detailed description of each file. Each file is also documented extensively in the reference and template BSPs (see the code comments in each file).
The following files, in `installDir/vxworks-6.x/target/config/bspname`, may need to be created or modified:

- **README** documentation file
- **Makefile** makefile for building the BSP
- **config.h** header file for configuring the OS
- **bspname.h** header file for non-configurable definitions
- **romInit.s** includes the `romInit()` routine and any subroutines used by `romInit()`.
- **sysALib.s** contains assembly routines that are not part of `romInit()` (optional).
- **sysLib.c** contains the `sysHwInit()` routine and additional C language routines specific to the target hardware.
- **sysDev.c** If used, this file contains the device driver interface to the physical hardware device, `Dev`.

Note that the actual device driver files are kept in following directories:

For VxBus model device drivers:

```
installDir/vxworks-6.x/target/src/hwif/drvType
```

There may be multiple `sysDev.c` files.

- **target.ref** BSP documentation file (uses the `apigen` markup language). For more information on BSP documentation files, see B. Implementing Documentation Guidelines, or the reference entry for `apigen`. (VxWorks 5.5 users can consult the reference entry for `refgen`.)

The following files, located in `installDir/vxworks-6.x/target/config/all`, may be copied to the BSP directory and modified during BSP development by adding diagnostic statements. Typically, the files used in the final version of the BSP are not modified.

- **configAll.h** global header file for configuring the OS
- **bootInit.c** contains the `romStart()` routine (for BSP directory builds only)
- **usrConfig.c** contains the `usrInit()` and `usrRoot()` routines (for BSP directory builds only)

Typically, the following file, in `installDir/vxworks-6.x/target/config/comps/src`, is not modified:

- **usrKernel.c** This file contains all the `#ifdef` s that are controlled by makefile macros to configure the kernel.

**README**

Use this plain-text file to document the history of the BSP. That is, include information on when you wrote the BSP, any modifications made, which BSP was used as a template, what general changes you made, and so on.
2 Overview of a BSP

2.3 Components of a BSP

Makefile

This file controls the build of the VxWorks image. You must set several variables within this makefile. The required variables are listed and described in the 2.3.5 Required Macros, p.41.

config.h

This header file contains all #include and #define macros specific to configuring the CPU and board architectures. It also contains any necessary overrides to the macro definitions in configAll.h, which you must include using #include. You must also include the bspname.h header in this file.

bspname.h

This header file contains all the header information for the BSP that is not related to OS configuration. The information in this header file is required under all configurations of VxWorks using this BSP. This file includes any headers for device drivers.

Use bspname.h to set all non-optional, board-specific information, as defined in this file; including definitions for the serial interface, timer, and I/O devices.

This file is intended for constant information that is not subject to user configuration. If any macros or values defined in this file can be changed to customize this system, define those macros or values in config.h instead.

NOTE: In general, use the config.h file to define configurable values and the bspname.h file to define values fixed in hardware.

When developing your BSP, it is helpful to use a sample header file as a starting point. In most cases, the sample file requires minimal modification because most constant names, basic device addresses, and so on are already defined in the sample file. Define the following in bspname.h:

interrupt vectors and levels
  Define all interrupt vectors and levels that are dictated by hardware in bspname.h.

I/O device addresses
  Define all I/O addresses fixed by hardware in bspname.h.

meaning of device register bits
  For on-board control registers, define a macro value for each bit or group of bits in each register. Place such macro definitions in bspname.h if there is no better location (such as a device-specific header file) for them.

system and auxiliary clock parameters
  Define maximum and minimum rates.

NOTE: It is advisable to include macros describing all available bits, or groups of bits in each control register, even if those bits are not used by the BSP.
sysALib.s

This file contains the RAM image’s entry point, `sysInit()`. `sysInit()` performs any required hardware-specific initialization before jumping to `usrInit()` in `usrConfig.c` for BSP directory builds and `prjConfig.c` for project builds.

Any additional utility routines that must be written in assembly language and are required during normal system operation are also contained in `sysALib.s`.

romInit.s

This assembly file contains the `romInit()` routine, which is the entry point for bootstrapping, plus any `romInit()` subroutines. The `romInit()` routine must be the first routine in the text segment of `romInit.s`.

At power-up (cold start) the processor begins execution at `romInit()`. For warm starts, the processor begins execution at `romInit()` plus a small offset (see `sysToMonitor()` in `sysLib.c`). Most hardware and device initialization is performed later in the boot sequence by `sysHwInit()`, which is located in `sysLib.c`. The job of `romInit()` is to perform the minimal setup needed to transfer control to `romStart()`, located in:

- `installDir/vxworks-6.x/target/config/all/bootInit.c` (BSP directory builds only)
- `installDir/vxworks-6.x/target/config/comps/src/romStart.c` (project builds only)

The minimal setup includes:

- Initializing the processor (this code is specific to the processor but not the board, and thus can be copied from a reference BSP):
  - mask processor interrupts
  - set the initial stack pointer to `STACK_ADRS` (defined in `configAll.h`)
  - disable processor caches

- Initializing access to target DRAM as needed for the following (this code is board-specific):
  - wait states
  - refresh rate
  - chip-selects
  - disabling secondary (L2) caches (if needed)

At the end of the initialization sequence, `romInit()` jumps to `romStart()` in `bootInit.c`, passing the start type. The start type is `BOOT_COLD` for a cold boot, or the parameter passed from `sysToMonitor()` on a warm boot.

For more information, see `romInit.s` in a reference BSP or the template `romInit.s` file in the template BSP. Also see 2.3.8 Hardware Considerations, p.44.
sysLib.c

The `sysLib.c` file contains the routines that directly or indirectly initialize all hardware device drivers. The principal routines for initializing the hardware drivers are `sysHwInit()` and `sysHwInit2()`, but additional driver initialization routines are called by `usrRoot()`, such as `sysClkConnect()`, which calls `sysHwInit2()` in a 32-bit boot sequence. These routines are described in 2.3.4 Required Routines, p.32. Also see the source code in your reference BSP or the template BSP.

While `sysLib.c` is the largest BSP file, in the early phases of BSP development it is advisable to implement only the basics, including `sysModel()`, `sysBspRev()`, `sysHwInit()`, `sysHwInit0()` (for some BSPs), `sysHwInit1()` (64-bit only), `sysHwInit2()`, and `sysMemTop()` (32-bit only).

The file `sysLib.c` also includes the following NVRAM stub drivers during initial development:

```c
#include "mem/nullNvram.c"
#include "vme/nullVme.c"
```

Additional information about NVRAM support is provided in 3.3.4 Enabling NVRAM, p.72.

The `sysHwInit()` routine is the heart of `sysLib.c`, and most of the initial work is done here. `sysHwInit()` is the routine that resets all devices to a quiescent state so that they do not generate interrupts later on when interrupts are enabled.

**NOTE:** When hardware features are missing, it is usually safe to code an empty stub in `sysLib.c`. If any return value is required, the stub can return `ERROR` or `NULL`, whichever is appropriate.

target.ref

This file describes the BSP, and is used to generate automatic online documentation in HTML format. To format the documentation as HTML, type `make man` in the BSP directory.

For more information on updating and using this file, see 3.3.8 Updating BSP-Specific Documentation, p.96, and B. Implementing Documentation Guidelines.

board.jpg

This optional file contains a JPEG image of the target system board. You may want to label the images to show the serial connector, power connector, and Ethernet connector.

sysDev.c

VxBus drivers do not require `sysDev.c` files. (For more information on VxBus, see the VxBus Device Driver Developer's Guide: Device Driver Fundamentals.) For information about `sysDev.c` files, see VxWorks Legacy Driver Reference Guide.
hwconf.c

The *hwconf.c* file contains hardware configuration information used by VxBus. This information is kept in tables, and is read by VxBus during configuration. There is no executable code present in this file. For more information on VxBus, see the *VxBus Device Driver Developer’s Guide*.

configAll.h

This file contains the default OS configuration for all BSPs, and should only be changed under extreme circumstances. Changing this file makes your BSP incompatible with all default VxWorks installations. Any make variable defined in this file can be overridden in *config.h*.

⚠️ **CAUTION:** Changing *configAll.h* can impact other VxWorks users sharing your VxWorks installation.

bootInit.c

The *bootInit.c* file (for BSP builds) contains *romStart( )*, which is the first C routine to execute during the VxWorks boot process. This routine is architecture-independent and should not require changing.

 الإمارات **NOTE:** The *bootInit.c* file is shared by all BSPs. Ensure that changes made to *bootInit.c* do not impact the functionality of other BSPs.

The routine *romStart( )* is essentially a loader. It performs the necessary decompression and relocation for the ROM images. First, it copies the text and data segments from ROM to RAM. Then, it clears those parts of the main RAM not otherwise used. Finally, *romStart( )* decompresses the compressed portion of the image. Different configuration options can modify how these operations are performed.

 Emirates **NOTE:** For project builds, *romStart( )* is located in *installDir/vxworks-6.x/target/config/comps/src*.

usrConfig.c

The *usrConfig.c* file (for BSP builds) contains the *usrInit( )* and *usrRoot( )* routines. When *usrRoot( )* completes, the VxWorks boot is complete. Both *usrInit( )* and *usrRoot( )* are architecture-independent, and thus should not require changing.

 Emirates **NOTE:** The *usrConfig.c* file is shared by all BSPs. Because of this, you must ensure that changes made to *usrConfig.c* do not impact the functionality of other BSPs.
During BSP development, the `usrConfig.c` file is usually copied into the BSP directory and modified to allow the user to control what hardware is initialized (this is achieved by removing portions of `usrRoot()` using `#if FALSE/#endif` pairs). The temporary copy must be discarded when the BSP is complete. Therefore, changes are limited to debug code only.

NOTE: For project builds, the project created `prjConfig.c` contains `usrInit()` and `usrRoot()`.

### 2.3.2 Derived Files

The most obvious files derived from the BSP files are the VxWorks images. There may be multiple copies of VxWorks images present in a BSP directory, each representing a different build type. For example, it is possible for the BSP directory to contain `bootrom`, `bootrom_uncmp`, `vxWorks` (and its `vxWorks.sym` file), `vxWorks.st`, and `vxWorks.res_rom` as well as several other images all at the same time.

In addition, building the source files in the BSP directory generates object files. There is not a one-to-one correspondence between source files and object files.

Another source of derived files is the `target.ref` documentation file in the BSP directory. This file is maintained in an unprocessed form that is difficult to read. To process the `target.ref` file, issue a `make man` command in the BSP directory. This command results in a new HTML file that is suitable for online viewing. For more information on the HTML output or the `target.ref` file, see B. Implementing Documentation Guidelines.

Other objects derived from files in the BSP directory are related to the project facility. Each time a bootable project is created from a BSP directory, a `projName.wpj` file is created. A project also contains several derived source and header files including a makefile for use with the project facility, a `prjObjs.lst` file containing a list of all the object modules used for the build, and directories containing the object modules and VxWorks images themselves. Note that although these objects are derived from information in the BSP directory, the objects themselves are not located in the BSP directory.

### 2.3.3 Required Data Variables

You must declare and initialize the following data variables in your BSP:

```c
sysPhysMemDesc[]  determines physical memory layout (32-bit only)
sysPhysMemDescNumEnt number of entries in `sysPhysMemDesc[]` (32-bit only)
sysPhysRamDesc[]  determines kernel RAM memory layout (64-bit only)
sysPhysRamDescCount number of entries in `sysPhysRamDesc[]` (64-bit only)
sysDma32MemDesc[] determines 32-bit DMA memory layout (64-bit only)
```
The `sysPhysMemDesc[]` array holds descriptions of different memory ranges on the board.

```c
struct PHYS_MEM_DESC sysPhysMemDesc[];
```

The array is used by several VxWorks libraries, including the memory management subsystem. This array must be initialized by the BSP. Typically, it is initialized statically during compile time, but it may also be initialized dynamically early in the boot process.

Values for statically defined `sysPhysMemDesc[]` entries are assigned using descriptive macro names.

The kinds of memory described by `sysPhysMemDesc[]` entries include system RAM, flash, or ROM, memory mapped processor registers, board registers, memory mapped device registers including PCI memory space, vector table, static RAM, or anything else that uses a memory mapped address range.

The fields included in the `PHYS_MEM_DESC` structure are, in order:

- **virtualAddr**: the virtual address of the beginning of the memory region
- **physicalAddr**: the physical address of the beginning of the memory region
- **len**: the length, in bytes, of the memory region
- **initialStateMask**: a mask for the initial VM state

---

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sysDma32MemDescCount</td>
<td>number of entries in <code>sysDma32MemDesc[]</code> (64-bit only)</td>
</tr>
<tr>
<td>sysUserMemDesc[]</td>
<td>determines user reserved memory layout (64-bit only)</td>
</tr>
<tr>
<td>sysUserMemDescCount</td>
<td>number of entries in <code>sysUserMemDesc[]</code> (64-bit only)</td>
</tr>
<tr>
<td>sysPmMemDesc[]</td>
<td>determines persistent memory layout (64-bit only)</td>
</tr>
<tr>
<td>sysPmMemDescCount</td>
<td>number of entries in <code>sysPmMemDesc[]</code> (64-bit only)</td>
</tr>
<tr>
<td>sysBootLine</td>
<td>address of boot line</td>
</tr>
<tr>
<td>sysExcMsg</td>
<td>catastrophic message area</td>
</tr>
<tr>
<td>sysFlags</td>
<td>boot flags</td>
</tr>
<tr>
<td>hcfDeviceList[]</td>
<td>static hardware configuration information</td>
</tr>
<tr>
<td>hcfDeviceNum</td>
<td>number of devices in <code>hcfDeviceList[]</code></td>
</tr>
<tr>
<td>devInitCfgData[]</td>
<td>device configuration information</td>
</tr>
</tbody>
</table>
**initialState**

the initial state to which the memory management library should set the memory

**NOTE:** VxWorks for 32-bit architectures supports 32-bit virtual addressing. However, on hardware that provides a physical address space larger than 32 bits, VxWorks supports (on certain architectures) a larger physical address space. For specific information on the physical addressing space for your target architecture, see the *VxWorks Architecture Supplement*.

The masks used in **initialStateMask** are defined in *vmLib.h*:

```
VM_STATE_MASK_VALID
VM_STATE_MASK_WRITABLE
VM_STATE_MASK_CACHEABLE
VM_STATE_MASK_MEM_COHERENCY
VM_STATE_MASK_GUARDED
```

Additional mask values may be available on some architectures. Check *vmLib.h* and the appropriate architecture supplement for more information.

The following state values are available:

```
VM_STATE_VALID
VM_STATE_WRITABLE
VM_STATE_CACHEABLE
VM_STATE_MEM_COHERENCY
VM_STATE_GUARDED
```

Additional state values may be available on some architectures. Check *vmLib.h* and the appropriate architecture supplement for more information.

**sysPhysMemDescNumEnt** (32-bit only)

The **sysPhysMemDescNumEnt** variable contains the number of entries in **sysPhysMemDesc[ ]**.

```
int sysPhysMemDescNumEnt
```

This variable is typically initialized statically at compile time.

**sysPhysRamDesc[ ]** (64-bit only)

The **sysPhysRamDesc[ ]** array holds descriptions of different kernel RAM memory ranges on the board for 64-bit BSPs.

```
LOCAL PHYS_MEM_DESC sysPhysRamDesc []
```

The first region of kernel RAM memory must be mapped at **LOCAL_MEM_LOCAL_ADRS**. More regions can be mapped at the discretion of the kernel. By supporting multiple memory regions, the BSP can provide an arbitrary amount of memory for the kernel.

When memory auto-sizing is used, additional blank entries are initially specified. These blank entries are populated as the auto-sizing routine probes the memory regions. Other entries are only used if the BSP discovers a kernel RAM memory block during system boot. The kernel and its included components are responsible
for partitioning the memory into the user-configured regions (kernel common heap, kernel proximity heap, and so on).

The fields included in the 64-bit PHYS_MEM_DESC structure are, in order:

- **virtualAddr**
  - the virtual address for each block of kernel RAM memory region

- **physicalAddr**
  - the physical address for each block of kernel RAM memory region

- **len**
  - the length, in bytes, of the memory region

- **initialStateMask**
  - a mask for the initial VM state

  The masks used in **initialStateMask** are defined in **vmLib.h**. The mask names begin with **VM_STATE_MASK_** (For example, **VM_STATE_MASK_VALID**, **VM_STATE_MASK_WRITABLE**, and so on.)

- **initialState**
  - the initial state to which the memory management library should set the memory

  The states used in **initialState** are defined in **vmLib.h**. The state names begin with **VM_STATE_** (For example, **VM_STATE_VALID**, **VM_STATE_WRITABLE**, and so on.)

**sysPhysRamDescCount** (64-bit only)

The **sysPhysRamDescCount** variable contains the number of entries in **sysPhysRamDesc[]**.

```c
LOCAL int sysPhysRamDescCount
```

Typically this variable is initialized statically at compile time.

**sysDma32MemDesc[]** (64-bit only)

The **sysDma32MemDesc[]** array holds a region of RAM that is reserved for 32-bit DMA operations in 64-bit BSPs.

```c
LOCAL PHYS_MEM_DESC sysDma32MemDesc []
```

Typically, the BSP must ensure that the physical addresses for this RAM fall below the 4 GB address space boundary. However, some hardware architectures with IOMMU capabilities allow this memory to be placed elsewhere in the physical address space. It is the responsibility of the BSP developer to ensure that the memory described by this descriptor can be accessed by devices that are only capable of providing a 32-bit address for DMA operations. Descriptors of this type must provide a physical address, but do not provide a virtual address. This memory should be reserved by the BSP only if **INCLUDE_CACHE_DMA32_LIB** is included as a component.

The fields included in the 64-bit PHYS_MEM_DESC structure are, in order:

- **virtualAddr**
  - the virtual address for each block of 32-bit DMA memory region
2 Overview of a BSP
2.3 Components of a BSP

physicalAddr
the physical address for each block of 32-bit DMA memory region

len
the length, in bytes, of the memory region

initialStateMask
a mask for the initial VM state
The masks used in initialStateMask are defined in vmLib.h. The mask names begin with vm_STATE_MASK_ (For example, VM_STATE_MASK_VALID, VM_STATE_MASK_WRITABLE, and so on.)

initialState
the initial state to which the memory management library should set the memory
The states used in initialState are defined in vmLib.h. The state names begin with vm_STATE_ (For example, VM_STATE_VALID, VM_STATE_WRITABLE, and so on.)

sysDma32MemDescCount (64-bit only)
The sysDma32MemDescCount variable contains the number of entries in sysDma32MemDesc[ ].
LOCAL int sysDma32MemDescCount
Typically this variable is initialized statically at compile time.

sysUserMemDesc[ ] (64-bit only)
The sysUserMemDesc[ ] array holds a region of RAM that is owned by the application in 64-bit BSPs.
LOCAL phys_mem_desc sysUserMemDesc[ ]
The kernel does not perform any mapping operations on this memory region. If a user application wants to map this RAM, it must allocate the required virtual address space and perform the mapping operation itself. This memory should be reserved by the BSP if and only if INCLUDE_USER_RESERVED_MEMORY is included as a component.
The fields included in the 64-bit phys_mem_desc structure are, in order:
virtualAddr
the virtual address for each block of 32-bit DMA memory region
physicalAddr
the physical address for each block of 32-bit DMA memory region
len
the length, in bytes, of the memory region
initialStateMask
a mask for the initial VM state
The masks used in initialStateMask are defined in vmLib.h. The mask names begin with vm_STATE_MASK_ (For example, VM_STATE_MASK_VALID, VM_STATE_MASK_WRITABLE, and so on.)
initialState
the initial state to which the memory management library should set the memory

The states used in initialState are defined in vmLib.h. The state names begin with VM_STATE_. (For example, VM_STATE_VALID, VM_STATE_WRITABLE, and so on.)

sysUserMemDescCount (64-bit only)

The sysUserMemDescCount variable contains the number of entries in sysUserMemDesc[].

LOCAL int sysUserMemDescCount

Typically this variable is initialized statically at compile time.

sysPmMemDesc[] (64-bit only)

The sysPmMemDesc[] array holds a region of RAM that is reserved for persistent memory in 64-bit BSPs.

LOCAL PHYS_MEM_DESC sysPmMemDesc[]

Descriptors of this type must provide a physical address, but do not provide a virtual address. This memory should be reserved by the BSP only if INCLUDE_EDR_PM is included as a component.

The fields included in the 64-bit PHYS_MEM_DESC structure are, in order:

virtualAddr
the virtual address for each block of 32-bit DMA memory region

physicalAddr
the physical address for each block of 32-bit DMA memory region

len
the length, in bytes, of the memory region

initialStateMask
a mask for the initial VM state

The masks used in initialStateMask are defined in vmLib.h. The mask names begin with VM_STATE_MASK_ (For example, VM_STATE_MASK_VALID, VM_STATE_MASK_WRITABLE, and so on.)

initialState
the initial state to which the memory management library should set the memory

The states used in initialState are defined in vmLib.h. The state names begin with VM_STATE_ (For example, VM_STATE_VALID, VM_STATE_WRITABLE, and so on.)
sysPmMemDescCount (64-bit only)

The sysPmMemDescCount variable contains the number of entries in sysPmMemDesc[].

LOCAL int sysPmMemDescCount

This variable is typically initialized statically at compile time.

sysBootLine

The sysBootLine string pointer points to the boot line that is configured for use when booting the system.

char *sysBootLine

The boot information is parsed using bootLineCrack(). It is typically set to the value BOOT_LINE_ADRS, and points to the boot line information in NVRAM.

sysExcMsg

The value of sysExcMsg should be initialized at compile time to EXC_MSG_ADRS, which is set to a default value in configAll.h.

char *sysExcMsg

When there is a serious or catastrophic problem during the boot process, the system puts error messages at the location specified by sysExcMsg. This memory can then be examined from an external agent such as the bootrom image, or with an on-chip debugger.

sysFlags

The sysFlags variable holds the boot flags, which control factors such as whether to perform an autoboot or whether to use TFTP as the boot device.

int sysFlags; /* boot flags */

This variable is used extensively by bootrom. The value of sysFlags is initialized in usrConfig.c.

hcfDeviceList[]

The hcfDeviceList[] array holds information about static hardware configuration. Each record contains the name of the device, a counter for the device (in other words, a unit number), the type of bus on which the device resides, which is typically PLB (using the preprocessor macro VXB_BUSID_PLB), a pointer to another table containing configuration data specific to the device, and the number of entries in the configuration data table.

hcfDeviceNum

The hcfDeviceNum contains a count of the number of devices represented in hcfDeviceList[].
devUnitCfgData[ ]

The `devUnitCfgData[ ]` tables hold configuration information specific to a single device. These data are kept in records containing the following three fields:

- the name of the configuration datum
- the data type of the configuration datum (which can be integer, string, or address/pointer)
- the value

### 2.3.4 Required Routines

The following routines must be present in your BSP. Failure to provide any of these routines results in unresolved references at link time.

- `sysBspRev( )` returns the BSP version and revision number
- `sysClkConnect( )` connects a routine to the system clock interrupt
- `sysClkDisable( )` turns off system clock interrupts
- `sysClkEnable( )` turns on system clock interrupts
- `sysClkInt( )` handles system clock interrupts
- `sysClkRateGet( )` gets the system clock rate
- `sysClkRateSet( )` sets the system clock rate
- `sysHwInit( )` initializes the system hardware to a quiescent state
- `sysHwInit0( )` early BSP-specific hardware initialization (for some BSPs)
- `sysHwInit1( )` initializes all board hardware and virtual memory (64-bit only)
- `sysHwInit2( )` initializes and configures additional system hardware
- `sysMemDescGet( )` gets the physical address map of the hardware (64-bit only)
- `sysMemDescInit( )` initializes the hardware (64-bit only)
- `sysMemTop( )` gets the address of the top of logical memory (32-bit only)
- `sysModel( )` returns the model name of the target system board
- `sysNvRamGet( )` gets the contents of non-volatile RAM
- `sysNvRamSet( )` sets the contents of non-volatile RAM
- `sysSerialChanGet( )` gets the `SIO_CHAN` device associated with a serial channel
- `sysSerialHwInit( )` initializes the BSP serial device(s) to a quiescent state
- `sysSerialHwInit2( )` connects BSP serial device interrupts
- `sysToMonitor( )` transfers control to the ROM monitor

At the beginning of the development process, the required routines are very simple. In most cases, these routines are expanded over the course of the
development effort. The following pages describe the initial code for each required routine, and the file in which it is usually included.

sysBspRev( )

The routine `sysBspRev()` is identical in all BSPs. This routine can be taken directly from the reference or template BSP and does not require modification.

Timer Driver Routines

NOTE: The information in this section is applicable to legacy model driver maintenance only. Wind River strongly recommends that you use VxBus for new development and that you migrate any existing development work where feasible. For more information on VxBus model device drivers, see *VxBus Device Driver Developer’s Guide: Device Driver Fundamentals*.

Timer driver routines include the following:

- `sysClkConnect()`
- `sysClkDisable()`
- `sysClkEnable()`
- `sysClkInt()`
- `sysClkRateGet()`
- `sysClkRateSet()`

There are several timer drivers provided by Wind River. Files for these drivers are located in:

```
installDir/vxworks-6.x/target/src/drv/timer
```

If one of the existing timer drivers works for your target board, your `sysLib.c` file can simply include the appropriate file from the `timer` directory. For example, the MIPS R4000 processor includes an on-chip timer. This timer driver is provided in:

```
installDir/vxworks-6.x/target/src/drv/timer/mipsR4kTimer.c
```

If your board uses this processor, your `sysLib.c` file can include the driver as follows:

```
#include "timer/mipsR4kTimer.c" /* system timer */
```

In some cases, you may need to include both a header file from:

```
installDir/vxworks-6.x/target/h/drv/timer
```

And a source file from:

```
installDir/vxworks-6.x/target/src/drv/timer
```

Check the following directory to see whether a header file for your driver is available:

```
installDir/vxworks-6.x/target/h/drv/timer
```

If a standard timer driver is not provided for your target hardware, you may be able to copy an existing timer driver file to the BSP directory and make minor modifications. If you do this, it is wise to change the file name and routine names to avoid confusion. For example, some BSPs for ARM processors started with the
ambaTimer.c file and renamed it as the ixm1200Timer.c file. Be sure to document the name of the original driver in the source file.

In rare situations, you may need to create a custom sysTimer.c file. If this is the case for your BSP, you can copy routine prototypes for the sysClk*() routines from any of the existing timer drivers in:

```
installDir/vxworks-6.x/target/src/drv/timer
```

sysMemDescInit() (64-bit only)

The sysMemDescInit() function performs any initialization that the BSP must perform before sysMemDescGet() is called.

```c
void sysMemDescInit (void)
```

The kernel calls sysMemDescInit() once during system startup.

```
 /*********************************************************************/
 /* sysMemDescInit - Perform initialization required by sysMemDescGet(). */
 /* */
 /* This routine is used during system boot to perform any initialization */
 /* required by the BSP so that sysMemDescGet() can return correct */
 /* information about the memory configuration. */
 /* */
 /* RETURNS: N/A. */
 */
void sysMemDescInit
{
 void
}
{ LOCAL BOOL memFirstTime = TRUE;

/*
 * The first time we're called, we patch the descriptors based
 * upon run-time tests of the various memory segments. The
 * details of how the memory sizing tests are performed is
 * outside the scope of this document.
 */

if (memFirstTime)
{
 size_t ramSize;
 PHYS_ADDR physicalAddrEnd;
#ifdef INCLUDE_CACHE_DMA32_LIB
 size_t dma32Size = DMA32_HEAP_SIZE;
#else
 size_t dma32Size = 0;
#endif /* INCLUDE_CACHE_DMA32_LIB */
#ifdef INCLUDE_USER_RESERVED_MEMORY
 size_t userSize = USER_RESERVED_MEM;
#else
 size_t userSize = 0;
#endif /* INCLUDE_USER_RESERVED_MEMORY */
#ifdef INCLUDE_EDR_PM
 size_t pmSize = PM_RESERVED_MEM;
#else
 size_t pmSize = 0;
#endif /* INCLUDE_EDR_PM */
#ifated(INCLUDE_SYMMETRIC_IO_MODE) /* defined(INCLUDE_SYMMETRIC_IO_MODE) */
size_t romSize = ROM_SIZE;
#else
 size_t romSize = 0;
#endif /* !defined(INCLUDE_SYMMETRIC_IO_MODE) */
#endif /* defined(INCLUDE_SYMMETRIC_IO_MODE) */
#if defined INCLUDE_BIOS_E820_MEM_AUTOSIZE
```

sysMemDescGet() (64-bit only)

The sysMemDescGet() function returns the physical address map of the hardware.

/*********************************************************/
* sysMemDescGet - Return memory descriptors describing memory layout.
* This routine is used during system boot to describe the memory layout to the kernel and included components. It supports memory autoconfiguration. It also supports carving memory from the first contiguous block of RAM for 32-bit DMA and persistent memory. This routine assumes that enough memory is provided in the 0th descriptor to support all of these segments.
* NOTE - To simplify the example, this routine presumes that LOCAL_MEM_AUTOSIZE is defined. A proper implementation would include appropriate ifdef's to exclude the autoconfig code when it is not needed.
/*********************************************************/
* RETURNS: OK, if a valid PHYS_MEM_DESC is copied to the caller. */

STATUS sysMemDescGet
{
    MEM_DESC_TYPE memDescType, /* type of memory being queried */
    int index, /* index of memory block for the type */
    PHYS_MEM_DESC *pMemDesc /* to return the description */
} {  
    PHYS_MEM_DESC *pDesc;
    int descCount;

    /* Provide the correct array and size parameters based upon the  
     * query type provided by the caller. */
    if (memDescType == MEM_DESC_RAM)
    {
        pDesc = sysPhysRamDesc;
        descCount = sysPhysRamDescCount;
    }
#else // INCLUDE_USER_RESERVED_MEMORY
    else if (memDescType == MEM_DESC_USER_RESERVED_RAM)
    {
        pDesc = sysUserMemDesc;
        descCount = sysUserMemDescCount;
    }
#endif /* INCLUDE_USER_RESERVED_MEMORY */
#else // INCLUDE_EDR_PM
    else if (memDescType == MEM_DESC_PM_RAM)
    {
        pDesc = sysPmMemDesc;
        descCount = sysPmMemDescCount;
    }
#endif /* INCLUDE_EDR_PM */
#else // INCLUDE_CACHE_DMA32_LIB
    else if (memDescType == MEM_DESC_DMA32_RAM)
    {
        pDesc = sysDma32MemDesc;
        descCount = sysDma32MemDescCount;
    }
#endif /* INCLUDE_CACHE_DMA32_LIB */
else
{
    return ERROR; /* no support for MEM_DESC_APPL_RAM in this BSP */
}

if (index >= descCount) /* no descriptor at requested index */
    return ERROR;

/*
 * copy the descriptor. Note that it's OK to return a descriptor that
 * has a length of 0. This allows the BSP to allocate "empty" entries
 * in the descriptor arrays, and fill them in at run-time without having
 * to alter the logic of this routine.
 */
*pMemDesc = pDesc[index];
return OK;
}

sysMemTop( ) (32-bit only)

The sysMemTop() routine is defined to be the address of the first unusable address at the top of memory. Technically, it is equal to sysPhysMemTop -
2 Overview of a BSP

2.3 Components of a BSP

USER_RESERVED_MEM - PM_RESERVED_MEM. VxWorks will not use any memory at or above this address.

During early BSP development, the sysMemTop() routine in sysLib.c simply returns a constant value based on LOCAL_MEM_LOCAL_ADRS, LOCAL_MEM_SIZE, PM_RESERVED_MEM (VxWorks 6.x only), and USER_RESERVED_MEM.

In most implementations, sysMemTop() calls sysPhysMemTop() to obtain the address at the top of physical memory (for more information, see sysPhysMemTop() (32-bit only), p.37). Later, during BSP cleanup, the sysPhysMemTop() routine can be enhanced to allow automatic memory sizing.

The following is a macro-controlled version of sysMemTop() that can be used by any BSP to support fixed memory size:

```c
char *sysMemTop()
{
    static char * memTop = NULL;
    if ( memTop == NULL )
    {
        memTop = sysPhysMemTop () - USER_RESERVED_MEM;
        #ifdef INCLUDE_EDR_PM
        /* account for ED&R persistent memory */
        memTop = memTop - PM_RESERVED_MEM;
        #endif
    }
    return memTop ;
}
```

sysPhysMemTop() (32-bit only)

The sysPhysMemTop() routine in sysLib.c returns the address of the top of physical memory. This routine is required for VxWorks 6.x BSPs and, although it is not strictly required for VxWorks 5.5 BSPs, it is included in most. This routine can be enhanced to allow automatic memory sizing.

```c
/***********************************************************************
* sysPhysMemTop - get the address of the top of physical memory
* This routine returns the address of the first missing byte of memory,
* which indicates the top of memory.
* Normally, the user specifies the amount of physical memory with the
```
macro LOCAL_MEM_SIZE in config.h. BSPs that support run-time memory sizing do so only if the macro LOCAL_MEM_AUTOSIZE is defined. If not defined, then LOCAL_MEM_SIZE is assumed to be, and must be, the true size of physical memory.

NOTE: Do not adjust LOCAL_MEM_SIZE to reserve memory for application use. See sysMemTop() for more information on reserving memory.

RETURNS: The address of the top of physical memory.

ERRNO

SEE ALSO: sysMemTop() 

char * sysPhysMemTop (void)
{
LOCAL char * physTop = NULL;

if (physTop == NULL)
{
   physTop = (char *)(LOCAL_MEM_LOCAL_ADRS + LOCAL_MEM_SIZE);
}

return (physTop) ;
} 

sysModel()

The sysModel() routine in sysLib.c returns a string pointing to the model number of the board. For simple BSPs, this routine returns a constant string. If the BSP supports multiple board models, sysModel() requires a method to distinguish between boards to determine which of the related strings to use. The following is an example:

/***************************************************************************/
/* sysModel - return the model name of the CPU board */
/* This routine returns the model name of the CPU board. */
/* RETURNS: A pointer to the string. */
/***************************************************************************/
char * sysModel (void)
{
   return (SYS_MODEL);
} 

NVRAM Routines

The sysNvRamSet() and sysNvRamGet() routines can be provided initially by nullNvRam.c, which is included from sysLib.c. Later, during BSP cleanup, the appropriate standard version should be used or a custom version should be created. Standard versions are available in:

installDir/vxworks-6.x/target/src/drv/mem
Hardware Initialization Routines

Hardware initialization routines include the following:

sysHwInit()  
sysHwInit0() (for some BSPs)  
sysHwInit1() (64-bit only)  
sysHwInit2()  
sysSerialHwInit()  
sysSerialHwInit2()  

You must create these custom routines. However, during initial development, empty stubs can be used.

When you are ready to add these routines, check the reference BSP to see how it handles devices that are common to your board. However, you must be sure to add code to quiesce all devices on your board, including those that are not present on the reference board. Otherwise, you may encounter problems once interrupts are enabled.

The cache initialization routines, including the routines to initialize L2 cache, should be called from sysHwInit(). The requirement is that these routines must be called before cacheEnable() during the boot process. For more information, see 3.3.5 Enabling Cache and MMU Devices, p.73.

sysSerialChanGet()  

The sysSerialChanGet() routine in sysSerial.c returns the per-port SIO_CHAN structure required for each serial I/O (SIO) device. If a fixed number of SIO devices are supported, this routine returns a pointer to a statically allocated structure. The following is a sample for a generic BSP supporting exactly two serial devices:

```c
LOCAL SIO_CHAN boardSccChan1;
LOCAL SIO_CHAN boardSccChan2;

/*******************************************************************************
* sysSerialChanGet - get the SIO_CHAN device associated with a serial channel
* This routine gets the SIO_CHAN device associated with a specified serial
* channel.
* RETURNS: A pointer to the SIO_CHAN structure for the channel, or ERROR
* if the channel is invalid.
*/
SIO_CHAN * sysSerialChanGet
{
    int channel /* serial channel */
{
    if (channel == 0)
        return ((SIO_CHAN *) &boardSccChan1);
    else if (channel == 1)
        return ((SIO_CHAN *) &boardSccChan2);
    else
        return ((SIO_CHAN *) ERROR);

```
The `sysToMonitor()` routine in `sysLib.c` is called to reboot the system when `ctrl+x` is pressed on the console keyboard or, on some systems, when certain processor errors occur at interrupt level. When `sysToMonitor()` hands control over to the boot monitor, the system must be set to a state where no interrupts occur; failure to do this causes the reboot to fail. The following template can be used as a starting point. However, the **TO BE DONE** sections must be replaced by code that performs the stated operations.

```c
STATUS sysToMonitor
{
    int startType /* parameter passed to ROM to tell it how to boot */
{
    intCpuLock();
    // Warm reboot address */
    // NOTE: the address to use here is processor-dependent */
    FUNCTPTR pRom = (FUNCPTR) (ROM_TEXT_ADRS + some offset);
    // lock interrupts */
    intCpuLock();
    // disable processor-level cache */
    cacheDisable(INSTRUCTION_CACHE);
    cacheDisable(DATA_CACHE);
    // TO BE DONE */
    // disable aux clock, if one is provided by the BSP */
    // TO BE DONE */
    // disable local devices on the board, e.g. serial devices */
    // TO BE DONE */
    // reset the serial device */
    sysSerialReset();
    // set processor state to reasonable value */
    // TO BE DONE */
    // jump to bootrom entry point */
    (*pRom) (startType);
    // in case we ever continue from ROM monitor */
    return (OK);
}
```

A template for this routine is also available in the template BSP for your architecture, and examples are available in the reference BSPs included with your installation. However, you are encouraged to review the above template, as functionality in the template and reference BSPs may be incomplete.
2.3.5 Required Macros

BSP macros are defined in either `Makefile` or `config.h`, with some macros defined in both files. The following macros are required for all BSPs unless otherwise specified:

**CPU** (defined in `Makefile`)
The target CPU, which must be the same as for the reference BSP.

**CPU_VARIANT**
The target CPU variant (defined in `Makefile`).
This macro is not required for all architectures. For more information, see the *VxWorks Architecture Supplement* for your target processor.

**TOOL** (defined in `Makefile`)
The host tool chain (for example, `diab`), which must be the same as in the reference BSP. Most BSPs support both the Wind River Compiler (`diab`) and the Wind River GNU Compiler (`gnu`). IA-32 and Intel 64 BSPs also support the Intel C/C++ compiler (`icc`).

**TGT_DIR** (defined in `Makefile`)
The path to the target directory. The default is:

```
installDir/vxworks-6.x/target
```

**TARGET_DIR** (defined in `Makefile`)
The BSP directory name.

**VENDOR** (defined in `Makefile`)
The name of the target board manufacturer.

**BOARD** (defined in `Makefile`)
The target board name.

**ROM_TEXT_ADRS** (defined in `Makefile` and `config.h`)
The boot ROM entry address in hexadecimal notation. For most target systems, this is set to the beginning of the flash address area. However, there may be some hardware configurations that use an area at the start of flash for the reset vector. In this case, the address is offset accordingly.

The offset is typically architecture-dependent. Therefore, the low-order bytes of this macro can be copied from a reference BSP.

**NOTE:** See the *VxWorks Architecture Supplement* and the `target.ref` included with your reference BSP for dependencies and limitations.

**ROM_LINK_ADRS** (optional, defined in `Makefile` and `config.h`)
If used, this macro specifies the boot ROM link address in hexadecimal form. For most boards, it is set to the beginning of the flash address area. If this address is present, the linker uses it to link the boot ROM image. Otherwise, `ROM_TEXT_ADRS` is used as the link address.

**ROM_WARM_ADRS**
(optional, defined in both `Makefile` and `config.h`, or in either)
The boot ROM warm boot entry address in hexadecimal form. This is usually a fixed offset beyond the cold boot entry point, `ROM_TEXT_ADRS`. The offset is architecture-dependent, and can be obtained from the reference BSP or from
the bcopyLong() or copyLong() call in bootInit.c. The bootInit.c file is located in:

    installDir/vxworks-6.x/target/config/all

If ROM_WARM_ADRS is defined, the code in sysToMonitor() does an explicit jump to ROM_WARM_ADRS when a switch to the hardware ROM code is desired.

ROM_SIZE (defined in Makefile and config.h)
The flash area size in hexadecimal notation.

RAM_LOW_ADRS and RAM_HIGH_ADRS (defined in Makefile and config.h)
RAM_LOW_ADRS is the address at which the VxWorks image is loaded.
RAM_HIGH_ADRS is the destination address used when copying the boot ROM image to RAM.

NOTE: RAM_LOW_ADRS and RAM_HIGH_ADRS are absolute addresses, typically chosen to be at an architecture-specific offset from the start of DRAM. For information on the normal VxWorks memory layout, see the memory layout diagram in the appropriate chapter of the VxWorks Architecture Supplement.

HEX_FLAGS (defined in Makefile)
Defines the architecture-specific flags for the objcopy utility that generates S-record files.

MACH_EXTRA (defined in Makefile)
Defines any extra machine-dependent files. In early development, make this an empty declaration.

BSP_VER_1_1 (defined in config.h)(deprecated in this release)
BSP_VER_1_2 (defined in config.h) (deprecated in this release)
BSP_VERSION (defined in config.h)
BSP_REV (defined in config.h)
These macros indicate the version and revision numbers for the BSP.

For VxWorks 6.9 BSPs and later, the value of BSP_VERSION is coded to the VxWorks two-digit version number and BSP_REV reflects the BSP revision. The initial revision of the BSP is set to 0 and is incremented for each revision of the BSP for a given BSP_VERSION; the BSP_REV resets to 0 when BSP_VERSION is incremented for the next release of VxWorks.

For example, BSP_VERSION 6.9 BSP_REV /5 would change to BSP_VERSION 6.x BSP_REV / 0 if re-released for VxWorks 6.x (a later version of VxWorks).

These macros must be defined before the configAll.h file is included. Note that BSP_VERSION and BSP_REV should contain strings.

NOTE: If your application contains hard coded tests of the BSP_VERSION macro for values such as 1.2 or 2.0, these tests will fail when they are performed against a BSP that implements the new version numbering convention.

The following is an example containing values for a typical new (revision 0) BSP:

    /* BSP version/revision identification, before configAll.h */

    #define BSP_VERSION        "6.9"       /* vxWorks 6.9 */
#define BSP_REV            "/0"        /* 0 for first revision */
#include <configAll.h>

**NOTE:** The BSP version and revision must be included in the BSP README file as well as in the config.h file. The description and definition in these two files must agree.

**LOCAL_MEM_LOCAL_ADRS** (defined in config.h)
The start of the on-board memory area.

**LOCAL_MEM_AUTOSIZE** or **LOCAL_MEM_SIZE** (defined in config.h)
Only one of these two macros is defined for a given BSP.
- If defined, **LOCAL_MEM_SIZE** defines the fixed (static) memory size.
- If defined, **LOCAL_MEM_AUTOSIZE** indicates that the BSP determines the actual memory size when booting, and that no static size is assumed.

**INCLUDE_BIOS_E820_MEM_AUTOSIZE**
Detects usable memory. (Intel architecture only.)

**ROM_BASE_ADRS** (defined in config.h)
The ROM start address in hexadecimal form. **ROM_BASE_ADRS** can be used anywhere in the BSP to get the base address of ROM; it can be replaced with other definitions, such as **FLASH_BASE_ADRS**.

**NOTE:** See the VxWorks Architecture Supplement and the target.ref included with your reference BSP for dependencies and limitations.

**ROM_WARM_ADRS** (defined in config.h)
The warm boot entry address. This is usually defined as an offset from **ROM_TEXT_ADRS**.

**USER_RESERVED_MEM** (defined in config.h)
Size (in bytes) of the memory region reserved for application use. For more information, see Wind River Technical Note #41. The default value is 0 for all BSPs.

**VMA_START** (optional, defined in Makefile)
This macro is used for creating hex files (for example, bootrom.hex). The default value is 0x00000000. If defined, it should be a hex address with a leading 0x. In some BSPs, this macro is defined as follows:

\[ VMA\_START = 0x$(ROM\_TEXT\_ADRS) \]

**NUM_TTY** (defined in config.h)
The number of TTY ports to be configured at boot time. The default value is 2.

**NOTE:** Unless otherwise specified, the hexadecimal addresses in the macros in Makefile must *not* include a leading 0x. The copies in config.h must include the leading 0x.

Some additional architecture-specific macros may also be required in the makefile. For example, the i960 architecture must be instructed as to where to link the initial boot record. For architecture-specific information, see the appropriate chapter of the VxWorks Architecture Supplement.
2.3.6 Optional Routines

Most BSPs provide routines beyond those that are required. Typically, an auxiliary clock and a serial port are provided. In addition, some kind of Ethernet interface is supported. Other hardware is often made available as well. Descriptions of the routines required to support these hardware features are included in the 3.3.9 Providing Additional Optional Device Support, p.96, and in the VxBus Device Driver Developer’s Guide.

2.3.7 Optional Macros

The following macro is optional for all BSPs:

**BOOT_EXTRA** (defined in Makefile)

Defines extra object modules to be added to ROM compressed images (*vxWorks.*st_rom, bootapp, bootrom). “Extra modules” are any modules besides **romInit.o** in the BSP directory which are needed early in the initialization of the board (**romInit()**).

Note that these modules are not linked with the compressed VxWorks image. They are linked with the startup code that initializes the board before the compressed VxWorks image is unpacked and copied to RAM. If the same module is referenced by other BSP code that is part of the compressed VxWorks payload image, the module must be added to **MACH_EXTRA** as well (see 2.3.5 Required Macros, p.41).

**NOTE:** There are other optional macros not covered here that are used in VxWorks BSPs. For more information, see the reference BSPs provided in your installation.

**NOTE:** See the target.ref for optional macros unique to your reference BSP.

2.3.8 Hardware Considerations

There are a number of hardware issues that relate to BSP development. For the most part, these issues are more relevant during board design than during BSP development. However, some BSP issues must be addressed. If the design of your target hardware is not already in place, it is worthwhile to ensure that the hardware is designed in a way that simplifies BSP development.

When developing a BSP, you must know how the boot type is passed to **romStart()**. This depends on the processor architecture, but is generally passed either in a register or on the stack. If the boot type is passed in a register, the register is determined by the C function calling convention for your processor. If it is passed on the stack, the stack is usually placed such that it begins before the text section and grows in the opposite direction.

A bank of LED indicators that can be used without any hardware initialization is a good feature to have in your target hardware. The first stage in BSP development should always be to write, test, and debug code that can set, clear, and modify the LEDs. This is the case even if an on-chip debugging (OCD) device, or another type of hardware debugger, is available.
Most boards use dynamic memory for the main bank of RAM, rather than static memory. When dynamic memory is used, a memory controller is required in order to provide memory refresh cycles. The memory controller must be initialized very early during system initialization, usually before any C code is run. For this reason, it is customary to write the memory controller initialization code in assembly, to be linked with the boot loader image. For more information, see the memory controller section of the VxBus Device Driver Developer’s Guide.

2.4 The Development Environment

The development environment consists of a mechanism to build an object module, a method to download the image to the target, and a mechanism for testing the downloaded code. It includes several items: a hardware debugger (if you use one), Workbench, a compiler and other development tools, and a means of downloading VxWorks images for debugging.

NOTE: A hardware debugger, such as an on-chip debugging (OCD) system, is a good investment and is generally considered a requirement for efficient BSP development.

When porting a BSP, it is usually best to start with a debugging method that includes some simple test code rather than a full-blown VxWorks image. For example, if your board has LEDs, the first code you write may be the code to turn the LEDs on and off. By starting with this simple kind of code, you can verify the development environment in addition to creating a routine that is useful later in the BSP development process.

The development environment also includes compilers, linkers, and OS library files. The BSP provides board-specific functionality, and the BSP routines often depend on OS routines that are not normally provided by Wind River in source form. Object modules for these routines are provided in a library file in the VxWorks installation. The old VxWorks library naming convention was that libraries were named lib<CPUTOOL>vx.a, where CPU is the CPU architecture of your board, and TOOL is the toolchain you are using to build your BSP. An example of an older library is libPPC860gnuvx.a.

Starting with VxWorks 5.5, the library files are split up into several different files. The lib<CPUTOOL>vx.a files are still present as of VxWorks 5.5, but they are stubs only. The new files are kept in subdirectories under the following directory:

\texttt{installDir/vxworks-6.x/target/lib}

The subdirectories are named as follows:

\texttt{installDir/vxworks-6.x/target/lib/ARCH/CPUTOOL}
\texttt{installDir/vxworks-6.x/target/lib/ARCH/CPUTOOL/common}

Where CPU and TOOL are as described above, and ARCH is the abbreviation for the processor architecture family. For example, PowerPC libraries for use with the Diab C/C++ Compiler toolchain with the 860 processor are located in the following directories:
Several library files are kept in these two directories; each library file contains object modules for individual components.

In VxWorks 6.x, processor variants have been eliminated. For example, all PowerPC processors are now indicated using PPC32. For more information, see the VxWorks Architecture Supplement: CPU and TOOL Make Variables.

2.4.1 BSP Debugging Methods

Part of setting up your development environment is choosing a BSP debugging method. This section provides an overview of the hardware debugging options available to a BSP developer. For more detailed information, see 5. Debugging Your BSP.

Primitive Tools

For most of the early work of porting a BSP, the system is not initialized enough to use a host-based debugger, a target-based debugger, logMsg(), or printf() to obtain debugging information. Therefore, some method of low-level debugging must be selected.

A BSP developer can begin porting a BSP with no more debugging information than what is provided by an LED that can be turned on or off under software control. Later in the development process, a polled-mode serial output routine, possibly called printf(), can be created to print complex diagnostic information to the serial console.

Another debugging method consists of writing information to non-volatile RAM (NVRAM). This information can be displayed using another computer at a later time and can be used to obtain information about the state of the target processor.

Both of the above methods are considered primitive. Other methods are generally available to make the port easier and faster.

Native Debug ROMs

In some cases, the board manufacturer provides flash software that can be used to help debug the VxWorks BSP. To be useful, a ROM monitor must have breakpoint support. It is also helpful to have a mechanism by which to download the image.

An example of a debug ROM is ppcbug, which is provided by Motorola.

ROM Emulator

A ROM emulator is a device that plugs into a ROM socket on your target system and emulates the behavior of a ROM part.

From the target system, this device functions as a ROM (with a possible difference in timing behavior). From the development host, a ROM emulator allows you to
see every transaction visible to the ROM; typically, either all accesses to memory or all accesses to ROM. At a minimum, this device allows you to see what instructions are being executed during the initial phases of the boot sequence.

A ROM emulator eliminates the need to program flash devices and may allow you to see the steps of program execution. However, it does not allow access to the processor registers. A ROM emulator can be used in conjunction with LED or `printf()` debugging (see Primitive Tools, p.46). Note that this combination is better than either method by itself.

### On-Chip Debugging (OCD) Devices

For BSP debugging, a hardware debugger is often a better solution than any of the basic tools presented above. Many modern processors provide a debug bus that allows access by external debugging hardware to useful processor information such as register values, interrupt mask, and other aspects of the processor’s internal state. One example of this type of interface is the JTAG port available on many ARM, Intel XScale, PowerPC, MIPS, and other processors. OCD devices, such as the Wind River ICE 2, can access this port. This access allows you to begin debugging much earlier in the BSP development process.

An OCD device provides access to the target processor’s registers, which allows you to start using the OCD before any VxWorks initialization code is written. In fact, this type of tool is often used to determine how the initialization code must set specific processor configuration registers.

During development, the OCD is used like a standard software debugger—that is, it is used to trace and modify program execution, examine and sometimes modify register contents, set breakpoints, and so forth. In addition, the OCD can be used to program flash devices, which saves time each time a new version of the BSP under development must be tested.

Additional development time can be saved when both the target board development team and the BSP development team have access to an OCD device. In this case, each team has access to the same information about what is happening on the target board. This greatly reduces uncertainty about the cause of the problem and whether it is attributable to hardware or software.

**NOTE:** Some OCD-based debuggers assert the non-maskable interrupt (NMI) line to obtain access to the processor. This can cause the processor to fail in an unpredictable manner, seemingly independent of the code that you are trying to debug. Check with your OCD device vendor to see if this is the case for your debugger. For more information on non-maskable interrupts, see 2.5.5 Using Non-Maskable Interrupts, p.53.

### Logic Analyzer

A logic analyzer is a device regularly used during hardware development. If the BSP development and hardware development are closely coupled, the logic analyzer used to develop the hardware can also be used to assist in writing the BSP. However, it is rarely cost-effective to purchase an additional logic analyzer purely for BSP development purposes.
If a logic analyzer is used for BSP development, the processor must be configured in such a way that it always puts address requests to the external bus. This configuration may not be required by the hardware developers.

In some situations, an oscilloscope may also be useful for BSP development, especially when the BSP and hardware are developed in parallel.

**In-Circuit Emulator**

An in-circuit emulator replaces the target processor with an external device that emulates the processor. In-circuit emulators are not available for all processor and architecture types. If an in-circuit emulator is available for your target processor, it can be used for debugging during BSP development.

An in-circuit emulator provides all the advantages of an on-chip debugging (OCD) device and may include additional features and abilities beyond the OCD capabilities.

The disadvantages of an in-circuit emulator include cost and limited processor support. However, for processors with no OCD interface capability, an in-circuit emulator is sometimes the only viable solution for finding and fixing certain problems.

Like a logic analyzer, an in-circuit emulator is generally not cost-effective for BSP development alone.

### 2.4.2 WDB Debugging Interface

When no hardware debugger is available, try to minimize the amount of time during which you do not have access to the Wind River development tools, particularly the Wind debug target agent (WDB agent). Because the WDB agent is linked to the kernel, it can share initialization code with the kernel. Thus, after the initialization code has run, you can start either the WDB agent, the VxWorks kernel, or both.

The VxWorks WDB agent provides a powerful debugging environment and, if an OCD device is not available, it is possible to use the WDB agent for BSP debugging.

> **NOTE:** Wind River BSP developers rarely use the WDB agent for debugging. However, when no OCD device, ROM emulator, or in-circuit emulator is available, the WDB agent is a good choice.

> **NOTE:** By default, WDB is not enabled for any VxWorks image projects (VIPs). It can be enabled when VxWorks is configured, with the boot loader, or programmatically. See *VxWorks Kernel Programmer’s Guide: WDB Target Agent* for information about how to enable the WDB target agent.

Using the WDB agent, you can debug the VxWorks image to which it is linked. The target agent’s linked-in approach has several advantages over the older approach of using a ROM monitor (see *Native Debug ROMs*, p.46). For example:

- There is only one initialization code module to write. In a traditional ROM-monitor approach, you must write two initialization code modules: one for the monitor and one for the OS. In addition, the traditional approach
requires non-standard modifications to resolve contention issues over MMU, vector table, and device initialization.

- The code size is smaller because VxWorks and the target agent can share generic library routines such as `memcpy()`.
- Traditional ROM monitors debug only in system mode. The entire OS is debugged as a single thread. The WDB agent provides, in addition to system mode, a fully VxWorks-aware tasking mode. This mode allows debugging selected parts of the OS (such as individual tasks), without affecting the rest of the system.
- Because the Wind River development tools let you download and execute code dynamically, you can download extensions and use the WDB agent to debug the extensions. The downloadable extensions include application code, new drivers, extra hardware initialization code, and so on.

How you download the WDB agent and the VxWorks kernel depends on your phase of development. When writing and debugging the board initialization code, you must create your own download path. This is an advantage over the traditional ROM-monitor approach, in which you must create the download path for porting the monitor itself.

After you have the board initialization code working, how you proceed depends on the speed of your download path. If you have a fast download path, continue to use it for further kernel development. This is an advantage over the traditional ROM monitor approach that often forces you to use a serial-line download path. If you have a slow download path, you can burn the kernel into ROM, as well as the agent and as much generic VxWorks code as fits.

### 2.4.3 Workbench Libraries and Tools

When porting a VxWorks BSP, it is necessary to have the VxWorks libraries that come with a Wind River Workbench installation. The Workbench installation also provides header files, a compiler and other tools, and the framework for BSP development. Also, a BSP is not considered complete until bootable projects are tested and available.

### 2.4.4 Compiler and Tool Choice

When porting a BSP, there are often several choices of compilers available. At least one compiler is always available with Workbench, and that compiler is usually the best choice. However, there are often other choices available, either from Wind River or from third parties.

Whichever compiler is used, it must satisfy the following requirements:

- It must be available to all users of the BSP.
- It must be able to read and understand the VxWorks library format and object module format (OMF).
- It must be able to generate code that works with the debugger, if any, that is to be used for BSP development. It should also work with the debuggers that will be used during application development.
It must not generate code that silently performs certain activities such as memory allocation. For example, no BSP code or driver code should be written using C++ because the C++ compiler does silent memory allocation to allocate constructors, as well as other tasks that are not available until after the OS is completely booted or that cannot be done during ISR execution.

Also keep in mind that the BSP is recompiled often, and that the BSP may be compiled with a different compiler than the one that is used for development. For portability, it is wise to insure that the compiler is used with the most stringent options available, such as the `ansi -pedantic` flags of the GNU compiler.

To find out what compiler and linker flags are required, go to your reference BSP and build one of the standard VxWorks image types. Typically, this is `vxWorks` (a RAM image) or `vxWorks.res_rom_nosym` (a flash image).

There are also ancillary tools, such as an archiver, disassembler, linker, binary file dump program, and so on. These tools should be available as part of the package that the compiler comes with or as part of the debugger. If the package that is chosen is missing a tool, a compatible version of that tool may be available from some other source. For example, if a program to display object module symbols is missing, you can use the `nmarch` program (`nmppc` or `nmarm`, for example) that is part of Workbench.

### 2.4.5 Download Path

The following are some of the more common techniques for downloading code to the target during BSP development:

- Use the download protocol supplied in the board vendor’s debug ROMs. The drawback of this approach is that downloading is often slow. The advantage is that it is easy to set up.

- Program the image into ROM. This allows code to be put onto the target, but does not allow any mechanism for debugging other than the LED or `printf()` routines (see [Primitive Tools](#), p.46). However, debugging the LED or `printf()` routines is extremely difficult if this is the download mechanism.

- Use a ROM emulator (such as NetROM from AMC). The drawback of this approach is that it can take time for you to learn how to use the tool. The advantages include fast download times, portability to most boards, and a communication protocol that lets debugging messages pass from the target to the host through the ROM socket. For more information on ROM emulators, see [ROM Emulator](#), p.46).

- Use an OCD device (see [On-Chip Debugging (OCD) Devices](#), p.47). This allows you to single-step through much of the initialization code.

- Use an ICE (see [In-Circuit Emulator](#), p.48). The main drawbacks of this approach include a high procurement cost and a lack of availability for all processors. However, it does let you single-step through the initialization code and can significantly improve a product’s time to market.

- Use the WDB agent (see [2.4.2 WDB Debugging Interface](#), p.48). Once the WDB agent is available, many parts of BSP initialization can be downloaded and tested while running. In addition to a fast, efficient download path, this gives you access to the Wind River tools, including the full capabilities of the debugger.
After you have downloaded code to the target, examine memory or ROM to make sure your code is loaded in the right place. The GNU tools `nmarch` and `objdumparch`, supplied by Wind River, can be used on your compiled images to see what should be in the target memory. When reviewing this information, give special attention to the start addresses of the text and data segments.

## 2.5 Common Problems

Most of the problems listed in this section are the result of doing the right thing in the wrong place or at the wrong time. Context is important. To avoid introducing problems into your BSP, examine your reference BSP carefully before starting your development and ensure that the reference BSP developer did not make the following mistakes.

### 2.5.1 Failing to Include LOCAL_MEM_LOCAL_ADRS

Many BSP writers assume that `LOCAL_MEM_LOCAL_ADRS` is zero and fail to include it in macros that must be offset by the start of memory.

For example, the routine `sysPhysMemTop()` indicates the highest addressable memory present on the system. If autosizing is not used, such as during early parts of BSP development, this routine can be written to return a constant value. This value should be relative to `LOCAL_MEM_LOCAL_ADRS`, for example:

```c
char * sysPhysMemTop()
{
    return ((char *)(LOCAL_MEM_LOCAL_ADRS + LOCAL_MEM_SIZE));
}
```

Most BSP developers do not change the value of `LOCAL_MEM_LOCAL_ADRS`, therefore this problem tends to be copied and replicated throughout development projects.

### 2.5.2 Providing Too Much Device Initialization in romInit.s

Many BSP writers add too much device initialization code to `romInit.s`. Treat the initialization in `romInit.s` as a preliminary step only. All real device initialization should be handled in `sysHwInit()` in `sysLib.c`.

Many embedded devices have a requirement for a fast boot time. Developers working on such projects must ensure that the system boots quickly. This is usually achieved by removing initialization routines entirely, or by delaying the initialization until later in the boot process. Typically, efforts to minimize boot time do not require changes to `romInit.s`. However, if `romInit.s` includes too much initialization code, modification may be necessary.
2.5.3 Providing Insufficient Initialization in sysALib.s

Many BSP writers assume that any initialization done in `romInit.s` need only be done once. However, all initialization done by `romInit.s` should be repeated in the routine `sysInit()` in `sysALib.s`. If the initialization is not repeated, the BSP user must rebuild boot ROMs for simple configuration changes in their VxWorks image.

There are some kinds of initialization that do not need to be performed in `sysALib.s`, though these exceptions are limited. In most cases, the exceptions are limited to one of two situations. One kind of initialization that does not need to be done in `sysALib.s` is configuration of processor or board registers that can only be written once after initial power-on and that retain their original value regardless of subsequent attempts to change them. The other situation involves initialization that must be done in order for the code in `sysALib.s` to execute; for example, certain parts of memory controller initialization must be performed first.

2.5.4 Including Confusing Configuration Options

In the `config.h` file, the user should be presented with clear choices for configuring the BSP. Material that cannot be configured by the user should not be in `config.h`, but should be placed in `bspname.h` instead.

In addition, the user should not have to compute values to be typed into `config.h`. Instead, there should be macros for manipulation of the relevant data. For example, if a register must be loaded with the high 12 bits of an address, the user should only be required to enter the full address. Even better would be for the user to enter the name of a symbol or macro that refers to the address. The code, either the compiler’s preprocessor or the configuration code that is part of the BSP, should do the computation of the value to load in the register.

The following examples illustrate correct configuration options:

```c
/* Division Factor of BRGCLK shift, specified in hardware docs */
#define SCCR_DFBRG_SHIFT 0x000c
#define BRGCLK_DIV_FACTOR 4
/* set the BRGCLK division factor */
* SCCR(immrVal) = (* SCCR(immrVal) & ~SCCR_DFBRG_MSK) |
  (BRGCLK_DIV_FACTOR << SCCR_DFBRG_SHIFT);
```

Correct example 2:

```c
#define BRGCLK_FREQ    (SPLL_FREQ / ( 1 << (2 * BRGCLK_DIV_FACTOR)))
ppc860Chan [i].clockRate = BRGCLK_FREQ;
```

Correct example 3:

```c
lis r6, HIADJ(ROM_TEXT_ADRS) /* load r6 with the address */
addi r6, r6, LO(ROM_TEXT_ADRS) /* of ROM_TEXT_ADRS */
```

The following examples show incorrect configuration options:

```c
/* WRONG: DO NOT use a magic number in assembly source code! */
* SCCR(immrVal) = (* SCCR(immrVal) & ~SCCR_DFBRG_MSK) | 0x4000);
```

Incorrect example 2:

```c
/* WRONG: DO NOT use a magic number in C source code! */
ppc860Chan [i].clockRate = 0x16e3600;
```

Incorrect example 3:

```c
/* WRONG: DO NOT use a magic number in assembly source! */
lis r6, 0 /* load r6 with the address */
```
addi r6, r6, 0 /* of ROM_TEXT_ADRS */

Incorrect example 4:
/* WRONG: DO NOT define a macro as a magic number! */
#define BRGCLK_RATE 0x16e3600
ppc860Chan [i].clockRate = BRGCLK_RATE;

Incorrect example 5:
/* WRONG: DO NOT assume the high-order bits are zero! */
lis r6, 0 /* load r6 with the address */
addi r6, r6, LO(ROM_TEXT_ADRS) /* of ROM_TEXT_ADRS */

2.5.5 Using Non-Maskable Interrupts

Because the kernel lacks interrupts to protect its data structures while they are being updated, using non-maskable interrupts (NMI) should be avoided. If an NMI interrupt service routine (ISR) makes a call to VxWorks that results in a kernel object being changed, protection is lost and undesirable behavior can be expected. For more information on ISRs at high interrupt levels, see the VxWorks Kernel Programmer’s Guide.

Also, note that a VxWorks routine marked as interrupt safe does not mean it is NMI interrupt safe. On the contrary, many routines marked as interrupt safe are actually unsafe for NMI.

On some architectures, the use of NMI cannot be tolerated for anything other than rebooting the target or permanently halting the target in order to retrieve the contents of memory. For example, on PowerPC, if an NMI occurs while certain sections of code are being executed, information about the return address is irrevocably lost. In this case, the processor is never able to return from the NMI ISR to that previously running code, regardless of the action taken by the NMI ISR.
Porting a BSP to Custom Hardware

3.1 Introduction

This chapter describes the BSP porting process in detail. Before beginning this process, be sure your development environment is set up and properly configured. You should also have a basic understanding of how VxWorks is initialized.

Creating a new BSP using the Wind River tools requires that you handle the development in graduated steps, each building on the previous, as follows:

1. Set up your development environment. For more information, see 2.4 The Development Environment, p. 46.

2. Write the BSP pre-kernel-initialization code. This includes board initialization, memory initialization, and an LED driver (if applicable).

3. Start a minimal VxWorks kernel and add the basic drivers for an interrupt controller, timers, and serial devices.

4. Start the target agent and connect the Wind River development tools.

5. Complete the BSP. This can include adding support for buses, networking, boot loaders, SCSI, caches, MMU initialization, and direct memory access (DMA).

6. Generate a default project for use with the project facility.

The goal of this procedure is not only to create a new BSP, but also to minimize the time during which you do not have access to the Wind River development tools—in particular, the Wind Debug target agent (WDB agent). Because the WDB agent
is linked to the kernel, it can share initialization code with the kernel. Thus, after
the initialization code has run, you can start either the WDB agent, the VxWorks
kernel, or both.

NOTE: The name of the BSP must be consistent. The name of the BSP in xxBsp.cdf
in the BSP directory must be same in the directory where the BSP resides.

Assumed you have a BSP named "myBSP" stored in /target/config/myBSP, it must
be given in xxBsp.cdf as:

```
Bsp myBSP {
  NAME board support package
  CPU myCPU
  REQUIRES ...
  ...
}
```

3.2 Getting a Default Kernel Running

During the initial phases of BSP development, command-line tools are used to
build the code. No Wind River tools are available. A good early step in the
development process is to write simple code that helps to verify that the CPU is
working, making later debugging easier.

Your initial strategy might be to create a BSP directory and add the minimum
required files. This includes the files specified in 2.3.1 Source and Include Files, p.19.
You must also ensure that there are stubs for each of the routines described in
2.3.4 Required Routines, p.33.

Next, build your VxWorks image and verify that it contains the expected code at
the proper locations. For more information on what can be done at this stage of the
process, see 5. Debugging Your BSP. All of these steps can be done before hardware
is available.

NOTE: Code built for variants of VxWorks or for different releases of VxWorks is
not binary compatible between variants or releases. Code must be built specifically
for uniprocessor (UP) VxWorks, VxWorks SMP, 32-bit VxWorks, 64-bit VxWorks,
or for a VxWorks system based on variant libraries produced with a VxWorks
source build (VSB) project—or for any supported combination of these variants.
The only exception to this rule is that RTP application executables can be run on
both UP VxWorks and VxWorks SMP (when all else is the same).

3.2.1 Initializing the Board

The first task that the BSP must accomplish is to initialize the board’s registers to
the point that the processor executes instructions correctly. Typically, some of this
initialization is handled by the target hardware at power-up time. For example, as
specified in 2.2.3 Architecture Considerations, p.13, MIPS processors use a modeIn
pin to set the initial values of some processor registers. Other registers must be set
in `romInit()`. The selection of processor registers that must be set early in development depends on the processor family and your specific processor.

In addition to the minimum set of required processor registers, some board registers must be initialized. The selection of board control registers that must be set at this point is determined by the board design.

In most cases, the processor-specific requirements are to:

- Mask processor interrupts.
- Set the initial stack pointer to `STACK_ADRS` (defined in `configAll.h`).
- Disable processor caches.

### 3.2.2 Initializing Memory

Typically, the requirements for DRAM initialization include:

- wait states
- refresh rate
- chip-selects
- disabling secondary (L2) caches (if needed)

The code in `romInit.s` must initialize the processor and board control registers (BCRs), as well as the memory controller. Processor initialization on your board should be identical to the processor initialization in the reference BSP. You must modify the BCR initialization code provided by the reference BSP.

If your board does not use the same memory controller as the reference BSP, you may also need to write the memory controller initialization code. This code is often available from the memory controller vendor as assembly source code and can often be used without modification.

It is possible to use an OCD device to verify the values of the memory controller registers. Using the debugger, set the appropriate registers to the desired values and verify that RAM is readable and writable. Then use the working values to create the memory initialization routine `sysMemInit()`. For more information on OCD devices, see *On-Chip Debugging (OCD) Devices*, p.48.

Wind River provides a DDR controller driver, which supports the Freescale QorIQ series of processors. This driver can be used to initialize the DDR controller and system memory with the Serial Presence Detect (SPD) mode supported.

The DDR controller driver:

1. probes all valid DIMMs installed in DIMM sockets
2. reads out the SPD data from the I2C EEPROM on the DIMMs
3. works out the registers’ initial values
4. initializes the DDR controllers with these initial values
5. sets the LAWBAR registers for DDR controllers

Only the DDR3 DIMMs are supported. If the DDR3 DIMMs are used as the system memory devices on your board, refer to the `fslDdrCntrl` Freescale DDR controller library in the *VxWorks Driver API Reference Manual* for more information.
If other types of DDR controllers are used on your target, and a DDR controller driver with SPD support is desired, you can use the public utility routine library, spdLib, to port DDR controller driver. For more information, refer to the APIs in the spdLib and fslDdrCtlr libraries.

### 3.2.3 Using Debug Routines in the Initialization Code

It is usually helpful to validate the development environment by writing code to turn the board LEDs on and off. This allows you to include debug code that can provide information about the state of the board initialization.

On a well-designed board, the LEDs are addressable without having too many processor or board registers configured and without very much additional bus configuration. The LED code should be simple. The following is an example of C source code for a simple LED system (you may also use LED code from your reference BSP):

```c
/* sysLed.c - Wind River 8260 User LED driver */
/* Copyright 1984-2003 Wind River Systems, Inc. */
/*
 * modification history
 * --------------------
 * 01a,30jul01,g_h created from T2 SBC8260/sysLib.c */
/*
 * DESCRIPTION
 * This module contains the LED driver.

INCLUDES: sysLed.h
*/

/* includes */
#include "vxWorks.h"
#include "wrSbc8260.h"
#include "sysLed.h"

#elif INCLUDE_SYSLED
/* locals */
LOCAL UINT8 sysLed;

/*****************************/
/* sysLedInit - Initialize LEDs */
/* This routine initializes the LED variable to zero and clears all LEDs. */
/* RETURNS: N/A */
/* SEE ALSO: sysLedOn(), sysLedOff(), sysLedControl(). */
*/

void sysLedInit
{
    void
    }
    sysLed = 0;

    /*
    * Write to LED.
    */
    BSCR_LEDREGISTER = sysLed;
```
3 Porting a BSP to Custom Hardware
3.2 Getting a Default Kernel Running

static int sysLed = 0x00;

/******************************************************************************
* sysLedOff - Turn selected LED off
* This routine sets the selected LED to off.
* RETURNS: N/A
* SEE ALSO: sysLedInit(), sysLedOff(), sysLedControl().
***************************************************************************/
void sysLedOff
{
    UINT8 led
    {
        sysLed &= ~led;
        /*
         * Write to LED.
         */
        BSCR_LED_REGISTER = sysLed;
    }

/******************************************************************************
* sysLedOn - Turn selected LED on
* This routine sets the selected LED to on.
* RETURNS: N/A
* SEE ALSO: sysLedInit(), sysLedOn(), sysLedControl().
***************************************************************************/
void sysLedOn
{
    UINT8 led
    {
        sysLed |= led;
        /*
         * Write to LED.
         */
        BSCR_LED_REGISTER = sysLed;
    }

/******************************************************************************
* sysLedControl - Turn selected LED(s) on or off
* This routine sets the selected LED on or off.
* RETURNS: N/A
* SEE ALSO: sysLedInit(), sysLedOff(), sysLedControl().
***************************************************************************/
void sysLedControl
{
    int ledOn, led
    {
        /*
         * Check led state.
         */
        if (ledOn)
            sysLed |= led; /* Set LED on. */
        else
            sysLed &= ~led; /* Set LED off. */
/* Write to LED. */
SCR_LED_REGISTER = sysLed;
}
#endif /* INCLUDE_SYSLED */

If the LED manipulation routines are written before the environment is set up for C subroutine linkage, the routines must be written in standalone assembly language. In this case, the above sample code can be used as a starting design for your assembly code. Later on, during the cleanup phase of your BSP development, the LED routines can be rewritten in C.

In addition to the basic LED routines, a simple control loop is useful to flash the LEDs on and off, with a suitable delay between the flashes so that they can be seen easily.

Once the LED routines are written, your initialization code can include calls to turn on or off LEDs at specific points during initialization.

### 3.2.4 VxBus Initialization Sequence

VxBus compliant BSPs and drivers use a well defined initialization sequence. Drivers register with VxBus, and VxBus pairs devices with drivers to create instances. Each instance (device plus driver) goes through a three-pass initialization sequence. The following paragraphs describe the sequence that occurs for an instance that is discovered during the boot process. Device drivers can also be downloaded after VxWorks boots, in which case the same sequence is followed, but the operating system features available at each stage during initialization are different.

The first stage of instance initialization occurs early in system initialization, in `sysHwInit()`. At this time, no operating system features are available, so drivers are not able to allocate memory, connect interrupts, or use other operating system features.

The second stage of instance initialization occurs later in system initialization, in `sysHwInit2()`. The driver can now allocate memory, connect ISRs, and use other basic operating system features. Network and file system connections are not available at this time.

The third stage of instance initialization occurs asynchronously, in a separate task that is created late in `usrRoot()`.

Additional initialization can be done on a functionality-specific basis, such as connecting a network driver to the MUX. This is done by registering driver methods with VxBus.

For a more detailed description of VxBus initialization, see the *VxBus Device Driver Developer’s Guide*.

**NOTE:** VxBus device drivers register with VxBus in order to be initialized, this registration does not need to occur at boot time. One debug strategy for VxBus device drivers is to compile the driver into the system but not have the driver register with VxBus. Then, when the system has booted, you can call the registration routine. At this time, the driver is paired with any appropriate devices, and standard debug methods can be used for driver debugging.
### 3.2.5 Debugging the Initialization Code

The beginning portions of the ROM and RAM initialization sequences differ, but the sequences are otherwise the same. Details of what each BSP procedure must do are provided in 2.2 Boot Sequence, p.7. This section reviews the steps of the initialization sequence and supplies tips on what to check if a failure occurs at a particular step.

#### Initializing ROM-Based Image Types

This section describes the initialization sequence for the ROM-based VxWorks images `vxWorks_rom` and `vxWorks_resrom_nosym`.

**NOTE:** For 64-bit VxWorks, only project builds are supported (with `vxprj` and Workbench). You cannot use the legacy method using `bspDir/config.h` to configure VxWorks and `bspDir/make` to build it (this method is sometimes referred to simply as a BSP build).

---

**romInit.s: romInit( )**

At power-up (cold start) the processor begins execution at `romInit()`. `romInit()` performs the minimal setup necessary to transfer control to `romStart()` which is located in:

```
installDir/vxworks-6.x/target/config/all/bootInit.c
```

Most hardware and device initialization is performed later in the initialization sequence by `sysHwInit()` in `sysLib.c` (for 32-bit) and by `sysHwInit1()` in `sysLib.c` (for 64-bit).

`romInit()` is responsible for the following actions:

- Initializing the processor.
- Initializing access to target DRAM.
- Jumping to `romStart()` for further initialization; passing `BOOT_COLD` on a cold boot, or the parameter passed from `sysToMonitor()` on a warm boot.

For sample initialization, see the `romInit.s` file in your reference BSP.

Take care not to add too much initialization code to `romInit()`. It is better to do the minimum amount of initialization at this time and to perform additional initialization later in the process. Because some BSPs do more initialization in `romInit()` than necessary, it is wise to review the code from the reference BSP and remove any initialization code that is unnecessary at this stage of the boot sequence.

**bootInit.c: romStart( )**

The text and data segments are copied from ROM to RAM in one of the following ways:

- For `vxWorks_rom`, both the text and data segments are copied to RAM.
- For `vxWorks_resrom_nosym`, only the data segment is copied to RAM.

After the copy action, verify that the data segment is properly initialized. For example:
int thisVal = 17; /* some data segment variable */
...
if (thisVal != 17)
somethingIsWrong();

If something is wrong, check whether RAM access is working properly. For example:

int dummy;
...
dummy = 17;
if (dummy != 17)
somethingIsWrong();

If RAM access is working, check that the data segment was copied into memory at the right offset. This is only a problem for vxWorks_resrom_nosym images. The romStart() routine assumes that the data is located at some architecture-specific offset from the end of the text segment in ROM. The exact address used is an architecture-dependent offset from the etext symbol. For the specific value used by your architecture, see the copyLongs() or bcopyLongs() calls in romStart() which is located in:

installDir/vxworks-6.x/target/config/all/bootInit.c

Keep in mind that the memory offset can be different from the reference BSP offset if you are using alternative tools to create your ROM image or if you are using a custom linker script to create your VxWorks image. In these cases, you may need to adjust the offset accordingly.

The last task romStart() performs is to call the generic initialization routine usrInit() in usrConfig.c for BSP directory builds and prjConfig.c for project builds. The rest of the initialization sequence is described in Initializing All Image Types, p.64.

Creating Additional Diagnostic Routines

Once the processor and memory have been initialized, you have the opportunity to spend some time preparing additional diagnostic routines to help with the remainder of the development effort.

LED Routines

If you have not done so already, you may want to create LED routines for debugging purposes. For more information, see 3.2.3 Using Debug Routines in the Initialization Code, p.58.

Console Output Routines

In addition to the LED routines described above, it may also be helpful to have a polled-mode serial output routine. The first step in producing this routine is to create a more basic routine, possibly called something such as outConsole(), that does unformatted polled-mode serial output. This does not allow numbers to be displayed easily, but it does allow diagnostic output. The prototype of outConsole() is as follows:

```c
STATUS outConsole
{
    char *    buffer, /* buffer passed to routine */
    int       nchars, /* length of buffer */
    int       outarg /* arbitrary arg passed from fmt routine */
}
```
This routine disables all interrupts; use polled mode output for the serial device until all characters have been printed, then reset the interrupt mask to the previous value.

Once the `outConsole()` routine is created with the prototype specified above, it is possible to create a `printf()` routine to allow formatted output. The source code is similar to the following:

```c
int printf
{
    const char * fmt, /* format string to write */
    ...
    /* optional arguments to format string */
}
{
    va_list vaList; /* traverses argument list */
    int nChars;
    va_start(vaList, fmt);
    nChars = fioFormatV(fmt, vaList, outConsole, 1);
    va_end(vaList);
    return (nChars);
}
```

**NOTE:** This suggested source code for `printf()` assumes that formatted I/O is configured into the system. However, this may not be the case for your system. When you are ready to finalize your BSP, the `printf()` routine should be surrounded by `#ifdef INCLUDE_STDIO` and `#endif /* INCLUDE_STDIO */` statements to avoid pulling in the formatted I/O module unexpectedly. As an alternative, you can write a `printf()` routine without calling `fioFormatV()`. However, the non-standard I/O implementation requires more development effort.

### Copying Additional Code From the Reference BSP

If you followed the guidelines provided in earlier sections, your initial development for processor and memory initialization consisted of a number of files containing empty stub routines. At this point, you can copy relevant material from the reference BSP to your BSP directory.

It is also common practice for developers to copy `usrConfig.c` to the BSP directory (for BSP directory builds) and modify `Makefile` with the `USRCONFIG=` line such that a local version of the `usrConfig.c` file is used. In this practice, the contents of the `usrInit()` and `usrRoot()` routines are commented out using `#if 0 /* BSP development */` and `#endif /* BSP development */`. You may also want to add a call to a debugging output routine (such as `printf()` at the end of `usrInit()` (before the call to `kernelInit()`) and at the beginning of `usrRoot()`.

When BSP development reaches the stage where `usrInit()` is being executed, remove the commenting from short sections to verify that the system continues to work with these sections in place. This is done by moving the `#if 0 /* BSP development */` line to below the sections that should now be included. At the end of `usrInit()`, a call to `kernelInit()` is made. Once you reach the point at which all the `usrInit()` code is included and the call to `kernelInit()` is occurring, the system should execute the diagnostic message that you had placed at the beginning of `usrRoot()`. If it does not execute the message, there is a problem with the BSP. Once `usrRoot()` is successfully called, start removing the commenting from sections of `usrRoot()`.
NOTE: If the board fails to boot when a system module is initialized, it is almost always caused by a failure in the BSP.

Initializing RAM-Based Image Types

This section describes sysInit(), the initialization routine for the RAM-based VxWorks image. The routine is located in sysAlib.s. It is the VxWorks entry point and performs the minimal setup necessary to transfer control to usrInit() (in usrConfig.c for BSP directory builds and prjConfig.c for project builds). Most hardware and device initialization is performed later in the initialization sequence by sysHwInit() in sysLib.c.

Initializing All Image Types

The remainder of the initialization code is common to both ROM- and RAM-based images.

NOTE: usrInit() is in usrConfig.c for BSP directory builds and prjConfig.c for project builds.

usrConfig.c: usrInit()

From a BSP writer’s point of view, the main significance of usrInit() is that it clears the bss segment so that uninitialized C global variables are now zero, and then calls sysHwInit() (in sysLib.c) to initialize the hardware. If memory is set up properly, there is little that can go wrong in this routine.

sysLib.c: sysHwInit()

This is the heart of the BSP initialization code. This routine must reset all hardware to a quiescent state so as not to generate uninitialized interrupts later when interrupts are enabled.

Note that it is not sufficient to manipulate the processor interrupt mask. In the general case, it is possible for multiple devices to be connected to the same interrupt line. If this is the case, disabling the interrupt controller for that interrupt line prevents interrupts at the time sysHwInit() is executing. However, when interrupts are enabled later, or when the first device connected to a given interrupt line is enabled, other devices using the same interrupt line may cause unacknowledged interrupts, resulting in a system failure.

The first stage of VxBus initialization occurs during the call to the硬WareInterFaceInit() routine. This initializes a small pool of memory for use by VxBus and VxBus device drivers, and probes for buses and devices. The order of initialization in sysHwInit() should be initialization of hardware which is not associated with specific drivers, then a call to硬WareInterFaceInit(), and finally network and serial device initialization and initialization of legacy device drivers. Additional VxBus initialization is done from sysHwInit2(), discussed later. For more information about VxBus initialization, see the VxBus Device Driver Developer’s Guide: Device Driver Fundamentals.
usrConfig.c: usrInit()

After sysHwInit() completes, control returns to usrInit(). The last task usrInit() performs is a call to kernelInit() to start the VxWorks kernel. This is the end of the pre-kernel initialization code. The kernelInit() routine does not return. Rather, it starts the kernel with usrRoot() as the first task.

At this point, if you want to bring the kernel up under control of the WDB agent, you can start the agent. This is an optional step and is not typically performed. For more information, see 3.2.6 Starting the WDB Agent Before the Kernel, p.66.

> NOTE: usrInit() is in usrConfig.c for BSP directory builds and prjConfig.c for project builds.

kernelInit()

The kernelInit() routine source code is not normally available during BSP development. Instead, the object module is extracted from a library. At this juncture, the source code cannot be modified to add debugging information.

However, kernelInit() is very important during BSP development because, deep within the kernelInit() routine, interrupts are finally enabled. A very common mistake in BSP development is the failure to ensure that all interrupt sources are quiescent prior to enabling interrupts. If this is not done before the call to kernelInit(), the system typically hangs or fails, causing confusion for the BSP developer because the source of the problem is not obvious.

If kernelInit() is called but execution fails to reach the start of usrRoot(), or if the system behaves erratically in other ways after the call to kernelInit(), usually it is one of two problems.

Either sysMemTop() is returning a bad address or, more likely, some device has not been reset and is generating an interrupt. In the latter case, you must modify sysHwInit() to reset the interrupting device. This applies to 32-bit only.

To find the source of the interrupt, start by figuring out the interrupt vector being generated, applying any of the following techniques:

- Use a logic analyzer to look for instruction accesses to the interrupt vector table.
- Use an OCD device to set breakpoints in the interrupt vector table.
- Modify sysHwInit() to mask suspected interrupt vectors through an interrupt controller.
- Modify sysHwInit() to connect debugging routines to the suspected interrupt vectors using intVecSet() (you cannot use intConnect() because it calls malloc() and the VxWorks memory allocator is not yet initialized).

usrConfig.c: usrRoot()

The remainder of the VxWorks initialization is done after the kernel is started in usrRoot(). The details of the initialization process are covered in subsequent sections. In this phase, it is enough for usrRoot() to verify that sysHwInit() is properly written.

At this point, you have a working kernel but no device drivers. The only drivers required by VxWorks are a timer and possibly an interrupt controller. Most BSPs also have serial drivers.
3.2.6 Starting the WDB Agent Before the Kernel

This step is optional when creating a new BSP and is rarely performed. However, if you have a slow download environment, you may want to put everything in ROM as early as possible and thus save download cycles. In this case, starting the WDB agent before the kernel is desirable.

Keep in mind, there are several disadvantages to starting the agent before the kernel. These are:

- Once the hardware initialization code is written, bringing up the kernel takes less time than bringing up the agent. Because most developers consider a working kernel to be the more significant milestone, they usually start with the kernel.
- Starting the agent before the kernel is not a significant help to getting the basic kernel working. This is because the basic kernel adds little to what you have written already: the basic kernel adds only a timer driver and possibly an interrupt controller, which are simple devices.
- Starting the WDB agent before the kernel limits you to system-mode debugging.

Starting the agent before the kernel is described in the Wind River Workbench documentation and the VxWorks Kernel Programmer’s Guide. The following sections provide important additional information.

Caveats

Because the virtual I/O driver requires the kernel, add the following line to config.h:

```c
#define INCLUDE_WDB_VIO
```

There is an important caveat if you are planning to use the target agent serial-line communication path. When the kernel is first started, interrupts are enabled in the processor, but driver interrupt handlers are not yet connected. You must ensure that the serial device you use for agent communication does not generate an interrupt.

If your board has an interrupt controller, use it to mask serial interrupts in `sysHwInit()`. Be aware that the target agent attempts to use all drivers in an “interrupt on first packet” mode. As a result, you should modify the serial driver to refuse to go into interrupt mode, even if the agent requests that mode.

System-Mode Debugging Techniques

After you have the agent working, you can use it to debug the VxWorks image to which it is linked. To save download time, link the VxWorks code you want to
test—driver code, in particular—into the ROM image. To avoid re-making ROMs, consider adding hooks to your BSP and driver routines as follows:

```c
void (*myHwInit2Hook)(void); /* declare a hook routine */
...
void sysHwInit2 (void)
{
  if (myHwInit2Hook != NULL) /* and conditionally call it */
  {
    myHwInit2Hook();
    return;
  }
  ... /* default code */
}
```

Adding hooks allows you to replace the routine from the debugger dynamically. For example, to override the behavior of `sysHwInit2()` above, create a new version of it called `myHwInit2()` in a module called `myLib.o`, and then type the following:

```
(gdb) load myLib.o
(gdb) set myHwInit2Hook = myHwInit2
(gdb) break myHwInit2
(gdb) continue
```

However, if you start the agent before the kernel, you must start the agent after the call to `sysHwInit()`. Therefore, you cannot override `sysHwInit()`. Alternatively, you can add additional hardware initialization code that is called before the kernel is started. In this case, you add a hook right before the call to `kernelInit()`.

As an alternative to hooks, you can call routines from the debugger using the GDB call procedure. For example:

```
(gdb) call myHwInit2
```

The advantage of using hooks instead of the call mechanism is that:

- Hooks let you avoid executing the original code.
- Hooks are much faster than the call mechanism.

If your board has an “abort” or “halt” button, consider connecting a debugging routine to the abort interrupt. Then you can set a breakpoint on your interrupt handler from the debugger. This provides a way for you to gain control of the system if it appears to fail. In this case, it is best to have the abort switch tied to a non-maskable interrupt (NMI).

⚠️ **CAUTION:** Only unrecoverable interrupts such as “abort” can be connected to an NMI. If a device interrupt is connected to an NMI, the kernel does not work properly.

### 3.2.7 Building and Downloading VxWorks

The VxWorks image you load to the target depends on the download method you use. The primary images are as follows:

- **vxWorks**
  This image starts execution from RAM. It must be loaded into RAM by some external means such as the board’s native debug ROMs.
This image starts execution from ROM, but its text and data segments are linked to RAM addresses. Early in the initialization process, it copies itself into RAM and continues execution from that point.

This image executes from ROM. Only the data segment is copied into RAM. For additional image types, see 2.2.2 Boot Sequence Configurations, p.10.

If your download path puts the image in RAM (such as when using a vendor debug ROM), use the vxWorks image. If your download path puts the image in ROM (such as when using NetROM), use either vxWorks_rom or vxWorks_resrom_nosym. The advantage of vxWorks_rom is that it can be more easily debugged because software breakpoints can only be set on RAM addresses. The advantage of vxWorks_resrom_nosym is that it uses less target memory. The makefile for both ROM images lets you specify an optional .hex suffix (for example, vxWorks_rom.hex) to produce an S-record file, in addition to the object file.

There is a file called depend.cputool that contains the file dependency rules used by the makefile. If this dependency file does not already exist, the makefile automatically generates it. If you add new files to the BSP, delete the dependency file and let the makefile generate a new file.

After you have downloaded your code to the target, examine memory or ROM to make sure that the code is loaded in the right place. Use the nm and objdump utilities on the VxWorks image to compare what should be in the target memory with what is actually there. Pay special attention to the start addresses of the text and data segments.

For additional information on building and downloading VxWorks, see the VxWorks Kernel Programmer’s Guide.

3.2.8 Interrupt Controllers

For information about interrupt controller driver design and construction, see VxBus Device Driver Developer’s Guide: Interrupt Controller Drivers.

3.2.9 DMA

This release implements a model for DMA. For more information, see VxBus Device Driver Developer’s Guide: Direct Memory Access Drivers.

3.2.10 Minimum Required Drivers

The only driver required by VxWorks is the system clock, although certain architectures, such as Intel Architecture and PowerPC, also require an interrupt controller driver.

Implementing a system clock driver typically involves using one of the existing drivers from:

installDir/vxworks-6.x/target/src/hwif/timer/
Or (if an interrupt controller driver is required):

\[\text{installDir/vxworks-6.x/target/src/drv/target/src/hwif/intrCtl/}\]

If you are reusing an existing driver, simply perform board-specific hardware initialization in \text{sysHwInit()} and connect the interrupt by calling \text{intConnect()} in \text{sysHwInit2()}. The timer drivers are simple devices that can be tested by modifying \text{usrRoot()} to perform some action periodically, such as blinking an LED.

For example:

```c
void myTestCode (void)
{
    while (1)
        taskDelay (5*sysClockRateGet());
        sysFlashLed();
    }
}
```

For a normal development system, it is often useful to have serial and Ethernet drivers as well, but these are not required. For additional information, see \textit{VxBus Device Driver Developer's Guide}.

### 3.2.11 Serial Drivers

Most BSPs include an SIO driver for the console serial port. In many cases, one of the standard drivers can be used. These drivers are located in:

\[\text{installDir/vxworks-6.x/target/src/hwif/sio}\]

For more information, see \textit{VxBus Device Driver Developer's Guide: Serial Drivers}.

Early in the development process, it was suggested that you remove most of the body of \text{usrRoot()} with \#if FALSE/\#endif pairs. In order to enable the serial driver, you must move the \#if FALSE line to a point further down in the code, below the point at which the serial devices are initialized. To test the port, modify \text{usrRoot()} to spawn some application test code that tests your driver.

For example, periodically print a message to the console as follows:

```c
void myTestCode (void)
{
    extern int vxTicks;
    char * message = "still going...\n";
    while (1)
        /* print a message every 5 seconds */
        taskDelay (5*sysClockRateGet());
        write (1, message, strlen (message));
}
```

If none of the standard drivers is applicable to your BSP, you can create a custom SIO driver using a similar driver from:

\[\text{installDir/vxworks-6.x/target/src/hwif/sio}\]

And the template driver in:

\[\text{installDir/vxworks-6.x/target/src/hwif/sio/vxbTemplateSio.c}\]
3.2.12 VxBus Initialization Routines

The following routines are provided by VxBus for the BSP to initialize bus types and device instances. For more information, see the VxBus Device Driver Developer’s Guide: Device Driver Fundamentals.

**hardWareInterFaceInit()**

Called once by the BSP at boot time from within `sysHwInit()`. This should be called after basic processor and board initialization has been performed to quiesce on-chip and on-board devices. If non-VxBus drivers are included in the system, their initialization should occur after the call to `hardWareInterFaceInit()`. This routine makes a call to `vxbInit()`, as described below.

```c
STATUS hardWareInterFaceInit()
```

Drivers for devices other than bus controllers should disable interrupt generation on the device. They are not required to perform any other actions at this time. No operating system facilities are available at this time, and requesting any operating system facility has unpredictable results (for example, crashing the system). The first-pass initialization routine must not allocate memory from the system memory pool, but some memory can be allocated using `hwMemAlloc()`. The driver is responsible for keeping track of what memory is allocated using `hwMemAlloc()` and what memory is allocated from the system memory pool.

**vxbDevInit()**

Called once by the BSP at boot time. This routine performs the second pass of bus and device initialization. The `init2` routine for each instance is called.

This routine should be called from within `sysHwInit2()` after interrupt handler initialization code, if any, is performed and after the system clock rate is set, but before the network is initialized.

```c
STATUS vxbDevInit()
```

The second-pass driver initialization routine is called from `sysHwInit2()`. This is where most drivers should initialize the hardware. Memory allocation is available.

Serial devices present a special case for the second-pass driver initialization routine. The serial device may be the system console. For this reason, serial drivers should complete their initialization in the second-pass initialization routine, and enable themselves to be connected to the I/O system so that they can be connected as the console.

**vxbDevConnect()**

Called once by the BSP at boot time from within `usrRoot()` but before the network initialization is performed. This routine performs the third and final pass of bus and device initialization. The connect routine for each instance is called.
The driver connection routine is called from late in `usrRoot()`. The purpose of this routine is to give the driver an opportunity to prepare itself for use by operating system facilities and middleware. This may include connecting itself to the I/O system, creating virtual block devices, and other similar actions.

At the time the driver connection routine is called, most operating system facilities are available. However, the network stack is not yet initialized. For details on exactly which operating system facilities are available, check the `usrRoot()` code in `usrConfig.c` for BSP directory builds and `prjConfig.c` for project builds.

In the case that an instance is discovered after system initialization (for example, due to a newly downloaded driver being associated with a previously orphaned device or to hot insertion of a device), all initialization routines are called consecutively at the time of instance creation.

### 3.3 Finalizing Your BSP

This section summarizes the remaining tasks essential to completing your BSP port. Included is a discussion of cleanup, timers, networking, and other issues.

#### 3.3.1 Removing Development-Related Changes

During BSP development, a number of files may be changed, and specific debugging routines are developed. These changes should be cleaned up before the BSP is finalized.

After the BSP is working with a minimum set of drivers, the private copies of `usrConfig.c` (BSP directory builds only) and `bootInit.c` should be removed from the BSP directory so that the BSP uses the generic versions of these files located in:

```
installDir/vxworks-6.x/target/config/all
```

To reinstall the generic versions, remove the following lines from your BSP `Makefile`:

```
BOOTINIT = bootInit.c
USRCONFIG = usrConfig.c
```

Execute a `make clean`.

Previously, you may have masked out unwanted configuration code with `#if FALSE/#endif` pairs. Now the unwanted code should be removed in a more standard way. That is, you must redefine the appropriate macros in `config.h`. Keep in mind the macro dependencies listed in the `usrDepend.c` file located in:

```
installDir/vxworks-6.x/target/src/config/
```

The LED routines used for debugging are typically short and are likely to be useful for applications after BSP development is complete. For these reasons, they should not be removed. If the routines have not been rewritten in C, you may want to rewrite them now in order to ease the future support burden.
3.3.2 Creating Workbench Projects

The Wind River Workbench documentation contains the necessary information for creating, configuring, and building projects. These steps can all be handled through the project facility. For more information, see the Wind River Workbench By Example or online help.

3.3.3 Adding Other Timers

Your driver can include an auxiliary clock driver—if your target hardware supports it—as well as a high-resolution timestamp driver.

The auxiliary clock is used by the VxWorks spy() utility, and also by certain Wind River host tools. For more information on the auxiliary clock interface, see the reference entries for the various sysAuxClk() routines and the VxBus Device Driver Developer’s Guide.

The auxiliary clock interface consists of the following routines:

- `sysAuxClkConnect()` connect a routine to the auxiliary clock interrupt
- `sysAuxClkDisable()` turn off auxiliary clock interrupts
- `sysAuxClkEnable()` turn on auxiliary clock interrupts
- `sysAuxClkInt()` handle auxiliary clock interrupts
- `sysAuxClkRateGet()` return the current auxiliary clock interrupt rate
- `sysAuxClkRateSet()` set the auxiliary clock interrupt rate

The high-resolution timestamp driver is currently used only by the Wind River System Viewer, but writing a timestamp driver can be useful to you for future debugging and can also be useful for application developers using your BSP. The interface can be found in the following header file:

```
installDir/vxworks-6.x/target/h/drv/timer/timestampDev.h
```

3.3.4 Enabling NVRAM

VxWorks defines an interface for reading and writing to a persistent storage area. This interface is called the non-volatile memory library. VxWorks uses this library to store boot information.

Early in the development process, Wind River recommends that you include `nullNvRam.c` in `sysLib.c`. When you finalize your BSP, it is best to replace `nullNvRam.c` with a more appropriate interface so that boot information can be kept after the board is powered off.

The driver files in the following directory include generic versions of NVRAM support routines:

```
installDir/vxworks-6.x/target/src/drv/mem/
```

These files include:
3.3 Finalizing Your BSP

nvRam.c
Supports NVRAM that is addressed as if it were RAM (for example, battery-backed RAM)

eRamToFlash.c
Supports target systems with flash but no other non-volatile memory

nullNvRam.c
Supports target systems with no electronically programmable non-volatile memory

NOTE: Wind River does not recommend using nullNvRam.c in your final BSP unless it is the only option available for your target system.

Each of the standard files provides the required routines sysNvRamSet() and sysNvRamGet(), and can be #included in sysLib.c. The nvRamToFlash.c version provides read and write wrappers that call the flash memory routines sysFlashGet() and sysFlashSet(). If nvRamToFlash.c is used, the sysFlashGet() and sysFlashSet() routines must be provided by the BSP. However, if the nvRam.c version works for your board, no additional work is necessary.

If a more complex system is required or if the board’s NVRAM does not work with the provided routines, NVRAM support can be provided by the BSP. In many BSPs, the required routines are put in the sysLib.c file. However, the preferred method is to put NVRAM support in a separate file called sysNvRam.c. In either case, the routines sysNvRamSet() and sysNvRamGet() must be provided to prevent undefined references when building VxWorks.

If it is necessary to write custom versions of these routines, the routines from the standard libraries in the following directory can be used as a reference:

installDir/vxworks-6.x/target/src/drv/mem/

3.3.5 Enabling Cache and MMU Devices

The next step in the BSP development process is to get the BSP working with the cache and MMU enabled. The following sections outline the steps required to add and enable cache and MMU support to a BSP.

The cache and MMU are often interdependent, making it impractical to deal with one in isolation of the other. On some architectures, the MMU must be enabled and supported if you want to use the cache. On most architectures, the MMU supports varying cache attributes on a page by page basis.

Enabling the cache can significantly improve performance. The level of cache support required by software is highly dependent on the hardware architecture and processor. On some processors (especially CISC processors and some multi-CPU RISC processors), cache coherency is handled automatically by the target hardware. On other processors, operating system and driver software must take steps to manage the cache state to ensure coherency between CPU(s) and devices, or multiple CPU(s) within a processor.

MMU support is a prerequisite for systems that require paged memory management and memory protection. In VxWorks, this includes features such as:

- mapping pages in virtual space to physical memory
- setting caching attributes on a per-page basis
• setting protection attributes on a per-page basis
• setting a page mapping as valid or invalid
• locking and unlocking TLB entries for pages of memory
• enabling page optimization

In addition to these basic features, MMU support is needed to enable the following useful debugging and kernel or application hardening features:
• task stack overrun and underrun detection
• interrupt stack overrun and underrun detection
• non-executable task stacks
• text segment write-protection
• exception vector table write-protection

Finally, MMU support is required for SMP systems.

Cache and MMU BSP Development Overview

NOTE: Before implementing cache support in your BSP, consult the VxWorks Architecture Supplement to familiarize yourself with cache and MMU issues that are particular to your hardware platform.

Typically, implementing cache and MMU support in a BSP consists of the following:
1. Manage the cache and MMU during reset and initialization. (See Managing Cache and MMU During Reset and Initialization, p.74.)
2. Initialize the cache and MMU support layers. (See Initialize VxWorks Cache and MMU Support Layers, p.75.)
3. If applicable, provide architecture-specific cache and MMU support. (See Providing Architecture-Specific Cache and MMU Support, p.82.)
4. If applicable, support the L2 and L3 caches. (See Supporting L2 and L3 Cache (Optional), p.83.)
5. If applicable, support cache locking. (See Supporting Cache Locking (Optional), p.84.)
6. Define the cache and MMU configuration (components, parameters, and macros). (See Defining the Cache and MMU Configuration, p.85.)

Managing Cache and MMU During Reset and Initialization

The following sections describe the cache and MMU support that may be required for reset and initialization code. This includes romInit( ) (the boot loader entry point), sysInit( ) (the VxWorks entry point), and sysToMonitor( ) (the VxWorks reset function).

NOTE: The handling described in this section is not required for all architectures. The exact requirements vary depending on your target architecture.
Your architecture code may provide support routines for the functionality described in this section or you may need to implement the functionality in the BSP. For example, the MIPS architecture provides a TLB clear routine `mipsTlbClear()`, and various cache reset routines for different varieties of MIPS processors. When possible, you should use all such architecture support routines when supporting the cache and MMU for your target hardware.

### Managing Cache During Reset and System Initialization

The following cache handling may be required in `romInit()` (depending on your architecture) when the system is booted or reset:

- Enable instruction and/or data caches. This can drastically improve performance when booting RAM-resident runtime images, which must be copied from ROM to RAM by the `romStart()` routine. If the `BOOT_CLEAR` option is set in the boot flags, `romStart()` clears all of the `LOCAL_MEM_space`, including `USER_RESERVED_MEM` and `PM_RESERVED_MEM`, because memory is presumed to contain garbage on a cold boot (from power off).
- Initialize instruction and data caches to make sure all entries are marked invalid. This is done to prevent the possibility of uninitialized data being written to RAM.

The following cache handling may be required (depending on your architecture) during the initialization of the VxWorks image (`sysInit()`, `sysHwInit()`):

- Flush and invalidate all caches in preparation for VxWorks hardware and system initialization.

The following cache handling may be required (depending on your architecture) by the VxWorks system reset function, `sysToMonitor()`:

- Flush and invalidate all caches in preparation for reset.

### Managing MMU During Reset and System Initialization

The following MMU handling may be required in `romInit()` when the system is booted or reset:

- Enable and clear the MMU to a reset/initialization state. The goal of this step is to return the MMU to a quiescent state.
- Set up the MMU to support execution of the boot loader. The goal of this step is to set up the MMU such that the VxWorks boot code can be executed. For RAM-based boot loader images, the MMU must be initialized to allow the boot loader image to be copied to RAM and executed.

The following MMU handling may be required during the initialization of the VxWorks image (`sysInit()`, `sysHwInit()`):

- In `sysInit()`, initialize the MMU to support VxWorks kernel execution. The requirements for this depend on a number of factors, including your memory map, and the multiprocessing mode (UP, SMP, AMP).
- In `sysHwInit()`, add MMU entries to map device registers or buses that must be visible to device drivers. This must be completed prior to calling `hardWareInterFaceInit()`.
Initialize VxWorks Cache and MMU Support Layers

This section outlines the cache and MMU support provided by the architecture and OS software layers, and how to initialize the modules that implement these support layers.

Initialize VxWorks Cache Support Layers

VxWorks provides the bulk of cache support in three layers outside the BSP:

- the core OS (cacheLib)
- the Architecture Independent Manager (AIM)
- the architecture support code in installDir/vxworks-6.x/target/src/arch/

The cacheLib module defines a cache API to cover all the typical cache operations required by the OS and device drivers. The initialization routine for cacheLib (cacheLibInit()) calls an architecture-specific initialization routine (cacheArchLibInit() by default) to initialize the cacheLib structure. The cacheLib structure is a set of pointers to routines that implement all the required cache operations for a given architecture.

Most architectures support various processors with different cache requirements. Therefore, cacheLib provides an architecture initialization function pointer (sysCacheLibInit) that can be set to the cache initialization routine that is appropriate for the architecture variant. To enable this method of variant-specific cacheLib initialization, the _ARCH_MULTIPLE_CACHELIB macro must be defined. This is typically done in the architecture-specific header file (archPpc.h, for instance).

For architectures where _ARCH_MULTIPLE_CACHELIB is defined to be TRUE, the BSP or architecture must set the sysCacheLibInit pointer to point to a function within the architecture code that performs the appropriate initialization of the cacheLib structure.

The setting of sysCacheLibInit is handled differently by the various architectures as follows:

ARM and XScale
   Handled by the architecture code for most BSPs. Some BSPs that support processors with no cache set it to NULL.

IA-32
   Does not use sysCacheLibInit (_ARCH_MULTIPLE_CACHELIB is defined to be FALSE). Generic cache support is provided by cacheArchLib and cacheALib.s.

Intel 64
   Does not use sysCacheLibInit.

MIPS
   sysCacheLibInit is determined by the CACHE_LIB_INIT_FUNC macro. (See Defining the Cache Configuration, p.85.)

PowerPC
   Does not use sysCacheLibInit (_ARCH_MULTIPLE_CACHELIB is defined to be FALSE). Instead, each architecture cache support library defines its own instance of cacheArchLibInit(). For each variant, one of these cache support libraries is selected at build time.
In most architectures (except IA-32) the architecture cache library initialization function initializes the AIM cache library \(\text{aimCacheLib}\) by setting up a \texttt{CACHECONFIG} structure and calling \texttt{aimCacheInit( )}. \texttt{aimCacheInit( )} in turn initializes the top level \texttt{cacheLib} structure.

The cache AIM provides additional architecture-independent functionality beyond \texttt{cacheLib}. It handles many of the details that are necessary for the successful implementation of the \texttt{cacheLib} routines, and distills the level of functionality that needs to be provided by architecture code to a minimum. It allows architecture code to specify nothing for a given operation, in which case it substitutes its own reasonable default. It lets the architecture code specify certain primitive operations which work in conjunction with its implementation of the \texttt{cacheLib} routines. Or, it lets the architecture completely override any given operation.

The resulting cache module hierarchy is shown in Figure 3-1.

**Figure 3-1**  
**Cache Module Hierarchy**

---

**Initializing VxWorks MMU Support Layers**

As with cache support, VxWorks provides the bulk of MMU support in three layers outside the BSP:

- the core OS (\texttt{vmBaseLib, vmLib})
- the AIM MMU library (\texttt{aimMmuLib})
- the architecture support code in \texttt{installDir/vxworks-6.x/target/src/arch/}

These upper layers are primarily configured through components, parameters, preprocessor macros, build options, and the physical memory descriptor table. Most of this configuration is covered in *Defining and Initializing the Physical Memory Descriptor Table*, p.78 and *Defining the Cache and MMU Configuration*, p.85.
All architectures support a number of MMU configurations, and the appropriate architecture support module is linked into the VxWorks image based on the architecture variant selected.

In most architectures (except ARM and Intel architectures) the architecture MMU library initialization function initializes the AIM MMU library (`aimMmuLib`) by calling `aimMmuLibInit()`. ARM and IA-32 MMU modules interface directly with `vmLib` and `vmBaseLib`.

The MMU AIM provides an abstraction layer to interface with the underlying architecture-dependent MMU code. This allows uniform access to the hardware-dictated MMU model that is typically CPU-specific. This abstraction layer also adds support for two new routines, `vmPageLock()` and `vmPageOptimize()`, to the VxWorks `vmLib` API. For more information, see the reference entries for these routines.

The resulting MMU module hierarchy is shown in Figure 3-2.

### Figure 3-2 MMU Module Hierarchy

```
OS and Device Drivers

vmBaseLib and vmLib

aimMmuLib

architecture or variant-specific MMU library

(ARM/XScale and IA-32)
```

#### Defining and Initializing the Physical Memory Descriptor Table

For 32-bit BSPs, the physical memory descriptor table (`sysPhysMemDesc[]`) is created and initialized in the BSP `sysLib.c` file. The table is an array of structures of type `PHYS_MEM_DESC` (defined in `vmLib.h`).

```c
typedef struct phys_mem_desc {
    VIRT_ADDR virtualAddr; /* block virtual address */
    PHYS_ADDR physicalAddr; /* block physical address */
    msize_t len; /* block length */
    UINT initialStateMask; /* mask parameter to vmStateSet */
    UINT initialstate; /* state parameter to vmStateSet */
} PHYS_MEM_DESC;
```

Each `PHYS_MEM_DESC` table entry describes a region of memory in terms of its virtual address, physical address, length, cache attributes, and protection.
attributes. At a minimum, define a PHYS_MEM_DESC entry for each non-contiguous region of physical memory that needs to be accessed by the kernel, drivers, and applications. Multiple PHYS_MEM_DESC entries may also be defined for regions of contiguous memory that need different protection or cache attributes. Examples of candidates for a PHYS_MEM_SEC table entry are kernel memory, application memory, and device memory.

The values for initialStateMask and the initialState fields are formed by combining bit masks found in vmLib.h. Mask bits are or'ed together in the initialStateMask structure element to describe which state bits are being specified in the initialState structure element.

The following is a partial example definition, taken from the pcPentium4 BSP:

```c
#define VM_STATE_MASK_FOR_ALL \  VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE | VM_STATE_MASK_CACHEABLE
#define VM_STATE_FOR_IO \  VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE_NOT
#define VM_STATE_FOR_MEM_OS \  VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE
#define VM_STATE_FOR_MEM_APPLICATION \  VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE

PHYS_MEM_DESC sysPhysMemDesc [] =
{
    /* lower memory for invalid access */
    {
        (VIRT_ADDR) 0x0,
        (PHYS_ADDR) 0x0,
        _WRS_BSP_VM_PAGE_OFFSET,
        VM_STATE_MASK_FOR_ALL,
        VM_STATE_FOR_MEM_OS
    },
    /* lower memory for valid access */
    {
        (VIRT_ADDR) _WRS_BSP_VM_PAGE_OFFSET,
        (PHYS_ADDR) _WRS_BSP_VM_PAGE_OFFSET,
        0xa0000 - _WRS_BSP_VM_PAGE_OFFSET,
        VM_STATE_MASK_FOR_ALL,
        VM_STATE_FOR_MEM_OS
    },
    /* video ram, etc */
    {
        (VIRT_ADDR) 0x000a0000,
        (PHYS_ADDR) 0x000a0000,
        0x00060000,
        VM_STATE_MASK_FOR_ALL,
        VM_STATE_FOR_IO
    },
    /* upper memory for OS */
    {
        (VIRT_ADDR) LOCAL_MEM_LOCAL_ADRS,
        (PHYS_ADDR) 0x0,
        _WRS_BSP_VM_PAGE_OFFSET,
        VM_STATE_MASK_FOR_ALL,
        VM_STATE_FOR_MEM_OS
    },

    /* upper memory for device */
    {
        (VIRT_ADDR) LOCAL_MEM_LOCAL_ADRS,
        (PHYS_ADDR) 0x0,
        _WRS_BSP_VM_PAGE_OFFSET,
        VM_STATE_MASK_FOR_ALL,
        VM_STATE_FOR_MEM_OS
    },
}
```
In addition to defining the `sysPhysMemDesc[]` table itself, the BSP must initialize the `sysPhysMemDescNumEnt` global variable with the number of entries in the table. For example:

```c
int sysPhysMemDescNumEnt = NELEMENTS(sysPhysMemDesc);
```

The `sysPhysMemDesc[]` table can be modified at runtime. This is useful, for example, with PCI drivers that can be auto-configured, which means that memory requirements are detected at runtime. In this case, the size and address fields can be updated programmatically for the corresponding `sysPhysMemDesc[]` entries. Make such updates before the virtual memory (VM) subsystem is initialized by `usrMmuInit()` (for example, during execution of `sysHwInit()`). Also, in this scenario, the `sysPhysMemDescNumEnt` value must be computed at runtime.

For kernel configurations that do not include RTP support, the `sysPhysMemDesc[]` table is read and processed directly by `vmGlobalMap` and the `vmGlobalMapInit()` routine. When RTP support is enabled (INCLUDE_RTP), the address space library (adrSpaceLib) creates a copy of `sysPhysMemDesc[]` that does not map the space between the top of the kernel heap and `sysMemTop()`. This space is reserved for real time processes (RTPs), shared libraries (SLs), and shared data regions (SDs). The modified copy of `sysPhysMemDesc[]` is then processed by `vmGlobalMap` and the `vmGlobalMapInit()` routine.

For more information, also see the reference entry for `vmGlobalMap`.

The following `sysPhysRamDesc[]` example is for the case when memory auto-sizing is enabled (has "dummy" entries).

```c
LOCAL PHYS_MEM_DESC sysPhysRamDesc [] =
{
    /* Kernel System memory in first 2GB */
```
(VIRT_ADDR) LOCAL_MEM_LOCAL_ADRS,
(PHYS_ADDR) LOCAL_MEM_VIRT_TO_PHYS(LOCAL_MEM_LOCAL_ADRS),
LOCAL_MEM_SIZE, /* may be modified at runtime */
VM_STATE_MASK_FOR_ALL,
VM_STATE_FOR_MEM_OS
},

/* Dummy entries*/

{
 (VIRT_ADDR) MEM_DESC_ADDR_KERNEL_ASSIGNED,
(PHYS_ADDR) 0,
0,
0
},

{
 (VIRT_ADDR) MEM_DESC_ADDR_KERNEL_ASSIGNED,
(PHYS_ADDR) 0,
0,
0
},

{
 (VIRT_ADDR) MEM_DESC_ADDR_KERNEL_ASSIGNED,
(PHYS_ADDR) 0,
0,
0
}
};
LOCAL int sysPhysRamDescCount = NELEMENTS(sysPhysRamDesc);

#endif /* INCLUDE_CACHE_DMA32_LIB */

LOCAL PHYS_MEM_DESC sysDma32MemDesc[] =
{
 /* if 32-bit DMA is needed, this entry is used */

{
 (VIRT_ADDR) MEM_DESC_ADDR_KERNEL_ASSIGNED, /* kernel assigned */
(PHYS_ADDR) 0, /* computed at runtime */
0, /* computed at runtime */
VM_STATE_MASK_FOR_ALL,
#if ((USER_D_CACHE_MODE & CACHE_SNOOP_ENABLE) == CACHE_SNOOP_ENABLE)
VM_STATE_FOR_MEM_OS
#else
VM_STATE_FOR_IO
#endif /* (USER_D_CACHE_MODE & CACHE_SNOOP_ENABLE) == CACHE_SNOOP_ENABLE */
}
};
LOCAL int sysDma32MemDescCount = NELEMENTS(sysDma32MemDesc);

#endif /* INCLUDE_CACHE_DMA32_LIB */

#ifdef INCLUDE_USER_RESERVED_MEMORY
/*
 * sysUserMemDesc is used to provide user reserved memory
 * if INCLUDE_USER_RESERVED_MEMORY is defined.
 */

LOCAL PHYS_MEM_DESC sysUserMemDesc[] =
{


Providing Architecture-Specific Cache and MMU Support

The following sections outline some of the specific BSP support required by some architectures. In addition to reviewing this section, be sure to read the applicable chapter of the VxWorks Architecture Supplement to find any other architecture-specific support you may need to provide.

Defining Buffers for Cache Read and Flush

Some architectures require the BSP to provide a buffer or a pointer to be used by architecture cache routines. For example, ARM and XScale BSPs must provide a buffer in virtual address space (sysCacheFlushReadArea) and define the ARMCACHE macro. Some PowerPC BSPs need to define a pointer (cachePpcReadOrigin) used for cache flush operations.

Defining and Initializing Cache Configuration Variables

In some PowerPC BSPs, you must define and initialize a set of global variables that describes the characteristics of the primary instruction (I) and data (D) caches (number of lines, align size). The exact names of the variables depend on the architecture variant, and hence the particular cache support routine in the architecture code being used. The variables to be defined are as follows:

```c
UINT32  ppcVariantICACHE_LINE_NUM;
UINT32  ppcVariantDCACHE_LINE_NUM;
UINT32  ppcVariantCACHE_ALIGN_SIZE;
```

Where Variant is the architecture variant, and can take one of the following values: 405, E500, or E200.

The variables can be initialized statically and at runtime in sysHwInit().
Providing Architecture-Specific Support for the MMU

Some architectures require the BSP to provide board-specific support for the architecture MMU routines. For example, ARM and XScale BSPs must define the ARMMMU macro. Some PowerPC BSPs need to define and initialize the sysBatDesc[] array. For more information, see the architecture information in the VxWorks Architecture Supplement.

Supporting L2 and L3 Cache (Optional)

Cache support is hardware dependent. For specific information regarding cache support for your target architecture, see the VxWorks Architecture Supplement.

Implementing L2 Cache Support

For BSP L2 cache support, the following routines must be supplied and added to sysCache.c (or sysL2Cache.c) in the BSP directory:

```c
STATUS sysL2CacheInit (void)   /* initialize L2 cache support */
```

This routine initializes the L2 cache support hooks to point to functions within sysL2Cache.c. For instance:

```c
_pSysL2CacheEnable = sysL2CacheEnable;
_pSysL2CacheDisable = sysL2CacheDisable;
_pSysL2CacheFlush = sysL2CacheFlush;
_pSysL2CacheInvFunc = sysL2CacheInvFunc;
```

These cache support hooks are utilized by all the architecture cache support modules (cacheArchLib.c, cacheAimxxxLib.c) for L2 cache handling. The sysL2CacheInit() routine should be called from sysHwInit(). For instance, in sysHwInit() in sysLib.c:

```c
#if defined(INCLUDE_CACHE_SUPPORT) 
&& defined(INCLUDE_CACHE_L2) 
&& defined(USER_L2_CACHE_ENABLE)
    sysL2CacheInit();
#else
    sysL2CacheDisable();
#endif
```

Implement the following four basic L2 cache routines in sysL2Cache.c. If only a single L2 cache type (data, for instance) is supported, then the cacheType parameter can be ignored by the routine implementation.

```c
LOCAL void sysL2CacheEnable /* Enable L2 cache */
{
    CACHE_TYPE cacheType /* cache type: INSTRUCTION_CACHE, DATA_CACHE */
}

LOCAL void sysL2CacheDisable /* Disable L2 cache */
{
    CACHE_TYPE cacheType /* cache type: INSTRUCTION_CACHE, DATA_CACHE */
}

LOCAL void sysL2CacheFlush /* Flush L2 cache */
{
    CACHE_TYPE cacheType /* cache type: INSTRUCTION_CACHE, DATA_CACHE */
}

void sysL2CacheInvFunc /* Invalidate L2 cache */
{
    CACHE_TYPE cacheType /* cache type: INSTRUCTION_CACHE, DATA_CACHE */
}```
Implementing L3 Cache Support

For BSP L3 cache support, the following routines must be supplied and added to `sysCache.c` (or `sysL3Cache.c`) in the BSP directory.

```c
STATUS sysL3CacheInit (void)
```

This routine initializes the L3 cache support hooks to point to functions within `sysL3Cache.c`. For instance:

```c
_pSysL3CacheInvalEnableFunc = sysL3CacheInvEnable;
_pSysL3CacheFlushDisableFunc = sysL3CacheFlushDisable;
```

These cache support hooks are utilized by all of the architecture cache support modules (`cacheArchLib.c`, `cacheAimxxxLib.c`) for L3 cache handling. The `sysL3CacheInit()` should be called from `sysHwInit()`. For example, in `sysHwInit()` in `sysLib.c`:

```c
#if (defined INCLUDE_CACHE_L3)
   /* initialize and enable the L3 cache */
   sysL3CacheInit();
#endif /* INCLUDE_CACHE_L3 */
```

Implement the following two basic L3 cache routines in `sysL3Cache.c`. If only a single L3 cache type (data, for instance) is supported, then the `cacheType` parameter can be ignored by the routine implementation.

```c
void sysL3CacheInvEnable /* invalidate and enable L3 cache */
(CACHE_TYPE cacheType /* cache type: INSTRUCTION_CACHE, DATA_CACHE */)
```

```c
void sysL3CacheFlushDisable /* flush and disable the L3 cache */
(CACHE_TYPE cacheType /* cache type: INSTRUCTION_CACHE, DATA_CACHE */)
```

Supporting Cache Locking (Optional)

For cache lock support, the following routines are optional and can be implemented in `sysCacheLockLib.c` and `sysCacheLockALib.s`, or another appropriate cache library.

The following code illustrates one possible set of routine prototypes. Because these routines are never called by the core VxWorks routines, you have some discretion in the design.

```c
STATUS sysCacheLock /* lock cache line(s) */
(CACHE_TYPE cacheType, /* cache type: INSTRUCTION_CACHE, DATA_CACHE */
 void * adrs, /* address */
 UINT32 bytes /* number of bytes */);
```

```c
STATUS sysCacheUnlock /* unlock the cache */
(CACHE_TYPE cacheType /* cache type: INSTRUCTION_CACHE, DATA_CACHE */);
```

```c
STATUS sysL2CacheLock /* lock L2 cache line(s) */
(CACHE_TYPE cacheType, /* cache type: INSTRUCTION_CACHE, DATA_CACHE */
 void * adrs, /* address */
 size_t bytes /* number of bytes */)
```
Defining the Cache and MMU Configuration

The following sections describe the CDF components, CDF parameters, and preprocessor macros that are typically involved in configuring the cache and MMU in the BSP.

CDF components and parameters are defined in CDF specifications (installDir/vxworks-6.x/target/config/comps/VxWorks). These definitions can be overridden with CDF specifications in the BSP directory, typically found in 20bsp.cdf, or another .cdf file. This type of override only affects VxWorks Image Project (VIP) builds.

Pre-defined components can be pulled into build images by referring to them in #define statements in sysLib.c or another file included by sysLib.c (typically, config.h). This causes the component to be added to VIPs and command-line builds. For example:

```
#define INCLUDE_CACHE_SUPPORT /* add INCLUDE_CACHE_SUPPORT component */
```

Similarly, pre-defined CDF parameters can be overridden in the BSP code by re-defining them sysLib.c, config.h, or other file included by sysLib.c). For example:

```
#undef USER_I_CACHE_MODE
#define USER_I_CACHE_MODE CACHE_WRITETHROUGH
```

There are some cases (as with L3 cache support) where no CDF components or parameters are defined to handle the configuration of the feature. In these cases, there are preprocessor macros that are defined and used for this purpose.


Defining the Cache Configuration

This section lists the VxWorks CDF components, CDF parameters, and preprocessor macros that are used to enable and configure cache support in the BSP.

INCLUDE_CACHE_SUPPORT (Component)

Definition:
Includes cache support for OS and drivers (cacheLib).

Associated Parameters:
USER_D_CACHE_MODE, USER_I_CACHE_MODE

In the BSP:
If you want to include cache support in the BSP, define INCLUDE_CACHE_SUPPORT in config.h. Conditionalize code that initializes and supports the primary and secondary caches on this macro.
USER_D_CACHE_MODE (Parameter)

Definition:
Defines the cache write policy supported by the data cache. Cache policy masks are defined in cacheLib.h. They describe attributes such as writethrough, copyback, and write-allocate.

In the BSP:
Define USER_D_CACHE_MODE in config.h according to the supported data cache mode. Use the USER_D_CACHE_MODE macro to conditionalize code that needs to take the D-cache mode into account.

Note that this parameter is also passed to vmGlobalMapInit() to set the default data cache mode for the global kernel MMU mappings (sysPhysMemDesc[]).

USER_I_CACHE_MODE (Parameter)

Definition:
Defines the cache mode(s) supported by the instruction cache. Because instruction caches are inherently read-only, this parameter only serves to enable instruction cache support code.

In the BSP:
If your hardware supports an instruction cache, define USER_I_CACHE_MODE to something other than CACHE_DISABLED.

INCLUDE_CACHE_ENABLE (Component)

Definition:
Enables cache support. This component is included in VxWorks Image Projects (VIPs) by most profiles, including the default profile. It causes cacheEnable() to be called for the I-cache and/or D-cache, depending on the value of the parameters USER_I_CACHE_ENABLE and USER_D_CACHE_ENABLE.

Associated Parameters:
USER_I_CACHE_ENABLE, USER_D_CACHE_ENABLE

In the BSP:
This component is used for project builds only, and is managed completely outside of the BSP itself. Therefore, nothing needs to be done with this component in the BSP.

USER_I_CACHE_ENABLE (Parameter)

Definition:
Enables instruction cache support.

In the BSP:
If you want to enable the instruction cache in the BSP, define USER_I_CACHE_ENABLE in config.h. Conditionalize code that initializes and supports the instruction cache based on this macro.

USER_D_CACHE_ENABLE (Parameter)

Definition:
Enables data cache support.
In the BSP:
If you want to enable the data cache in the BSP, define
`USER_D_CACHE_ENABLE` in `config.h`. Conditionalize code that initializes and supports the data cache based on this macro.

**INCLUDE_CACHE_ARCH (Component)**

Definition:
Defines architecture-level support for cache. This component is defined by an architecture-specific CDF in:

```
installDir/vxworks-6.8/config/comps/vxWorks/arch/arch
```

which pulls in generic cache support for the given architecture. Some architectures also define variant-specific cache architecture components, such as:

```
INCLUDE_CACHE_ARCH_ARMARCH6
INCLUDE_CACHE_ARCH_BCM125X
INCLUDE_CACHE_ARCH_PPC60X
```

These components serve a similar purpose to `INCLUDE_CACHE_ARCH`, except they define architecture support for architecture variants.

In the BSP:
Your BSP may optionally re-define `INCLUDE_CACHE_ARCH` to pull in architecture-variant cache support. Alternately, if an architecture-variant cache support component is already defined, (such as `INCLUDE_CACHE_ARCH_BCM125X`), your BSP should instead REQUIRE this component in its definition of `INCLUDE_CACHE_BSP`.

**INCLUDE_CACHE_BSP (Component)**

Definition:
Defines BSP-level support for cache.

In the BSP:
The BSP may override this component with a `REQUIRES` clause that pulls in the architecture variant cache support that is appropriate for the given board. For an example, see the `20bsp.cdf` file in any of the following BSPs:

```
wrPpmc74xx
integrator1136jfs
arm_ebmpcore
sb1480_0_mipsi64
```

**INCLUDE_CACHE_L2 (Component, PowerPC only)**

Definition:
Includes L2 cache support. (In VxWorks 6.8 and prior, this only applies to the PowerPC architecture.) This component is defined in the architecture-specific CDF for PowerPC:

```
installDir/vxworks-6.8/config/comps/vxWorks/arch/ppc/arch.cdf
```

In the BSP:
If the BSP or architecture supports an L2 cache, define `INCLUDE_CACHE_L2` in `config.h`. Conditionalize code that initializes and supports the L2 cache on this macro.
**INCLUDE_CACHE_L2_ENABLE (Component, PowerPC only)**

**Definition:**
Provides a parameter to enable L2 cache support. (In VxWorks 6.8 and prior, this only applies to the PowerPC architecture.) This component is defined in the architecture-specific CDF for PowerPC:

```
installDir/vxworks-6.x/config/comps/vxWorks/arch/ppc/arch.cdf
```

**Associated Parameters:**
**USER_L2_CACHE_ENABLE**

**In the BSP:**
There are no requirements for the BSP. The L2 cache support is enabled using the associated parameter, `USER_L2_CACHE_ENABLE`.

**USER_L2_CACHE_ENABLE (Parameter, PowerPC only)**

**Definition:**
Enables L2 cache support. (In VxWorks 6.8 and prior, this only applies to the PowerPC architecture.) This component is defined in the architecture-specific CDF for PowerPC:

```
installDir/vxworks-6.x/config/comps/vxWorks/arch/ppc/arch.cdf
```

**In the BSP:**
If the BSP or architecture supports an L2 cache, define `USER_L2_CACHE_ENABLE` in `config.h`. Conditionalize code that initializes the L2 cache on this macro.

**INCLUDE_L2_CACHE (Macro, PowerPC only)**

**Definition:**
Provides an alternate way to include support for the L2 cache in some BSPs.

**In the BSP:**
If you want to support an L2 cache in your BSP, define the `INCLUDE_L2_CACHE` macro in `config.h`. Use this macro in your BSP code to conditionalize the initialization and support of the L2 cache.

**INCLUDE_L3_CACHE (Macro, PowerPC only)**

**Definition:**
Includes support for the L3 cache.

**In the BSP:**
If you want to support an L3 cache in your BSP, define the `INCLUDE_L3_CACHE` macro in `config.h`. Use this macro in the BSP code to conditionalize the initialization and support of the L3 cache.

**CACHE_LIB_INIT_FUNC (Parameter, MIPS only)**

**Definition:**
The `sysCacheLibInit` function pointer (used by `cacheLib`) is initialized to this macro. It defaults to `cacheMipsLibInit`, setting up the cache handling to be initialized by `cacheMipsLib`. (See Initialize VxWorks Cache Support Layers, p.76.)
In the BSP:
If your BSP has specialized cache requirements not handled by
`cacheMipsLib`, then override this macro to the appropriate initialization
routine (for example, `cacheOcteonLibInit()` or `sysCacheSb1Init()`). (See
`installDir/vxworks-6.x/target/src/arch/mips`.)

**ROM_AUTO_CACHE_DETECT (Macro, MIPS only)**

Definition:
MIPS only: This macro enables auto-detection of cache attributes such as
number of lines and line size.

In the BSP:
Define `ROM_AUTO_CACHE_DETECT` if you want the boot loader code
(`romInit()`) to auto-detect cache hardware attributes using the CONFIG1
register. For details about how this is done, see `romCacheAutoDetect()` in:
`installDir/vxworks-6.x/target/src/config/mipsCommon/romMipsInit.s`

---

**Defining the MMU Configuration**

This section lists the VxWorks CDF components, CDF parameters, and
preprocessor macros that are used to enable and configure MMU support in the
BSP. (For more information, see the *VxWorks Kernel Programmer’s Guide: Memory
Management*.)

**INCLUDE_MMU_BASIC (Component)**

Definition:
Enables basic MMU support in the architecture and OS layers. This
component pulls in the `vmBaseLib` module. Full MMU support is
included when RTP support is enabled (`INCLUDE_RTP`).

Associated Parameters:
`VM_PAGE_SIZE`, `USER_I_MMU_ENABLE`, `USER_D_MMU_ENABLE`

In the BSP:
If you want to include basic MMU support in your BSP, define
`INCLUDE_MMU_BASIC` in `config.h`. Conditionalize BSP code that
supports the MMU on this macro.

**VM_PAGE_SIZE (Parameter)**

Definition:
Virtual memory page size—the number of bytes in memory pages
described by MMU page table entries.

In the BSP:
You can optionally override the value in `config.h`. All architectures have a
default value already defined in the following files.

For project builds:
`installDir/vxworks-6.x/target/config/comps/src/configAll.h`

or for command-line builds:
`installDir/vxworks-6.x/target/config/all/configAll.h`
USER_I_MMU_ENABLE (Parameter, PowerPC only)

Definition:
Enables instruction MMU handling in PowerPC architecture code. See:
installDir/vxworks-6.x/target/src/arch/ppc/mmu*

In the BSP:
To enable the instruction MMU, define this macro in config.h.

USER_D_MMU_ENABLE (Parameter, PowerPC only)

Definition:
PowerPC only: Enables data MMU handling in PowerPC architecture code. See:
installDir/vxworks-6.x/target/src/arch/ppc/mmu*

In the BSP:
To enable the data MMU, define this macro in config.h.

INCLUDE_BOOT_MMU_BASIC (Component)

Definition:
Includes the INCLUDE_MMU_BASIC component in a boot application (PROFILE_BOOTAPP) boot loader.

Associated Parameters:
BOOT_DISABLE_MMU_BEFORE_ENTRY

In the BSP:
If you want your boot application boot loader to include INCLUDE_MMU_BASIC, define INCLUDE_BOOT_MMU_BASIC in config.h.

Note that this is needed in rare cases only.

BOOT_DISABLE_MMU BEFORE_ENTRY (Parameter)

Definition:
Disables the MMU prior to jumping to the entry point of a boot application image. The default is TRUE.

In the BSP:
To leave the MMU enabled prior to jumping to the boot application image, override the value to FALSE.

INCLUDE_PAGE_SIZE_OPTIMIZATION (Component)

Definition:
Enables page size optimization support. This allows more efficient management of virtual memory with fewer translation table walks or translation looksaside buffer (TLB) misses. For more information, see the reference entries for vmBaseLib and vmPageOptimize( ).

In the BSP:
To enable MMU page optimization, define this macro in config.h.
3. Porting a BSP to Custom Hardware
  3.3 Finalizing Your BSP

INCLUDE_MAPPED_KERNEL (Component, MIPS only)

Definition:
Builds the VxWorks image to run in mapped virtual memory space. (See also SW_MMU_ENABLE.)

In the BSP:
Conditionalize code and data that describes the memory map and MMU handling based on this component. This component is used in many MIPS BSPs. For an example of how this component is used, see an existing MIPS BSP (such as cav_cn3xxx_mipsi64r2sf). Do not define this parameter in config.h. It should be defined at build time only.

SW_MMU_ENABLE (Parameter)

Definition:
Enables software emulation of an MMU. This allows processors that have no MMU or systems running with the MMU unit disabled to execute RTPs. It does not allow arbitrary address translation, invalid page, protection, or cache attribute settings. Only identity mapping (virtual address == physical address) is possible.

In the BSP:
If your board does not have an MMU—or you want to run with the MMU disabled—and still want to support RTPs (INCLUDE_RTP), override this parameter to TRUE.

For MIPS, this parameter is defined to be TRUE by default. If INCLUDE_MAPPED_KERNEL is defined by the build, the BSP redefines SW_MMU_ENABLE to be FALSE.

INCLUDE_MMU_MPU (Macro, ARM only)

Definition:
Enables MMU support through the MPU, not through a full page-table-style MMU.

In the BSP:
Define INCLUDE_MMU_MPU for MMU support using the MPU. Conditionalize BSP MPU support code on this macro. For an example, see one of the following BSPs: integrator1136jfs, mrvl_db88f5181_mpu, ti_dm355evb, or ti omapl137evm.

INCLUDE_AIM_MMU_CONFIG (Component)

Definition:
Enables the option to configure the MMU AIM using the INCLUDE_AIM_MMU_PT_PROTECTION and INCLUDE_AIM_MMU_MEM_POOL_CONFIG components and associated parameters.

In the BSP:
To configure the MMU AIM memory in your BSP, optionally define INCLUDE_AIM_MMU_CONFIG.

This configuration is not required if the MMU AIM defaults are acceptable.
INCLUDE_AIM_MMU_PT_PROTECTION (Component)

Definition:
Configures the MMU AIM to write-protect page tables.

In the BSP:
To always protect page tables, optionally define
INCLUDE_AIM_MMU_PT_PROTECTION.

INCLUDE_AIM_MMU_MEM_POOL_CONFIG (Component)

Definition:
Enables the option to set the initial page table size parameters for the
MMU AIM.

Associated Parameters:
AIM_MMU_INIT_PT_NUM, AIM_MMU_INIT_PT_INCR,
AIM_MMU_INIT_RT_NUM, AIM_MMU_INIT_RT_INCR

In the BSP:
If you want to set the initial page table parameters for the MMU AIM,
define INCLUDE_AIM_MMU_MEM_POOL_CONFIG and the four
associated parameters.

AIM_MMU_INIT_PT_NUM (Parameter)

Definition:
Size (in MMU pages) of the initial memory allocated for page tables.

In the BSP:
Define AIM_MMU_INIT_PT_NUM to the appropriate value in config.h.

AIM_MMU_INIT_PT_INCR (Parameter)

Definition:
Size (in MMU pages) of the additional memory allocated when more
memory is needed for page tables.

In the BSP:
Define AIM_MMU_INIT_PT_INCR to the appropriate value in config.h.

AIM_MMU_INIT_RT_NUM (Parameter)

Definition:
Size (in MMU pages) of the initial memory allocated for region tables.

In the BSP:
Define AIM_MMU_INIT_RT_NUM to the appropriate value in config.h.

AIM_MMU_INIT_RT_INCR (Parameter)

Definition:
Size (in MMU pages) of the additional memory allocated when more
memory is needed for region tables.

In the BSP:
Define AIM_MMU_INIT_RT_INCR to the appropriate value in config.h.
INCLUDE_MMU_ARCH (Component, ARM only)

Definition:
Enables generic MMU support for the ARM architecture.

In the BSP:
To pull in generic MMU support for ARM, define INCLUDE_MMU_ARCH.

INCLUDE_MMU_ARCH_ARM1136 (Component, ARM only)

Definition:
Enables MMU support for the ARM 1136JF architecture variant.

In the BSP:
To pull in ARM 1136JF MMU support, define INCLUDE_MMU_ARCH_ARM1136.

INCLUDE_MMU_ARCH_PPC60X (Component, PowerPC only)

Definition:
Includes PPC60x MMU function libraries.

In the BSP:
To pull in the PPC60x MMU function libraries, REQUIRE the
INCLUDE_MMU_ARCH_PPC60X component in the INCLUDE_MMU_BSP
component definition in the BSP.

INCLUDE_MMU_BSP (Component)

Definition:
Defines BSP-level support for the MMU.

In the BSP:
Your BSP may define this component with a REQUIRE clause that pulls
in the architecture-variant MMU support that is appropriate for the given
board. For an example, see 20bsp.cdf in one of the following BSPs:
wrPpmc74xx, integrator1136jfs, arm_ebmpcore, or mv6100.

INCLUDE_MMU_P6_32BIT and INCLUDE_MMU_P6_36BIT (Macros)

Definition:
IA-32 Pentium II, Pentium III, and Pentium 4 CPU types only: Enables
enhanced 32-bit (or 36-bit) MMU support. For more information, see the
target.ref file for the pcPentium4 BSP.

In the BSP:
Define one of these macros in config.h to enable 32- or 36-bit MMU
support. For more information, see the target.ref file for the pcPentium4
BSP.

Additional Information

For more information on supporting the cache and MMU, see the following
documentation:

- VxWorks Architecture Supplement
- VxWorks Kernel Programmer’s Guide: Memory Management
3.3.6 Testing Boot ROMs

Boot ROMs use the VxWorks kernel. The main differences between the bootrom image and the vxWorks image are:

- The bootrom image uses the following file to initialize the kernel just like a standard VxWorks image:

  installDir/vxworks-6.x/target/config/all/usrConfig.c (for BSP directory builds only)

  The final image is linked at a different address so that the boot ROM executes out of a different address space. Getting the boot ROM working involves essentially the same the steps described previously for the vxWorks image. The bootrom image includes the BOOTAPP components which add the boot shell and image booting functions.

  The following features are not compatible with the bootrom image:

  - INCLUDE_SHELL
  - INCLUDE_WDB_SHELL
  - INCLUDE_BANNER

  In addition, not all of the networking functionality is available to boot loaders. The FTP, TFTP, and RSH features are available to standard boot ROMs. To include other networking features, the boot loader must be created in Workbench using the PROFILE_BOOTAPP configuration profile.

  The bootrom image is compressed by default. It is decompressed and copied into RAM in the bootInit.c file.

  During cleanup, each variation of the bootrom configuration should be tested.

3.3.7 Providing Bus Interface Support

Bus configuration is often required before devices on the bus can be supported. This section specifies how to configure several bus types (for example, PCI and USB).

After the bus is configured, the devices on the bus can be configured.
VxBus Support

This section briefly discusses VxBus support. For more information on VxBus, see *VxBus Device Driver Developer’s Guide: Device Driver Fundamentals*.

For support of a specific bus type, VxBus-compliant BSPs must reference code that supports the bus type, such as PCI, and they must reference a device driver for the bus controller providing access to the bus. In general, this should be considered device driver development, rather than BSP development. When a driver for a new bus controller is developed, the driver should be made generic enough to be used for other boards that use that bus controller, and put in a driver directory in the source tree rather than in the BSP.

PCI Bus Interface Support

For information on VxBus and bus support, see *VxBus Device Driver Developer’s Guide: Development Strategies*.

VME Bus Interface Support

To include support for a VME bus interface, you must determine which VME bus controller is used on your target system and include a driver for that controller. The drivers for most standard VME bus controllers are located in:

```
installDir/vxworks-6.x/target/src/drv/vme
```

If the drivers in that directory are not applicable to your target system, you may need to provide your own driver. In this case, the template VME driver (`templateVme.c`) can be used as a starting point for a custom driver. However, if one of the standard drivers provided is very similar to the device you are using, you may want to use that driver as a starting point. Your reference BSP may also include VME support for your device (or a similar device). If this is the case, you can use the custom VME driver included in the reference BSP.

In addition to the VME bus controller driver, you must configure the addresses to use for the A16, A24, and A32 address spaces on the VME bus. Normally, the macros `VME_A32_MSTR_LOCAL`, `VME_A32_MSTR_SIZE`, `VME_A24_MSTR_LOCAL`, `VME_A24_MSTR_SIZE`, `VME_A16_MSTR_LOCAL`, and `VME_A16_MSTR_SIZE` must be defined. However, this requirement can vary depending on the specific VME controller driver used. Other configuration—such as data width, cycle type, and other VME-specific details—may also be required. Refer to the VME bus controller driver for information on exactly what routines must be called during initialization and what macros must be defined.

The `sysBusTas()` routine is required by the BSP. For boards with a VME bus and no other bus containing memory, the `sysBusTas()` routine is provided by the VME bus controller. However, for more complex buses, a custom version of this routine may be required. In the case of a single board computer, or other simple board without a complex bus structure, `nullVme.c` can be used to include `sysBusTas()` and other utility stub routines.

**NOTE:** Wind River does not currently offer the kind of plug-and-play support published in the VME-64 specification and its extensions.
USB Bus Interface Support

VxWorks support for USB is provided as an optional product that supports USB either as a host or as a device. For current information, refer to the Wind River USB documentation and your product release notes.

3.3.8 Updating BSP-Specific Documentation

A BSP is not generally considered complete until the BSP-specific user documentation file, target.ref, is updated. As a general rule, an updated target.ref file should be available before the BSP is used for any application development. For target hardware that is developed for “in-house” or internal use only, the documentation may be required early in the development process.

Once the core BSP is working and contains the bus support that is required for additional device drivers, it is good idea to update or create the target.ref file. The documentation file can also include information for any expected optional drivers and their current level of support. As support for each driver is added and tested, the target.ref file should be updated as appropriate.

During application development, the target.ref file can be processed and made available from Workbench (in standard HTML format) by issuing a make man command from a command or shell window in the BSP directory. See the Wind River Workbench By Example (or, for Tornado users, the Tornado User’s Guide) for more information on command-line setup and capabilities.

For complete information regarding the target.ref file, see B. Implementing Documentation Guidelines.

Update Infrastructure Files

The vx_postinstall script dynamically updates the online help table of contents with new BSPs. The script is located in installDir/vxworks-6.x/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.

3.3.9 Providing Additional Optional Device Support

Although not truly required for a functional VxWorks kernel, a BSP is not generally considered usable unless it supports, at a minimum, a system clock, a serial port, and an Ethernet interface. The routines required for the system clock and serial port(s) have already been discussed (see 3.2.10 Minimum Required Drivers, p.68, and 3.2.11 Serial Drivers, p.69, respectively). Support for the Ethernet interface and other optional drivers are discussed in the following sections.

Adding VxBus Devices

One of the goals of VxBus is to minimize the effort to port a driver to a BSP. In general, assuming you are using a well-written VxBus-compliant driver for a VxBus-compliant BSP (and where the device is available on a given bus type and the driver understands how to recognize the device on that bus type), the effort of
getting a driver to work should be no more than including the driver in the project configuration, or defining a macro to include the driver.

**Debugging VxBus Device Drivers**

When working with VxBus device drivers, start with the default register access functions provided by VxBus and avoid providing BSP-supplied access macros. If faster register access is required, add this support after the driver is working.

VxBus drivers can be initialized after the system has booted, when the debugger and other host tools are available. Because VxBus drivers use driver methods for initialization, rather than BSP-supplied tables, initialization of individual devices after the system has booted is usually relatively simple. The availability of the debugger and other host tools generally improves productivity enough to offset the effort to delay initialization.

When working with legacy drivers, the same condition holds: try to initialize the device after the system has booted, though some parts of the initialization may be more closely integrated with system startup. Therefore, it may be more difficult to initialize the device after the system has booted. However, use of the debugger and other host tools generally provides enough additional information that the effort to postpone initialization is worthwhile.

For more information about debugging device drivers, see the *VxBus Device Driver Developer’s Guide: Device Driver Fundamentals*.

**Adding Ethernet Devices**

The network stack currently requires that Ethernet devices be initialized by calling the `devEndLoad()` routine provided by each driver—for example, `fei82557EndLoad()` is provided by the fei enhanced network driver (END). The parameter to the `devEndLoad()` routine is a character pointer. In order for each Ethernet device to be controlled by a given driver, the `devEndLoad()` routine is called twice. The first time, it is called with a pointer to a zero-length string. At this time, the driver is required to copy its own device name into the string. The network stack then fills in the unit number and initialization string from a table provided by the BSP and calls the `devEndLoad()` routine again. This time, the driver initializes the device and prepares itself for operation.

For this process to work, the BSP must provide a table called `endDevTbl[]` containing the end load strings. This table is usually defined in `configNet.h` and is generally organized in one of two ways.

The simplest organization is that the table is initialized at compile time to contain the initialization strings for the supported devices. This is the most straightforward organization because it is a simple list of devices that are available. Also, the BSP puts the driver’s `devEndLoad()` routine into the appropriate field of `endDevTbl[]`, no additional code is required. The drawback to this method is that the system may print a warning message if any of the supported devices are not actually present when the system boots. Configuration of the initialization string must be done statically at compile time using a constant string, rather than being configurable with descriptive macros.

A better organization is for `endDevTbl[]` to be allocated for several entries with the values for each entry being filled as each device is discovered at boot time. During the discovery phase of system initialization, the individual fields of the
initialization string are filled in according to the value of descriptive configuration options, and then the driver’s `devEndLoad()` routine is called. For this method to work, the BSP must provide a wrapper function for the driver’s load routine. A list of sample routines is as follows:

sysDec21x40EndLoad()
Creates a load string and loads a `dec21x40` (dc) device.

sysEl3c90xEndLoad()
Loads an instance of the `el3c90xEnd` driver.

sysEndLoad()
Creates a load string and loads the END devices.

sysFei82557EndLoad()
Loads a `fei82557` (fei) device.

sysLn97xEndLoad()
Creates a load string and loads a `ln97x` (lnPci) device.

sysMotCpmEndLoad()
Loads an instance of the `motCpmEnd` driver.

sysMotFecEndLoad()
Loads an instance of the `motFecEnd` driver.

**Media Access Controller (MAC) Address for Ethernet Devices**

Most modern Ethernet devices require the hardware address (MAC address) to be specified externally from the Ethernet controller chip. For peripheral boards, such as PCI network cards, the MAC address is supplied in ROM on the board and configured into the Ethernet controller during board power-up.

However, for processor boards with on-board Ethernet interfaces, the MAC address typically must be set by software. In most cases, the board is designed so that either three or six octets of non-volatile memory (ROM or flash) are reserved for the MAC address. The MAC address is read from ROM and programmed into the Ethernet device. If six octets are stored in ROM, they are used as the MAC address. Otherwise, the manufacturer’s organizationally unique identifier (OUI) is used for the first three octets, and the value in ROM is used for the last three octets.

However, this mechanism is not supported on all boards or by all BSPs. Instead, some boards and BSPs use a constant for the MAC address. In this situation, only one board of a given type can be used on a given subnet. If multiple boards are used, the resulting address conflicts can disrupt systems on the entire subnet. To avoid this issue in your BSP, set the lower three octets of the MAC address to the lower three octets of the device’s IP address.

**Adding Other Devices**

Similar to the Ethernet END driver load routines described in *Adding Ethernet Devices*, p.97, support routines required by drivers are considered optional to the BSP. The following list is a small sample of some of the other types of device initialization routines that can be included in a BSP:

sysAtaInit()
Initializes the EIDE/ATA interface.
sysIbcInit()
   Initializes the ISA Bridge Controller (IBC).

sysL2CacheInit()
   Initializes the L2 cache.

sysScsiInit()
   Initializes an on-board SCSI port.

sysTffsInit()
   Performs board-level initialization for TrueFFS.

There may also be custom hardware that must be supported in order for the BSP to be useful. The custom hardware support you require may not be provided in the reference BSP and may not be similar to other device support provided by Wind River. If this is the case, you must design the interface between the driver and the BSP.

For more information about adding support for devices, refer to the VxBus Device Driver Developer’s Guide.

### 3.3.10 Troubleshooting and Debugging

This section provides several suggestions for troubleshooting techniques and debugging shortcuts.

**SCSI Cables and Termination**

A poor cable connection or poor SCSI termination is one of the most common sources of erratic behavior, of the VxWorks target hanging during SCSI execution, and even of unknown interrupts. The SCSI bus must be terminated at both ends, but make sure that no device in the middle of the daisy chain has pull-up terminator resistors or any other form of termination.

**Data Coherency Problems**

Data coherency problems usually occur in hardware environments where the CPU supports data caching. First, disable the data caches and verify that data corruption is occurring. If the problem disappears with the caches disabled, the coherency problem is related to caches. (Caches can usually be turned off in the BSP by \#undef USER_D_CACHE_ENABLE.) In order to further troubleshoot the data cache coherency problem, use the cacheDmaMalloc() routine in the driver for all memory allocations. However, if hardware snooping is enabled then the problem may lie elsewhere.

**Data Address in Virtual Memory Environments**

If the CPU board has a memory management unit (MMU), the driver developer must be careful when setting data address pointers during direct memory access (DMA) transfers. When DMA is used in this environment, the physical memory address must be used instead of the virtual memory address. This is because during DMA transfers from the SCSI bus, the SCSI or DMA controller is the bus
master and therefore, the MMU on the CPU cannot translate the virtual address to the physical address. Instead, the macro \texttt{CACHE\_DMA\_VIRT\_TO\_PHYS} must be used when providing the data address to the DMA controller.
4

Adding BSP Support for VxWorks Features

4.1 Introduction

This chapter provides guidance for adding support for certain VxWorks features to your custom BSP. In some cases, you may have a custom BSP that you are trying to port to a more recent version of VxWorks and you would like to add support for a feature that is new with that version. Or, the current version of a BSP you want to use as a base for your custom design does not include support for a feature that you would like to implement. In both cases, you may find the information in this chapter useful.

4.2 Power Management

For information on power management, see the VxWorks Power Management Developer’s Guide.

4.3 PROFILE_BOOTAPP

The PROFILE_BOOTAPP configuration profile can be used with Workbench or the vxprj command-line facility to create a VxWorks boot loader. This method is an alternative to the legacy boot loader model that required making updates directly to config.h and building the boot loader image from the command line using
make. This section describes how to add support for PROFILE_BOOTAPP to an existing BSP. It also includes information on how to migrate boot loader customizations from an existing custom BSP (that does not implement PROFILE_BOOTAPP) and create an updated BSP that supports this configuration profile.

For information on developing a boot loader application and using PROFILE_BOOTAPP to create a boot loader, see the VxWorks Kernel Programmer’s Guide: Boot Loader.

### 4.3.1 Adding PROFILE_BOOTAPP Support to a BSP

Unlike the config.h and make method used to create a boot loader in legacy BSPs, the PROFILE_BOOTAPP configuration profile does not scan the config.h file in order to determine the components to add to the boot loader. Instead, a component description file (CDF) in the BSP directory is used to hold the component definitions required by PROFILE_BOOTAPP in order to create a boot loader application. This CDF requires a definition of the minimal components required by the BSP and a definition of the drivers required in the boot loader. Your BSP can also use this CDF to define the default memory settings for the BSP. (For more information on CDFs, see VxWorks Custom Component and CDF Developer’s Guide.)

The minimum required components are defined in the Bsp data structure. The REQUIRE line defines the minimal list of components that the BSP cannot properly work without. The following example shows a BSP that requires four components to work properly.

```plaintext
Bsp pcPentium4 {
   NAME board support package
   CPU PENTIUM4
   REQUIRES INCLUDE_KERNEL \
      INCLUDE_PCPENTIUM4_PARAMS \
      INCLUDE_MMU_P6_32BIT \
      INCLUDE_CPU_LIGHT_PWR_MGR
   FP hard
}
```

To define the drivers required by the boot loader, there are two component definitions that need to be modified by a BSP to add a list of drivers. The INCLUDE_BOOT_NET_DEVICES component defines the list of networking devices required. The INCLUDE_BOOT_FS_DEVICES component defines the list of file system devices required. These components are defined by default when creating a PROFILE_BOOTAPP VxWorks Image Project (VIP) and can be manually removed by editing the kernel configuration in Workbench.

```plaintext
/* Network Boot Devices for a BSP
 * The REQUIRES line should be modified for a BSP.
 */
Component INCLUDE_BOOT_NET_DEVICES {
   REQUIRES INCLUDE_FEI_END
}

/* Filesystem Boot Devices for a BSP
 * The REQUIRES line should be modified for a BSP.
 */
Component INCLUDE_BOOT_FS_DEVICES {
   REQUIRES INCLUDE_BOOT_FD_LOADER
}
```

A console port is not defined by default in all BSPs. The boot shell for PROFILE_BOOTAPP requires a console port in order to allow the user to edit the
Adding BSP Support for VxWorks Features

4.3 PROFILE_BOOTAPP

You can require a console port on your BSP for use with PROFILE_BOOTAPP by adding the following to the BSP CDF file:

```c
/* Specify boot rom console device for this BSP */
Component INCLUDE_BOOT_SHELL {
    REQUIRES += DRV_SIO_NS16550
}
```

The plus equals (+=) in the REQUIRES line adds the serial driver to the existing component requirements for the boot shell component. If the plus (+) is eliminated, the other required components are replaced by this single serial driver component, and the boot shell does not work properly.

Optionally, a BSP can set up the RAM_LOW_ADRS and RAM_HIGH_ADRS definitions in a CDF file. If these values are set up properly, the VIP is setup automatically when it is created. The proper definition looks something like the following:

```c
Parameter RAM_HIGH_ADRS {
    NAME Bootrom Copy region
    DEFAULT (INCLUDE_BOOT_RAM_IMAGE)::(0x01E00000) \n        (INCLUDE_BOOT_APP)::(0x02000000) \n        (0x01C00000)
}

Parameter RAM_LOW_ADRS {
    NAME Runtime kernel load address
    DEFAULT (INCLUDE_BOOT_RAM_IMAGE)::(0x01C00000) \n        (INCLUDE_BOOT_APP)::(0x01E00000) \n        (0x00100000)
}
```

**NOTE:** This construct is only evaluated when the VIP is created to determine default values. This definition does not work if the parameter is defined in the `config.h` file. Definitions in `config.h` override the parameter defaults in the CDF files.

The PROFILE_BOOTAPP RAM image (INCLUDE_BOOT_RAM_IMAGE) is a downloadable and executable image that is used to test and debug your boot application. It can be used to test the boot shell and the ability to load VxWorks images prior to committing the PROFILE_BOOTAPP boot loader to ROM.

The last and default memory address value for RAM_LOW_ADRS and RAM_HIGH_ADRS is for the VxWorks image (vxWorks), which is located in low memory. The PROFILE_BOOTAPP RAM image is then located above the VxWorks image in memory. The PROFILE_BOOTAPP image is located in the highest memory regions above the PROFILE_BOOTAPP RAM image and the VxWorks image. The top of memory for the PROFILE_BOOTAPP image, or the RAM_HIGH_ADRS for the boot application must be at least one megabyte below the top of memory. This one megabyte region is reserved for compressed ROM images. The decompress code is copied to the RAM_HIGH_ADRS memory location and executed there to decompress the boot code into the RAM_LOW_ADRS memory. Therefore, the one megabyte of memory above the INCLUDE_BOOT_APP RAM_HIGH_ADRS is reserved for decompress code.

You will have problems loading these images if RAM_LOW_ADRS and RAM_HIGH_ADRS are not defined correctly. For example, if the existing boot loader in the target system is executing in a memory location that the PROFILE_BOOTAPP RAM image will overlap when loaded, you will not be able to load and execute the PROFILE_BOOTAPP RAM image.
The order in which default values are defined for `RAM_LOW_ADRS` and `RAM_HIGH_ADRS` is important. The `INCLUDE_BOOT_RAM_IMAGE` address definition must precede the `INCLUDE_BOOT_APP` address definition. The reason for this is because `INCLUDE_BOOT_APP` is always defined when `INCLUDE_BOOT_RAM_IMAGE` is defined. The assignment of the `RAM_LOW_ADRS` and `RAM_HIGH_ADRS` default value is made on the first component match. If `INCLUDE_BOOT_APP` is listed first, then the `INCLUDE_BOOT_APP` address is always a match, and the address for `INCLUDE_BOOT_RAM_IMAGE` is never matched. Therefore, `INCLUDE_BOOT_RAM_IMAGE` must always be listed before `INCLUDE_BOOT_APP`.

**Loading the ROM Compress Image to the Boot Device**

To load a ROM compress image to your boot device, perform the following steps:

1. Set the project to build a ROM compress image.
   ```bash
   vxprj build set default_romCompress
   ```
2. Specify that a .bin file be built.
   ```bash
   vxprj build vxWorks_romCompress.bin
   ```
3. Follow `target.ref` instructions on how to turn the `vxWorks_romCompress.bin` file into a bootable `bootrom.sys` file.

**4.3.2 Migrating Boot Loader Customizations to PROFILE_BOOTAPP**

If you have a custom BSP that uses the legacy method (`config.h` and `make`) to support building a boot loader application, you may want to migrate your BSP to support `PROFILE_BOOTAPP`. To do this, you must do the following:

**Step 1: Remove bootConfig.c**

The `bootConfig.c` file is not used with `PROFILE_BOOTAPP`. In BSPs using legacy boot methods, the boot loader code gets configuration information from `bootConfig.c`, and the VxWorks image gets configuration information from `usrConfig.c`. With `PROFILE_BOOTAPP`, both the boot loader and the VxWorks image use `usrConfig.c`. When migrating your BSP to use `PROFILE_BOOTAPP`, any definitions you created in the `config.h` file that are specific to the boot loader must be moved into CDF files. The standard BSPs supplied with this release can be used as templates for making these changes.

**Step 2: Create the Bsp Data Structure**

You must create the `Bsp` data structure for your BSP in order to use `PROFILE_BOOTAPP`. To create the `Bsp` data structure, use the data structures in the standard BSPs provided with this release as examples.

For more information on the `Bsp` data structure, see 4.3.1 Adding `PROFILE_BOOTAPP` Support to a BSP, p.102.
Step 3: Create your CDF

In order for your BSP to support PROFILE_BOOTAPP, you must also create an appropriate CDF for your BSP. To create the necessary CDF, use the CDFs in the standard BSPs provided with this release as examples.

For more information on BSP CDF requirements, see 4.3.1 Adding PROFILE_BOOTAPP Support to a BSP, p.102. For general information on CDFs, see VxWorks Custom Component and CDF Developer’s Guide.
5.1 Introduction

This chapter presents common debugging methods used during BSP development. Many of these methods are described in earlier chapters as part of the development process. This chapter provides you with an overview of the debugging methods available and describes how and why each method is typically used.

There are three major stages of debugging during BSP development. They are: board bring up, creation of a downloadable image, and creation of a boot ROM image. Not all stages are necessary for all projects.

Board Bring Up

During board bring up, the target hardware is still under development. Because of this, some components of the board may not yet be included, and others may not be complete. The major aspects of board initialization are determined at this stage.

Once board bring up is complete, the hardware is fully stable. At this point, an OCD register file is typically available to initialize the board. This register file is used during subsequent BSP development.

Downloaded Image

When the board initialization sequence is determined and a register file is available, general BSP development begins. At this point, you can:

- Create an image on the host using your chosen compiler and tools.
- Download the resulting image to the target using the OCD device.
- Optionally, set a breakpoint before execution begins.
- Initiate execution of the image with the OCD device.
- Perform normal debugging.

**Boot ROM Image**

The BSP is not complete until it can execute from a system without the assistance of the OCD device. At some point, a boot image is loaded into flash. This image initializes the CPU and, typically, copies the final image to RAM for execution.

There are several issues associated with debugging a flash image with an OCD device, primarily related to setting breakpoints and to the location that the OCD device uses to obtain symbols for debugging purposes. These are discussed in 5.3 Applying Advanced Debugging Techniques, p.115.

### 5.2 Applying Basic Debugging Techniques

This section describes some basic debugging methods that can be used with readily available tools such as on-board LEDs, NVRAM, and ROM monitors. These methods do not rely on an on-chip debugging (OCD) solution. For information on debugging using an OCD, see 5.3 Applying Advanced Debugging Techniques, p.115.

#### 5.2.1 Using LEDs as a Diagnostic Tool

As outlined in 3.2.3 Using Debug Routines in the Initialization Code, p.58, it is usually best to start the development process by writing code to manipulate an LED device on your target hardware. If no LED is available, it may be possible to manipulate the state of an externally available pin or trace. In this case, you can connect a logic analyzer or simple oscilloscope to the pin in order to watch the state of the signal.

By writing LED code first, you validate that the image is created with consistent addresses. The ability to turn an LED on and off indicates that the code is loaded where it needs to be, and that the boot loader and the image agree.

If the LED does not operate as expected, you must verify that the image is loaded in the expected location. The following section presents several methods used to determine if the code is loaded in the proper place.

#### 5.2.2 Verifying the Image Location

This section discusses how to find addresses and use them to verify your image location in memory.

**Finding Addresses in the Image File**

The GNU binary utilities provided with your VxWorks installation include two command-line tools that are useful for finding the addresses of symbols and code. The tools are accessed using the commands `nm` and `objdump`, with an architecture
specification appended to the name—for example, `nmip` or `objdump`. For further information on these commands, see the GNU binary utility documentation or the online help.

**Finding Symbol Addresses**

You can use `nm` to find a list of symbols included in an image. The output includes the address that the symbol resolves to, or 00000000 if the symbol is unresolved.

**Finding Code Block Addresses**

You can use `objdump` when you need to find the code that resides at a particular RAM address. For example, the following command shows the entire contents of a VxWorks image file:

```
objdump --disassemble-all --show-raw-instr vxWorks
```

**Finding Addresses in the Flash Image**

A common problem with the development environment is that the process of programming the image into flash does not put the data at the correct location. One way to test this is to examine the hex file—that is, `bootrom.hex` or `vxWorks_rom.hex`, and the contents of memory, to insure that they match as expected.

Hex files are usually used in the process of creating a flash image or burning the image to ROM. If the process being used to create and burn the flash image does not involve a hex file, skip the remainder of this section.

The following is the extra build output generated when creating `bootrom.hex`. Notice that the utility used to perform the conversion from ELF to hex is `objcopy`.

**Tornado 2.2:**

```
installDir/host/hostType/bin/objcopy -O srec \n   --gap-fill=0 bootrom outtmpl1
installDir/host/hostType/bin/objcopy -O srec \n   --ignore-vma --set-start=0x0 outtmpl1
```

**VxWorks 6.x:**

```
installDir/workbench-2.x/hostType/bin/objcopy -O srec \n   --gap-fill=0 bootrom outtmpl1
installDir/workbench-2.x/hostType/bin/objcopy -O srec \n   --ignore-vma --set-start=0x0 outtmpl1
```

This hex file is really a Motorola S-Record file that is assumed to start at address 0. Using your flash programmer, you must perform whatever bias or offset is required so that this file starts at the reset vector. Consult your flash programmer documentation for specific details.

**NOTE:** When using the Wind River ICE 2 tools to perform the flash programming, the conversion is from a hex file to a bin file. The bin file that the Wind River tools generate is a special format specific to the Wind River ICE 2 tools and is not the more commonly recognized bin format. Therefore, use caution when sharing bin files between Wind River ICE 2 tools and other flash programmers, the formats are not likely to be compatible.
The following example shows some data from a bootrom.hex file:

```
S00E000626F6F74726F6D2E686578C
S2140000048000039600000004800003D436F7079EA
S21400000107269676874203139383420323031B7
S21400002057696E66405269766572205379737465D9
... ...
S2140366400000000E000000010000000F0000000024
S2140366500000000000000000000000000000032
S214036660001377080013770800000000000000000FE
S804000000FB
```

The following is a sample of what each line contains:

```
S 2 14 000000 48 00 00 39 60 00 00 48 00 00 03D 43 6F 70 79 EA
S
```

- **S** – indicates the file is an S-record file

In the second byte:

1 – indicates the file uses 16-bit addressing

2 – indicates the file uses 24-bit addressing

3 – indicates the file uses 32-bit addressing

The next two bytes are record length in hexadecimal.

14 – indicates 16 data bytes + 1 checksum byte + 3 address bytes for a total of 0x14 bytes.

The next few bytes provide the address at which the data is placed. How many bytes is determined by the second byte, as described above. In this case, the second byte is 2, indicating that the file uses 24-bit addresses, or six bytes in the file.

**000000** – indicates the address

The remainder of the line contains data.

There is a simple calculation that can be used to help check the validity of this file. Subtract the last address from the first address and get the size of the data to be programmed.

To obtain the first address, look at the second line in the hex file.

To obtain the last address, look at the second-to-last line in the hex file. The two lines of the sample file are presented below, with spaces inserted between the fields for clarity.

```
S 2 14 000000 48000039600000004800003D436F7079EA
S 2 14 036660 00137708013770800000000000000000FE
```

**036660** – indicates the starting address of the last data line of the hex file. However, the data on that line fills through the 0x3666f address. Therefore, the end of the data is at the 0x36670 address.

In this example, you have 0x036670 - 0x000000 = 0x036670. Notice that

**edata - romInit** (that is, 0x36670 - 0x000000) is 0x033670 from the above **nmarch -n** output.

To verify that this is correct, you can check the output of **sizearch**.

For example:

```
C:\T22\ppc\host\x86-win32\bin\sizeppc bootrom
```

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>186422</td>
<td>36410</td>
<td>38544</td>
<td>261376</td>
<td>3fd00</td>
<td>bootrom</td>
</tr>
</tbody>
</table>

5 Debugging Your BSP

5.2 Applying Basic Debugging Techniques

In this case, add the text and data sizes. The sum of the sizes should be identical to the size calculated for the hex file: 186422 + 36410 = 0x036670 - 0x000000.

The fact that these calculations come out the same gives the hex file some validity. It is always a good idea to double check this calculation if your image is not working as expected.

Another check is to use the `objdump -D` command to look at the disassembly of the executable. The following is a small example of `objdump -D bootrom`:

```
bootrom: file format elf32-powerpc
Disassembly of section .text:
00100000 <_romInit>:
100000: 48 00 00 39 bl 100038 <cold>
100004: 60 00 00 00 nop
100008: 48 00 00 3d bl 100044 <warm>
10000c: 43 6f 70 79 .long 0x436f7079
100010: 72 69 67 68 andi. r9,r19,26472
100014: 74 20 31 39 andis. r0,r1,12601
100018: 38 34 2d 32 addi r1,r20,11570
10001c: 30 30 31 20 addic r1,r16,12576
100020: 57 69 6e 64 rlwinm r9,r27,13,25,18
100024: 20 52 69 76 subfic r2,r18,26998
<rest of output cut off here>
```

Notice how the data in the above S-record matches up with the data in the disassembly.

48 00 00 39
60 00 00 00
48 00 00 3d
43 6f 70 79

compared to:

48 00 00 39 60 00 00 00 48 00 00 3D 43 6F 70 79

After programming the flash, read back the flash contents to confirm that the flash is programmed correctly. Always check this when bringing up the board for the first time, to confirm that the development processes are valid.

Looking at the disassembly of an object file is helpful when, given the address of an exception, you need to find out where the exception took place in your code. The following command is helpful in this situation:

```
objdump arch --disassemble-all --show-raw-instr tmp.o
```

Finding Addresses in RAM

The previous sections describe several ways to determine what should be in RAM. However, this information must be correlated with what is actually there. The easiest way to do this is to connect an on-chip debugging (OCD) device to your target, then set a hardware breakpoint at the RAM entry point, and use the OCD device to examine memory. Without the OCD device, determining what is in RAM is more difficult.

If a vendor-supplied ROM monitor is available, it may be possible to use the ROM monitor to load the application and then check the contents of RAM. If two flash banks are available, the VxWorks image can be programmed into the other flash bank and examined from the ROM monitor. Finally, if the ROM monitor provides hardware breakpoint support, it may be possible to set a hardware breakpoint at the RAM address and then start executing the VxWorks boot application. When
the breakpoint is reached and execution stops, the memory can be examined using the ROM monitor.

If no ROM monitor is available, it may still be possible to do some debugging. During the initial phases of BSP development, when the BSP consists only of LED code, the entire image may fit into a small bank of NVRAM. After the system boots, this memory can be removed from the system and read on another system.

5.2.3 Verifying RAM

The section discusses how to verify RAM during run-time execution.

At this point, you are building an image and programming the flash with it. The image is located in the correct area and you are ready to power on and run the boot ROM code. You now need a way to debug the run-time image. The following discussion assumes that you do not have an OCD device, but you do have at least one LED available on your target hardware or a port pin that can be connected to a logic analyzer.

One useful bit of code that you can add to the `usrConfig.c` file or the `bootInit.c` file are LED routines such as those presented in 3.2.3 Using Debug Routines in the Initialization Code, p.58. If you followed the advice in that section, which is recommended as the first step in creating a new BSP, the code verifies that the tools are working and also provides a debugging tool.

If the system does not have an LED available, an alternative is to set up the routines discussed in this section so that they change the status of a port pin that you can then read with a logic analyzer.

This type of debugging can be extremely difficult, and it is strongly encouraged that you have better tools, such as an OCD device. If you do use LEDs for debugging, you must be very creative to see what is happening in the system.

One routine missing from the LED library described earlier is a routine to blink the LED on and off. A sample routine is as follows:

```c
void ledBlink(int n)
{
    int i,j;
    /*
    * The 200000 value is picked to be around 1 second of delay. Adjust as needed.
    */
    sysLedInit();
    for(j=0; j < n; j++)
    {
        for(i=0; i < 200000; i++) sysLedOn();
        for(i=0; i < 200000; i++) sysLedOff();
    }
    /* just to create a pause between blinks */
    for(i=0; i < 200000; i++) sysLedOff();
}
```

The call to make is the `ledBlink()` call. A useful approach is to start off with `ledBlink(1)` and then use `ledBlink(2)` and so forth. This is an easy way to tell where the system is within the code as it is executing. This is also a powerful run-time tool that can get you to the point at which you see the VxWorks boot banner and menu system on the serial output device.

Once the VxWorks boot banner successfully appears, you should be able to use `printf()` routines or some of the debugging facilities already built into the boot
menu to help debug. Even before printf() routines are available, some debugging information can be output on the serial device after sysHwInit() is called. Add the following code to the BSP sysLib.c file. This code should work with any BSP that includes a serial driver that can be operated with polled mode output.

```c
void sysPrintDebug(char *msg)
{
    unsigned long msgIx;
    int pollStatus;
    for (msgIx = 0; msgIx < strlen(msg); msgIx++)
    {
        do
            pollStatus = sioPollOutput(sysSerialChanGet(0), msg[msgIx]);
        while (pollStatus == EAGAIN);
    }
}
```

To produce both carriage return and line feed, format your end lines with “\r\n”.

For example:
```
sysPrintDebug("Made it to sysHwInit2().\r\n");
```

### 5.2.4 Verifying the Image and OS Configuration

This section discusses how to confirm that VxWorks is properly configured and your image includes the proper VxWorks components.

#### Post-Processed Compiler Output

When building a VxWorks image, many compile-time macros are expanded, and it is sometimes difficult to know what the actual numeric values of these macros are and which branch of conditionally compiled code is being used. The bootrom code, in particular, contains many conditional compilations. One way to find this information is to retrieve the post-processed compiler output.

To retrieve the post-processor output for a given object module, use the following command:
```
make ADDED_CFLAGS=-E file.o > file.i
```

Next, remove all of the lines starting with # and all blank lines.

Then, check the code to see what sections have been included and what values are being used.

#### Operating System Components Built Into the Image

The first step in debugging is to make sure the executable image contains the operating system components you desire.

**NOTE:** You must set up the VxWorks environment variables in order to run the tools specified below. The easiest way to do this is to use the configuration script created during your installation. For VxWorks 6.x, the `wrenv` script is located in your `installDir`. For previous versions of VxWorks, this is the `installDir/`host`/hostType/bin/torVars` script.
If you are not sure if a particular block of code is included in the final image, you can use the `#warning` macro to determine what components are included at compile time. When the compiler’s preprocessor encounters the `#warning` macro in an area of code included in the compilation, it prints the message following the the `#warning` keyword. For example, to see if memory auto-sizing is enabled, the code in `sysPhysMemTop()` in `sysLib.c` can include the following:

```c
#ifdef LOCAL_MEM_AUTOSIZE
    /* To Do Auto-sizing stuff */
    /* This BSP does not support auto-sizing */
    #warning We should NOT do auto-sizing
#else /* not LOCAL_MEM_AUTOSIZE */
    /* Don’t do auto-sizing, use defined constants. */
    #warning We should be here.
    sysPhysMemSize = (char *)(LOCAL_MEM_LOCAL_ADRS + LOCAL_MEM_SIZE);
#endif /* LOCAL_MEM_AUTOSIZE */
```

Upon building the code, you may see output similar to the following:

```
ccppc -mcpu=603 -mstrict-align -ansi -O2 -fvolatile -fno-built-in \-Wall -T/h -I. -IC:\T22\ppc\target\config\all \-IC:\T22\ppc\target/h -IC:\T22\ppc\target/src/config \-IC:\T22\ppc\target/src/drv -DCPU=PPC603 -DTOOL_FAMILY=gnu \-DTOOL=gnu-c sysLib.c
```

This method is helpful when you are not sure if a component is being included or not. Another option is to generate the post-processed compiler output and examine it (for more information on this method, see Post-Processed Compiler Output, p. 113). This is helpful to confirm that an expected include file is being picked up before another include file.

Another way to see if components are included is to look at the `nmarch -n` output. For example:

```
installDir/vxworks-6.x/target/config/wrSbc824x> nmppc -n bootrom
00100000 T _romInit
00100000 T _wrs_kernel_text_start
00100000 T _romInit
00100000 T _wrs_kernel_text_start
00100038 t cold
00100044 t warm
00100048 t start
001000a4 t ifpdr_value
00100230 t romInit824x
00100680 t romValidateTLPs
001006e4 t tlbloop
001006a8 t romMinimumBATsInit
0010073c t gcc2_compiled.
0010073c T _romStart
00100888 t copyLongs
0010093c t fillLongs
00100964 t gcc2_compiled.
00100964 t gcc2_compiled.
00100964 t memcpy
001009bc t bzero
00100a04 t adler32
```

5.3 Applying Advanced Debugging Techniques

This section presents the more advanced debugging techniques available to a BSP developer.

NOTE: The methods discussed in this section require an on-chip debugging (OCD) device. For more information on basic debugging techniques, see 5.2 Applying Basic Debugging Techniques, p. 108.

5.3.1 Symbols

When the OCD device is connected to the target board for debugging, the normal procedure is to load both the image and symbols from a file on the development host then start execution of the image. However, during the boot ROM stage of BSP development, it may be necessary to debug the image exactly as-is, without the additional processor initialization that the debugger requires.

To accomplish this, the boot image is programmed into flash memory on the target, but a copy is kept on the host for use during debugging. The OCD device reads the copy on the host in order to obtain symbol information. Care should be taken to insure that the copy of the image resident on the host matches the copy programmed into flash.

Because this is a bootrom image, the output is not very interesting. To see the symbols in the actual VxWorks image that is used by bootrom, look at the tmp.o image instead of the bootrom image. If bootrom_uncmp is the target image, the image contains the full information and no tmp.o is available.
5.3.2 Breakpoints

An OCD device provides debugging support for all stages of BSP development, using methods similar to a standard application debugger.

For architecture-specific information on breakpoints, including limitations and usage caveats, see the VxWorks Architecture Supplement.

Types of Breakpoints

To make full use of the OCD device’s debugging capability, it is necessary to understand the nature of breakpoints.

There are two kinds of breakpoints available—software breakpoints and hardware breakpoints.

Software Breakpoints

The type of breakpoints typically used during application development are software breakpoints. With software breakpoints, the debugger takes an opcode from RAM and replaces it with a special instruction that causes a debug exception, and sets up a handler to be executed when the debug exception occurs. When the handler runs, it checks to see if the developer has a breakpoint set at that location at the time. If the developer does not have a breakpoint set, the handler executes the saved opcode and continues execution of the program. However, if the developer set a debugging breakpoint to stop the application from executing, the exception handler turns control over to the debugger. At this time, the developer can perform whatever action is required.

Hardware Breakpoints

Hardware breakpoints allow similar functionality to software breakpoints, but they do so without replacing the opcode as is done by software breakpoints. To use this type of breakpoint, the processor must support hardware breakpoints. Different processors support different hardware breakpoint functionality, but it is common for the hardware breakpoint support to be limited to one or two breakpoints.

Because the hardware breakpoints have access to the processor internals, they can also provide additional functionality that is not available with software breakpoints. For example, on some processors, it is possible to set a hardware breakpoint to cause a debug exception when a given memory location is modified, or even when a given memory location is read. This allows watchpoints and other capabilities without the cumbersome mechanisms required to implement that functionality using software breakpoints.

The Boot Procedure as It Relates to OCD

The nature of breakpoints is important when debugging BSPs. Recall that on powerup, the processor starts executing at the reset vector address, usually contained in flash. The code contained in the flash copies itself to RAM and begins execution. Often, the boot loader then loads another image into a different location in RAM and starts that image executing.
If you want to debug an image during boot, there are two different conditions to be aware of:

- code running directly from flash
- code that is copied into RAM and executed

**Initial Breakpoints in Flash Images**

When the code is running directly from flash, opcodes are fetched from flash and executed. Because it is not possible to modify the flash to insert the special opcode that is used to generate the debug exception, software breakpoints are not available.

Instead, if the processor supports it, you must set a hardware breakpoint. The hardware breakpoint causes the processor to raise a debug exception, and control is transferred to the debugger.

**Initial Breakpoints in Downloaded Images**

When debugging a downloaded image, software breakpoints can be used, but there is a condition on the use of software breakpoints. Recall that setting a software breakpoint causes a memory location to be modified by saving the opcode to a safe location and replacing it with a special opcode to generate the exception. However, if the image is copied into RAM after the breakpoint is set, the copy operation overwrites the special opcode, and the breakpoint is lost. For this reason, software breakpoints must be set after the image is loaded.

The standard boot method is to load an image and start execution immediately after loading it. However, this does not give you time to set any software breakpoints.

When an OCD device is used, there are three ways to set software breakpoints early in the boot sequence: separate load-and-go instructions, hardware breakpoints, and forever loops.

**Separate Load-and-Go Instructions**

Many boot loaders allow an image to be loaded but not run, in addition to the normal load-and-go command. With the VxWorks boot loader, the normal `@` command causes the image to be loaded and execution to begin. However, it is also possible to use the `I` command to load the image, and the `g` command to start execution. Most other boot loaders provide similar functionality, though the commands may be different.

In this case, the sequence is to start the boot loader and use it to load the VxWorks image. Next, use the OCD device to stop the processor and set the desired software breakpoints, then resume operation. At this point, the boot loader is again executing. Use the boot loader `go` command to start VxWorks. When the breakpoint is hit, control returns to the OCD device.

**Hardware Breakpoint**

If available, the OCD device can set a hardware breakpoint at the RAM entry address. The boot loader is then used to load the image into RAM and start execution. As soon as control is given to the VxWorks initialization code, the
hardware breakpoint is encountered and the OCD device takes control. At this point, additional breakpoints can be set using the OCD device—these breakpoints can be either software breakpoints or hardware breakpoints.

**Forever Loop**

In some cases, it is most convenient to load the image the normal way, using the boot loader load-and-go instruction, and to not use a hardware breakpoint. In this case, it is possible to put a simple infinite loop at the beginning of the VxWorks initialization code. After execution begins, the OCD device interrupts the processor and takes control. At this time, breakpoints can be set at the desired locations, and control can be returned to the VxWorks initialization code.
6.1 Introduction

This chapter describes how to extend an existing uniprocessor (UP) BSP to work with a VxWorks symmetric multiprocessing (SMP) system.

This chapter assumes that a uniprocessor BSP is available for the architecture and board you want to use. If you do not already have a UP BSP available for your target hardware, you must first develop and test a UP BSP for your board. Once you have a validated UP BSP, use this chapter to extend the BSP to support VxWorks SMP. (Uniprocessor BSP development is covered in the early chapters of this document. For more information, see 1. Introduction.)

This chapter also assumes that SMP support is available in the VxWorks architecture code that supports your processor. That is, VxWorks must already include SMP support for your target architecture. For information on architecture support for VxWorks SMP, see your Platform release notes.

NOTE: Code built for variants of VxWorks or for different releases of VxWorks is not binary compatible between variants or releases. Code must be built specifically for uniprocessor (UP) VxWorks, VxWorks SMP, 32-bit VxWorks, 64-bit VxWorks, or for a VxWorks system based on variant libraries produced with a VxWorks source build (VSB) project—or for any supported combination of these variants. The only exception to this rule is that RTP application executables can be run on both UP VxWorks and VxWorks SMP (when all else is the same).
6.2 About VxWorks SMP

VxWorks SMP is a configuration of VxWorks designed for use in a symmetric multiprocessing (SMP) system. The SMP VxWorks image provides the same general functionality as a uniprocessor (UP) configuration. With symmetric multiprocessing, one instance of the operating system controls multiple processing units. This differs from an asymmetric multiprocessing (AMP) system where a separate instance of the operating system is executing on each processing unit in the system (see 7. Extending a BSP to an AMP System).

SMP changes the conventional multitasking priority-based preemptive programming paradigm, in that it allows multiple threads to truly execute concurrently. This is because a single instance of the OS runs on multiple CPUs executing in parallel. When migrating code from a conventional multitasking system to an SMP system, this fundamental difference must be taken into account.

For more information on VxWorks SMP, see the VxWorks Kernel Programmer’s Guide: VxWorks SMP.

6.3 Development Overview

In developing a BSP to support VxWorks SMP, you need to address the following issues:

- Review and verify the VxWorks SMP development prerequisites. (See 6.4 Development Prerequisites, p.121.)
- Familiarize yourself with the CDF components and parameters used in SMP, and plan your use of those entities accordingly. (See Using Component and Configuration Preprocessor Macros, p.125.)
- Update and add component description file (CDF) code to support SMP. (See 6.6.2 Updating and Adding CDF Code, p.126.)
- Determine your boot loader support and strategy. (See 6.6.3 Implementing a Boot Loader Strategy, p.128.)
- Implement SMP boot logic, including managing secondary CPUs during reset and reboot. (See 6.6.4 Implementing SMP Boot Logic, p.129.)
- Configure and prioritize inter-processor interrupts (IPIs). (See 6.6.6 Prioritizing Interprocessor Interrupts (IPIs), p.133.)
- Modify cache handling. (See 6.6.7 Modifying Cache Handling, p.134.)
- Implement SMP BSP APIs. (See 6.6.8 Implementing SMP BSP Routines, p.135.)
- Configure VxBus. (See 6.6.9 Configuring VxBus (hwconf.c), p.141.)
- Update documentation to reflect new features. (See Step 4: Document Your Work, p.124.)
- Validate and test your BSP. (See Step 5: Validation and Testing, p.124.)
6.4 Development Prerequisites

Before beginning your VxWorks SMP BSP development, be sure to have the following items available:

- complete CPU and board documentation
- VxWorks SMP architecture support for your target CPU (this information is available in your Platform release notes)
- a working and validated uniprocessor (UP) BSP for your target hardware

6.4.1 Hardware Documentation

Before beginning your development, you must have complete documentation for your target board and processor. This documentation is typically available from the processor or board vendor.

Your hardware documentation should include the following critical information. If this information is not available, contact your processor or board vendor.

- target RAM and memory map information
- bootstrap and secondary CPU start and stop mechanisms and states

NOTE: VxWorks boot loaders typically do little or nothing with secondary CPUs (basically, keep them quiescent). This can pose challenges for BSP development. SMP requires configuring, starting, and stopping secondary CPUs in a controlled manner.

- CPU interrupt controller and interrupt routing information

NOTE: Implementation of the interrupt controller for SMP is an architecture-level issue and beyond the scope of this guide. However, the BSP is responsible for the routing and prioritization of interrupts. If you do not have this information, contact your hardware vendor.

- information on how to generate and prioritize interrupt controller inter-processor interrupts

6.4.2 Hardware Support

In order to run VxWorks SMP, your target hardware must provide the following hardware facilities:

multicore

The system must contain at least two processing units, capable of fully independent execution.

NOTE: You can run an SMP image configured for one CPU as a debug configuration.

shared memory

Each CPU must see the exact same physical memory subsystem (hence the term *shared memory*). Having shared memory implies that, when accessing
memory, code executing on the system does not need to consider which CPU is executing the code. That is, there is no memory that is local to a CPU as there is in a loosely coupled multiprocessing system.

**NOTE:** You can route interrupts to a CPU other than the boot CPU, if the hardware has the capability.

atomic operations
The CPUs must be able to synchronize with one another using atomic operations on memory locations. This capability is the basis upon which spinlocks are built. Spinlocks are an important synchronization and mutual exclusion mechanism in an SMP system. (For more information on spinlocks, see the VxWorks Kernel Programmer’s Guide: VxWorks SMP.)

cache coherency
Assuming each CPU has one or more caches dedicated to it, the hardware must maintain consistency between those caches. This is especially critical for data caches—less critical for instruction caches. If two data cache lines in different data caches on different CPUs are pointing to the same memory location, the hardware should manage the state of those cache lines so that each CPU gets correct values. If one CPU flushes a line to memory, then all other cache lines referencing the same location must be automatically updated (invalidated, or refreshed depending on the coherency algorithm).

a mechanism for inter-CPU interrupts
The processor must support the ability for one CPU to interrupt another CPU. This is generally accomplished through a global interrupt controller device (see Configuring Interrupt Controller Drivers, p.141). There must be a way to set inter-processor interrupts (IPIs) at a priority that is above all other interrupts.

Also, depending on your target hardware, you should have device access for all CPUs. That is, each CPU should have access to all devices, and you should be able to route interrupts from these devices to any one of the CPUs through a global programmable interrupt controller. (For more information on the global programmable interrupt controller, see Configuring Interrupt Controller Drivers, p.141.)

### 6.4.3 VxWorks SMP Architecture Support

SMP requires architecture-specific support at the OS level before it can be supported on a particular target architecture or processor. These changes, in particular CPU interrupt and exception management, must be implemented by Wind River at the OS level. This guide supports BSP development for target architectures that already include operating system support for VxWorks SMP. Implementing your own architecture support in the operating system is not supported by Wind River and is beyond the scope of this guide.

For information on target architecture support for VxWorks SMP, see your Platform release notes or the Wind River Online Support Web site.

### 6.4.4 Existing BSP Support

This document assumes that your SMP BSP development builds on a base uniprocessor (UP) BSP. You must develop and validate a UP BSP before attempting
to add multicore support. (For information on developing a UP BSP, see chapters 1-4 of this document).

SMP-enabled BSPs must be built using Workbench or the vxprj command-line facility (the legacy config.h build method using make is not supported for SMP BSPs). Therefore, you must be sure that your UP BSP is fully integrated with the Workbench and vxprj build system. This means you must have component configuration (using CDFs) in place before beginning your BSP development. For more information on CDFs, see the VxWorks Custom Component and CDF Developer’s Guide.

6.4.5 VxBus Support

If you are developing for a symmetric multiprocessing (SMP) system, Wind River recommends that you use VxBus-enabled device drivers. Wind River does not provide legacy model drivers that are SMP safe. If you want to use a legacy model device driver in an SMP system, you must ensure that the driver is SMP safe.

For information on SMP support, see the VxWorks Kernel Programmer’s Guide: VxWorks SMP. For more information on VxBus and device driver development, see the VxBus Device Driver Developer’s Guide.

6.5 Implementation Overview

This section provides an overview of the development process for developing an SMP BSP from an existing UP BSP. (The UP development process is described in the early chapters of this document.)

Step 1: Plan and Design Your SMP Strategy

Before writing any code, Wind River recommends that you are familiar with the overall BSP development process and that you read this chapter thoroughly, making notes about how you plan to address the key SMP BSP issues presented in this chapter. This includes developing a strategy for handling:

- boot loaders
- boot logic
- inter-processor interrupts (IPIs)
- cache issues
- SMP BSP APIs
- VxBus and VxBus driver configuration

Step 2: Create and Build a Baseline BSP

One of the first steps in VxWorks SMP BSP development is to establish a baseline BSP. This BSP provides you with a stable base for your SMP development work. Wind River recommends that you perform your development within a source code version control system. In the absence of such a tool, you can copy a reference BSP
into a new directory. The key is to have a clean base that you can return to as a reference. For more information on creating a baseline BSP, see 6.6.1 Creating the Initial BSP, p.125.

Step 3: Perform Iterative Development

When developing an SMP BSP, it is helpful to employ an iterative development process. Whenever possible, build and test often, adding a small amount of functionality with each iteration. At some stages of development, this is difficult to do, because several different parts are required to work together, or nothing works at all. Even so, the following basic engineering principles apply:

- Design first.
- Create a functioning VxWorks image as early as possible in the development process. Checkpoint your source code, then incrementally add functionality, re-test, and checkpoint again. Note that 6.6 Implementation Details, p.125 presents the required development tasks in an order that is likely to facilitate iterative development.
- Incorporate peer review and conduct unit testing often during development.
- Update your overall design as issues are resolved.

Step 4: Document Your Work

Part of the overall BSP development process involves documentation. You should verify manual pages are generated for new or significantly modified BSP routines. You should also update the BSP target.ref file with SMP-specific details. This usually involves a detailed discussion of the following items:

- boot mechanism changes
- additional or modified configuration parameters
- supported and unsupported devices in SMP
- caveats and restrictions
- known problems

Step 5: Validation and Testing

During the early stages of development, testing is (by necessity) mostly ad-hoc, using expected system behavior and console messages as an indication that the system is working. Later in the development cycle, at the point that you are able to boot a VxWorks SMP image and bring up the target shell, you can begin to employ the BSP Validation Test Suite (BSP VTS) to automate your testing. The BSP VTS supports SMP kernel testing by employing test cases that exercise and stress SMP functionality. In addition, the BSP VTS can run all applicable test cases with an affinity to each CPU in the SMP system.

As part of testing your BSP, be sure to run the BSP VTS in both UP mode (the default) and SMP mode (specify the -smp option switch). Note that it is also possible to run sub-sets of the full BSP VTS suite. For instance, first try running the basic test suite, followed by the arch and os suites.

For more information on the BSP VTS, see the VxWorks BSP Validation Test Suite User’s Guide. You may also access online help from the tool itself. To do this, enter the following in a VxWorks Development Shell:

```
$ vxtest suiteRun -help
```
6.6 Implementation Details

This section provides a more detailed look at the BSP development process for VxWorks SMP. As mentioned previously, this section assumes that you are starting with a working UP BSP.

6.6.1 Creating the Initial BSP

SMP BSPs are generally treated as an extension of an existing UP BSP. As a result, they share the same directory with the UP BSP. SMP-specific changes are conditionally included using `#ifdef _WRS_CONFIG_SMP`. If there is an existing SMP BSP for the same target architecture or similar processor, you may want to use that as a baseline, or at least as a resource for design and implementation details. Otherwise, pick the UP BSP that supports your target hardware as the baseline.

Wind River recommends that you isolate the changes that are made to support SMP on your target hardware. This can either be accomplished with a source code version control system, or by creating a separate directory for the new BSP. The isolation will help with regression testing and keeping track of exactly what changes have been made to the baseline BSP.

If you copy the baseline BSP to a new BSP directory, the `TARGET_DIR` macro in `Makefile` must be changed to refer to the leaf directory where the BSP lives. See the `target.ref` for your reference BSP for any additional required changes.

Using Component and Configuration Preprocessor Macros

The VxWorks build system employs two levels of conditional compilation using preprocessor macros. At the lowest level are the macros that are defined when the target libraries are built. In this document, these are referred to as configuration macros. Configuration macros are available to BSPs at project build time because the target library build process creates the following header files:

```
libDir/h/config/vsbConfig.h
libDir/h/config/autoconf.h
```

Where `libDir` is:

- `installDir/vxworks-6.x/target/lib` for 32-bit UP
- `installDir/vxworks-6.x/target/lib_smp` for 32-bit SMP
- `installDir/vxworks-6.x/target/lib_lp64` for 64-bit UP
- `installDir/vxworks-6.x/target/lib_lp64_smp` for 64-bit SMP

If you are doing a VxWorks source build (VSB), the `libDir` is the resulting VSB directory when the VSB is created.

These files have various configuration options defined within them. The naming convention for configuration macros is `_WRS_CONFIG_FOO`.

In addition, the VxWorks project facility (CDF information used by Workbench and the `vxprj` command-line facility) causes VxWorks component and parameter macros to be defined based on which system components are included in a VxWorks image project (VIP), as well as how the project parameters are set. These definitions are found in the project directory, in the files `prjComps.h` (components)
and prjParams.h (parameters). The naming convention for VxWorks components is INCLUDE_prefix for OS components or DRV_driverType_driverName for VxBus device drivers. There is no naming convention for parameters. (For more information on CDFs, see the VxWorks Custom Component and CDF Developer’s Guide.)

Configuration macros generally are tested for being defined (#ifdef _WRS_CONFIG_FOO), though there may be some exceptions. For more information, refer to libDir/h/config/vsbConfig.h. Component macros are always tested for being defined (#ifdef INCLUDE_FOO). Parameter macros take on various values, including integers and strings, depending on the meaning and usage of the macro. Refer to the parameter definitions in the following:

```
installDir/vxworks-6.x/target/config/comps/vxWorks/*cdf
```

Some of the macros used for conditional compilation and configuration in VxWorks SMP BSPs are listed in Table 6-1.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>_WRS_CONFIG_SMP</td>
<td>Configuration</td>
<td>Kernel libraries have been built with SMP support.</td>
</tr>
<tr>
<td>VX_SMP_NUM_CPUS</td>
<td>Parameter</td>
<td>Number of CPUs assigned to run the SMP image.</td>
</tr>
<tr>
<td>INCLUDE_SMP_DEMO</td>
<td>Component</td>
<td>Include SMP demo code in the image.</td>
</tr>
</tbody>
</table>

To reduce footprint size, and allow the BSP to continue to work in UP mode, code and data structures that are specific to SMP should be surrounded by the following construction:

```
#ifdef _WRS_CONFIG_SMP
/* code or data */
#endif /* _WRS_CONFIG_SMP */
```

### 6.6.2 Updating and Adding CDF Code

BSP CDF code to support SMP should be located in a CDF file in the BSP directory. This file is generally named 20bsp.cdf. Additional CDF files may be defined therefore you may choose to include SMP-specific CDF code in a separate file.

BSP CDF files use the following naming convention: nnbsp.cdf, where nn is a two digit number. The number determines the order in which the CDF files in the BSP directory are read by the project tools.

**NOTE:** CDF files are sorted by file name (by number then alphabet) so that files with larger numbers (nn) generally override lower numbered files where conflicts exist. For example, if the global file 10bsp.cdf includes a generic definition for the component INCLUDE_FOO, a BSP-specific file named 20bsp.cdf can override the generic definition by including its own definition for INCLUDE_FOO.
For more information about working with CDF components and parameters, see the VxWorks Custom Component and CDF Developer's Guide.

**Mandatory changes**

**Bsp statement**

The **Bsp** statement must include an **MP_OPTIONS** clause, which specifies that the BSP supports SMP. The following is an example of a **Bsp** statement:

```plaintext
Bsp mySMPBsp {
  NAME My SMP BSP
  SYNOPSIS SMP support for the XYZ board
  CPU MIPSI64
  TOOL sfdiab
  ENDIAN big
  FP soft
  MP_OPTIONS SMP
  REQUIRES INCLUDE_IPI_SUPPORT \ DATA_MODEL ILP32 \ INCLUDE_VXBUS \ INCLUDE_KERNEL \ DRV_TIMER_XYZTIMER \ DRV_INTCTRL_CPU \ DRV_INTCTRL_GLOBAL \ DRV_SIO_XYZSIO
}
```

To indicate 64-bit support, include the following in the **Bsp** statement:

```plaintext
DATA_MODEL LP64
```

If the **DATA_MODEL** statement is not included, the BSP defaults to 32-bit support or you can indicate 32-bit support as shown in the example above.

The BSP defaults to UP and the BSP only builds a UP image, if the **MP_OPTIONS** statement is missing.

For more information about the **Bsp** statement, see the VxWorks Custom Component and CDF Developer’s Guide.

**VX_SMP_NUM_CPUS Parameter**

Set the **VX_SMP_NUM_CPUS** parameter to the default number of CPUs you want to be used for the VxWorks SMP runtime. For example:

```plaintext
Parameter VX_SMP_NUM_CPUS {
  NAME Number of CPUs enabled for SMP
  DEFAULT 8
}
```

Note that the BSP should not assume that the number of CPUs is always the default set by this parameter. You can change this parameter at project configuration time. This is particularly important when allocating device instances in **hwconf.c** (see 6.6.9 Configuring VxBus (hwconf.c), p. 141).

**Optional Changes**

**ENABLE_ALL_CPUS Parameter**

The **ENABLE_ALL_CPUS** parameter attempts to enable the number of CPUs specified with **VX_SMP_NUM_CPUS**. The default is **TRUE**, in which case VxWorks
boots with all CPUs enabled and running. You can set this parameter to FALSE for debugging purposes. In this case, only logical CPU 0 is enabled by the VxWorks initialization code. You can then use the `kernelCpuEnable()` routine to enable a specific CPU once the system has booted.

**VX_ENABLE_CPU_TIMEOUT Parameter**

The VX_ENABLE_CPU_TIMEOUT parameter sets the time-out value (in seconds) for the period during which additional CPUs may be enabled. When `kernelCpuEnable()` is called, the routine waits for the time defined by VX_ENABLE_CPU_TIMEOUT for any additional CPUs to come up. If ENABLE_ALL_CPUS is set to TRUE, the value of VX_ENABLE_CPU_TIMEOUT is used as the time-out period for enabling all CPUs. If any of the CPUs fail to come up correctly, an error is displayed in the console.

**INCLUDE_PROTECT_IDLE_TASK_STACK Component**

In SMP, the idle task is the spin loop that CPUs execute when waiting to be tasked. The INCLUDE_PROTECT_IDLE_TASK_STACK component provides a set of parameters for configuring the exception stack for the CPU idle task. In addition to being used for exception processing, the exception stack is also utilized for task switch hooks. This should be considered when configuring the stack size. If you include this component, you may also want to set the following parameters in your BSP:

**IDLE_TASK_EXCEPTION_STACK_SIZE Parameter**

The size (in bytes) of the idle task’s exception stack.

**IDLE_TASK_EXC_STACK_OVERFLOW_SIZE Parameter**

The size (in bytes) of the overflow protection area adjacent to the idle task’s exception stack.

**IDLE_TASK_EXC_STACK_UNDERFLOW_SIZE Parameter**

The size (in bytes) of the underflow protection area adjacent to the idle task’s exception stack.

### 6.6.3 Implementing a Boot Loader Strategy

Booting multicore processors immediately raises all the primitive SMP issues such as getting secondary CPUs to a known state and establishing control and communication between the CPUs. Your implementation choices are as follows:

- Use the VxWorks boot loader.
  
  Typically, the standard VxWorks boot loader is configured only to start a single CPU of a multicore part and to leave the other CPUs as nearly off as is possible (this is architecture dependent). Using a VxWorks boot loader generally requires modifications (romInit.s, sysALib.s) to add initialization and startup of secondary CPUs.

  Many Wind River-supplied BSPs use the VxWorks boot loader.

- Use a vendor boot loader.
  
  A vendor-supplied boot loader can be a better choice than using a VxWorks boot loader for SMP, depending on a variety of factors. Vendor boot loaders may have richer solutions for CPU-variant and secondary CPU initialization, board configuration, and so forth. Using a vendor boot loader to boot VxWorks
SMP requires an additional learning curve in order to use the vendor boot loader, and additional design and implementation effort to integrate booting VxWorks images. However, the total effort, reliability and long-term maintenance may be better than a custom built VxWorks boot loader, especially if you are mixing different operating systems on a multicore part.

- Use a mixed boot loader strategy.

One possible development strategy is to use the vendor boot loader to launch a minimal VxWorks boot application, which consists of a VxWorks bootloader shell and interface built as a small VxWorks image. Then, use the VxWorks boot application to launch a full VxWorks image. This provides a short path to getting a VxWorks image running on the target, and then provides the standard VxWorks boot prompt mechanism for launching the full VxWorks image. Sometime later in the development process, when the system is up and development is more stable, a deployable version of the boot loader can be developed.

The following BSPs offer examples of the above implementations:

- rmi_xlr_mipsi64sf—This BSP offers a choice of using the vendor boot loader or the VxWorks boot loader.
- sb1480—This BSP uses a Broadcom Common Firmware Environment (CFE) routine to initialize the system and launch the VxWorks boot loader.

For more information on boot applications, see 4.3 PROFILE_BOOTAPP, p.101 and the VxWorks Kernel Programmer’s Guide: Boot Loader.

6.6.4 Implementing SMP Boot Logic

One of the greatest challenges you must overcome when developing an SMP BSP is getting a multicore board out of reset with all of the CPUs and CPU-interconnects initialized correctly for VxWorks. If the processor and board are not sufficiently initialized, it may not be possible to start the secondary CPUs in an orderly fashion.

On multicore boards, one CPU is considered to be the bootstrap CPU. All other CPUs are referred to as secondary CPUs. Typically, the bootstrap CPU is the only CPU enabled out of cold reset and secondary CPUs do not execute any instructions until explicit steps are taken by the bootstrap CPU.

With those terms in mind, the VxWorks bootstrap process out of hardware reset must achieve the following state:

- The bootstrap CPU must be running VxWorks (only on the bootstrap CPU).
- The secondary CPUs must be quiescent and must be able to remain in that state indefinitely without any adverse affects on the system.
- The bootstrap CPU initializes the hardware as needed to achieve a quiescent state. If secondary images are booted (typically by the image running on the bootstrap CPU), they share system resources with the remainder of the OS.

The remainder of this section expands on the strategies and details of how to achieve this state using either a VxWorks boot loader or existing vendor boot support. It also discusses SMP-specific boot details for both bootstrap and secondary CPUs.
Bootimg the Bootstrap CPU

In VxWorks, the bootstrap CPU has ownership of (and responsibility for) several critical chip-wide and board-wide resources. For example, the bootstrap CPU must configure the system interrupt controller and all of the resources the controller uses directly. On multicore systems, the bootstrap CPU typically must also configure interconnect buses, mailboxes, global interrupt controllers, and other global resources. Therefore, the bootstrap CPU always has some unique functionality that is not shared by the secondary CPUs. The bootstrap CPU runs the boot ROM or boot application that contains the boot loader code. The bootstrap CPU then jumps to the image that is loaded by the boot loader (it is the only CPU running when the image is entered) and, if it is an SMP image, the bootstrap CPU enables the other CPUs assigned to it as part of the initialization process.

The bootstrap CPU must hold the secondary CPUs in a quiescent state until they are started by the `sysCpuEnable()` routine. Note that quiescent means the secondary CPU is most likely spinning somewhere early in its own bootstrap process.

A sample boot sequence for the bootstrap CPU might be as follows:

1. (boot loader, UP) Perform essential boot initialization functions that are specific to the CPU, such as status register initialization, local MMU and cache initialization, and BAR initialization.
2. (boot loader, UP) Perform system-global resource initialization such as memory controller initialization and system interrupt controller initialization.
3. (boot loader, UP) Load an SMP image and jump to its `sysInit()` routine.
4. (runtime, SMP) Re-initialize essential functions that are specific to the CPU.
5. (runtime, SMP) Re-initialize system-global resources such as interrupt controllers.
6. (runtime, SMP) Initialize inter-CPU synchronization. This could take the form of setting up a message bus to be used to communicate with the secondary CPUs (such as the Fast Messaging Network (FMN) in RMI CPUs), or setting up a known location in shared RAM with a magic number (as is done in the hpcNet8641 BSP).
7. (runtime, SMP) Bring the secondary CPUs out of reset, but held back using the inter-CPU synchronization method set in the previous step.
8. (runtime, SMP) Enter the kernel.
9. (runtime, SMP) When `sysCpuEnable()` is called by the core OS for a given CPU, signal the given secondary CPU to execute past the synchronization point in the boot loader and cause it to boot into the entry point passed to `sysCpuEnable()`.

Changes to Booting SMP Images in VxWorks 6.8 Update Pack 1 and Beyond

Beginning in VxWorks 6.8 Update Pack 1, the first CPU in a set of CPUs allocated to the SMP image may not be the bootstrap CPU. This is because beginning with Update Pack 1, VxWorks supports SMP images that run on a range of physical CPUs that does not begin with physical CPU 0. For more information on creating a VxWorks SMP configuration for use in an AMP system, see the VxWorks Kernel Programmer’s Guide: VxWorks SMP.
6 Extending a BSP to an SMP System

6.6 Implementation Details

Booting the Secondary CPUs

Once started, secondary CPUs perform essential boot initialization functions that are specific to the CPU, such as status register initialization, local MMU and cache initialization, and BAR initialization. After that, secondary CPUs take a distinctly different path from the bootstrap CPU.

VxWorks SMP requires secondary CPUs to wait—that is, not complete booting of VxWorks—until explicitly started by a call to `sysCpuEnable()` This can be delicate because in most cases, the secondary CPU is running some code from the boot loader and must be held back from proceeding.

A sample boot sequence for a secondary CPU might be:

1. (boot loader, UP) Perform essential boot initialization functions that are specific to the CPU, such as status register initialization, local MMU and cache initialization, and BAR initialization.
2. (boot loader, UP) Wait at the synchronization point for a signal from the bootstrap CPU that it is safe to proceed.
3. (boot loader, UP) When signaled by a call to `sysCpuEnable()` executing on another CPU, jump to the alternate entry runtime kernel point for secondary CPUs (usually `sysCpuInit()`).
4. (runtime, SMP) Reinitialize essential functions that are specific to the CPU.
5. (runtime, SMP) Enter kernel at the entry point provided by `sysCpuEnable()`.

6.6.5 Holding a Secondary CPU in Startup or Reset

The strategy used for holding a secondary CPU in startup is largely dependent on your target hardware—not all options are available for all hardware platforms—but generally falls into one of three models:

- **Method 1:** The first option is to spin in the VxWorks boot loader, possibly in `romInit()`, waiting for updates to well-known memory locations, shared semaphores, or something similar that informs the secondary CPU on how it should proceed.
- **Method 2:** The second option is to spin in a vendor-supplied boot loader and then use the vendor-supplied mechanism to jump into the VxWorks image.
- **Method 3:** The third option is to leave the secondary CPU suspended (literally doing nothing) until it is enabled with information about where to jump into VxWorks.

Each of the methods is expanded below. A given SMP BSP may use one or more of the above strategies (for example, if you need one implementation for cold boot and another for warm reboot). Regardless of what mechanism you choose, you must provide the same functionality in the BSP. That is, you must always hold the

---

**CAUTION:** If an SMP kernel is started on a non-bootstrap CPU, the boot code that runs on the first CPU (logical CPU 0, physical CPU X) must only initialize the hardware assigned to the set of CPUs in the SMP image. Touching hardware that is not specifically assigned to the CPU can corrupt the entire system.
secondary CPU at the earliest possible state until the core OS enables the secondary CPU.

- **Method 1: Spin in the VxWorks Boot Loader**
  
  In this scenario, a secondary CPU branches to code that spins reading some well-known memory locations for a known-good set of values. This happens as early as possible in `romInit()`. In a typical situation, 3 to 4 words in shared memory are used to encode how and where the secondary CPU should start (that is, the entry point and state). This shared memory is also used to enforce synchronization. For example, to avoid a secondary CPU starting with incomplete or corrupt data, the words are written by the bootstrap CPU in reverse order from the order that they are read by the secondary CPU. (For an example of this, see the `hpcNet8641d BSP waitForInitStartSet in romInit()`.)

  The secondary CPU should also modify the encoded locations in shared memory as soon as possible to a different well-known state, so that failure to launch does not simply loop back to `romInit()` and retry with bad values. Optionally (there is no Wind River API for this purpose), those memory locations can be used to share state information with the bootstrap CPU as the secondary CPU boots.

- **Method 2: Spin in the Vendor Boot Loader**
  
  This option is logically similar to option 1, but depends on a third-party API for reset and start details. Generally, the vendor API requires information about the entry point and any flags passed to the image, the CPU number, and possibly other options. This mechanism is used by the RMI XLR/XLS family with the RMI boot loader support enabled (that is, when `INCLUDE_RMI_BOOTLOADER_SUPPORT` is included in your VxWorks image).

  **NOTE:** Method 2 is similar to method 1 with the main difference being that in method 1, the processor spins in VxWorks code and in method 2, the processor spins in vendor-supplied code. Vendor boot loaders are naturally more focused on specific hardware and tend to offer more sophisticated auto-detection and configuration. They may also offer better support and maintenance on specific hardware. However, vendor solutions can be more complex and may not offer the same level of backward compatibility between hardware releases when compared to VxWorks boot loaders.

- **Method 3: Leave the Secondary CPU Suspended or Not Running**
  
  There are CPU-specific halt and start functions available in some hardware and firmware. For example, MIPS Cavium CNxxx families can be halted by writing a bit to a system register. In this case, the restart of an individual CPU usually involves significant work to set up the CPU correctly but the vendor’s SDK provides a public API that hides the details.

  This method is also used by the PowerPC `hpcNet8641d` for cold starts. And for ARM, secondary ARM CPUs are suspended in the VxWorks or vendor boot loader, waiting for a signal, interrupt, or IPI. The IPI handler provides information about where to jump into VxWorks.

  Note that although this option is available, it is usually not enough to simply shut a CPU down, especially if it is participating in a running system with potentially unwritten data, outstanding inter-CPU transactions, and similar unfinished work. Depending on the vendor API, the BSP may still be
6.6 Implementation Details

The sysToMonitor() routine must also handle halting and rebooting the secondary CPU using one of the methods described above. The method used for this warm reboot may or may not be the same as that used for cold boot.

At some point, the secondary CPU is awakened by a call to sysCpuEnable(). The time is highly variable. During development, the secondary CPUs are often started from the target console shell minutes or hours after the bootstrap CPU boots.

6.6.6 Prioritizing Interprocessor Interrupts (IPIs)

Any SMP-capable multicore processor must have provisions to let one CPU generate an interrupt on another CPU. This is called an inter-processor interrupt or IPI. IPIs constitute the basis of the cross-processor call (CPC) mechanism that allows one CPU to request that another CPU execute some function. This is the core of SMP scheduling, inter-CPU messaging, and debugging.

There are four IPI interrupt levels that are preassigned in an SMP system:

- 0:CPC
- 1:Debug
- 2:Scheduler
- 3:MIPC

**NOTE:** The above are preassigned values for SMP. SMP systems can have up to 32 IPI channels. The number of IPIs implemented is hardware specific.

From the hardware perspective, IPIs are generally implemented in a global interrupt controller device. That device is supported by a VxBus interrupt controller driver. For more information on VxBus and VxBus device drivers, see the *VxBus Device Driver Developer’s Guide*.

The integrity of the SMP operating system requires that the first IPI interrupt (IPI 0) have the highest priority in the system, and that this priority not be shared with any other interrupt. This is because cross-processor calls (which rely on IPI 0) are allowed in ISRs. Setting IPI 0 to the highest possible priority prevents a livelock that could occur if a nested interrupt service routine invokes a CPC.

Cross-processor call handlers themselves cannot issue CPCs.

As an example, assume that you are in the middle of sending an IPI 0 when a higher priority interrupt stops your IPI 0 operation. Now assume that the higher priority interrupt also wants to send an IPI 0. This is not possible because only one IPI 0 is allowed at any given time. The result of this scenario is a livelock. The higher priority interrupt waits forever because the IPI 0 that was interrupted can never complete and thus another IPI 0 cannot be issued.

Your SMP BSP must set the priority of IPI 0 interrupts above all other interrupt sources. The steps to accomplish this are hardware specific. Generally, the VxBus global interrupt controller driver is configured to route the IPI 0 interrupt to a given interrupt input for each CPU. The BSP must configure all CPUs and CPU interrupt controller drivers so that this interrupt input is serviced before all others.
For more information about configuring interrupt controller drivers, see *Configuring Interrupt Controller Drivers*, p.141.

For MIPS processors (as an example), this means that the mask for the IPI 0 interrupt must be carefully selected in the `intPrioTable` in the BSP. It may also require an alternate table to be used in the `ffsMsb` algorithm, if the IPI interrupt cannot be connected to the interrupt with the highest positional priority when scanned from left to right or right to left. For an example of how this is achieved, see the `xlrFfsTbl` and `intPrioTbl` in the following:

```c
installDir/vxworks-6.x/target/config/rmi_xlr_mipsi64sf/sysLib.c
```

### 6.6.7 Modifying Cache Handling

In most cases, modifications to cache handling are covered by the architecture and core OS code. However, if your BSP implements cache routines, you should consider the following:

- The VxWorks UP cache routines are designed around a uniprocessor system. Enabling and disabling the caches, invalidating, flushing, or clearing elements of the cache all have a CPU-specific nature, as they refer to the local CPU’s cache. In an SMP system, this CPU-specific nature is less meaningful. The systems that are supported by VxWorks SMP all provide hardware cache coherency, both between the individual CPUs in the SMP system and between the memory subsystem and the device address space.

In the remainder of this section, modifications to cache routines are listed under the corresponding `cachLib` routine. The BSP itself does not implement these routines directly. Rather, these routines call out to architecture-specific implementations or BSP-specific implementations of the required functionality. For more information, see the reference entries for `cacheLib` and `aimCacheLib`.

#### cacheEnable( ) and cacheDisable( )

The only way for the hardware cache coherency to be effective is to have the caches turned on at all times. VxWorks SMP therefore turns on the caches of each CPU as it is enabled, and never allows them to be disabled. Calling `cacheEnable()` in VxWorks SMP always returns OK. Calling `cacheDisable()` in VxWorks SMP always returns ERROR, with `errno` set to `S_cacheLib_FUNCTION_UNSUPPORTED`.

#### cacheClear( ), cacheFlush( ), and cacheInvalidate( )

Assuming that the hardware supports cache coherency in both instruction and data caches, these routines are not necessary. In that case, if these routines are called in VxWorks SMP, they perform no function (are no-ops) and simply return OK.

`cacheInvalidate( )` can be a valid operation for instruction caches when the processor does not support hardware coherency for them, as with ARM.
cacheLock() and cacheUnlock()

These routines are not supported in VxWorks SMP. If they are called, they return ERROR and set errno to S_cacheLib_FUNCTION_UNSUPPORTED.

Modifying Cache Modes to Support Coherency

As noted in 6.4.2 Hardware Support, p.121, SMP requires cache coherency within the CPUs and between CPUs. This is required to support inter-processor communication (IPIs, CPCs), atomic operations, and DMA. This may require specific cache modes. For example, one way of accomplishing cache coherency is through bus snooping. Note that this mode may not be enabled by default in the BSP (in which case, you must perform the necessary steps to enable the mode). In addition, some caches support both copyback and writethrough modes.

Coherency and atomic operations may not be supported in both modes. For more information about supported cache modes, see the VxWorks Architecture Supplement and your hardware documentation.

6.6.8 Implementing SMP BSP Routines

In addition to the routines that are required for any UP BSP, there are several routines that must be updated or created specifically for SMP BSPs. For the currently supported BSPs, implementations are provided in the sysLib.c and sysALib.s files in individual BSP directories. For example:

installDir/vxworks-6.x/target/config/bspName/sysLib.c
installDir/vxworks-6.x/target/config/bspName/sysALib.s

CPU Index and CPU ID

There must be a way for code to identify which physical CPU it is running on, as well as a zero-based, sequential index for core OS functions.

The physical CPU identification, or CPU ID, is generally used by drivers, interprocessor communication mechanisms such as MIPC, and other code that needs to identify the actual physical CPU that it is running on. The CPU ID is highly hardware dependent, and may not be sequential from CPU to CPU. For instance, it could be a bit mask, or other unique identifier. For this reason, the CPU ID may not be suitable for use as an index into data structures without first putting it through a transform function.

The CPU index is a zero-based, sequential index representing a CPU. Because the CPU index is sequential in nature, it is suitable for indexing data structures. Core OS functions generally deal with the CPU index. The CPU index is also sometimes referred to as the CPU number (or cpuNum) in core OS code.

In most architectures supported in VxWorks 6.8 and earlier releases, the CPU index is derived by performing a transform on the CPU ID to make it zero-based and sequential. This implies that SMP images must run on physical CPU 0 (the bootstrap CPU) through CPU N. This restriction is lifted in VxWorks 6.8 Update Pack 1. Therefore, calling code should never assume that the two are the same.
In general, support for determining the CPU ID and CPU index is provided by the architecture code with the \texttt{vxCpuIdGet()} and \texttt{vxCpuIndexGet()} routines. In some cases, these routines may depend on the BSP for their implementation.

**Logical and Physical CPU Index in VxWorks 6.8 Update Pack 1 and Beyond**

In order to support running SMP kernels on CPU sets that do not include the bootstrap CPU in VxWorks 6.8 Update Pack 1 and beyond, a clear distinction is made between the logical CPU index and the physical CPU index.

**NOTE:** Running SMP kernels on CPU sets that do not include the bootstrap CPU is currently supported for MIPS (as of VxWorks 6.8 Update Pack 1) and PowerPC (as of VxWorks Update Pack 2) processors only. However, the changes described in this section apply to all architectures.

The physical CPU index is directly tied to the hardware. The CPU with physical index zero is always considered the bootstrap CPU—that is, the CPU used to bring the processor out of a hard reset and boot the system. Subsequent CPUs are numbered from 1-$N$, and are considered to be secondary CPUs.

By contrast, the logical CPU index is local to the VxWorks instance. It is also zero-based and sequential. But logical CPU index zero does not necessarily correspond to the bootstrap CPU. It may map to any CPU in the processor. Logical CPU 0 refers to the first CPU in the set of CPUs assigned to an SMP image.

When reading the routine descriptions that follow, keep in mind that in VxWorks 6.8 and earlier releases, the logical CPU index and the physical CPU index—while different in concept—are always the same in practice. In VxWorks 6.8 Update Pack 1 and later, they are distinctly different in practice. When implementing routines in VxWorks 6.8 Update Pack 1 and beyond, you must be sure which type of index you are dealing with in order to implement the routine correctly.

\textbf{vxCpuPhysIndexGet()}

\begin{verbatim}
unsigned int vxCpuPhysIndexGet(void)
\end{verbatim}

The \texttt{vxCpuPhysIndexGet()} routine returns the physical index value for the CPU it is called from. The prototype for this function is declared in \texttt{installDir/vxworks-6.x/target/h/vxCpuLib.h}. An implementation of \texttt{vxCpuPhysIndexGet()} is provided by the architecture code for each supported architecture.

\textbf{vxCpuIndexGet()}

\begin{verbatim}
int vxCpuIndexGet(void)
\end{verbatim}

\textbf{NOTE:} If you are porting a BSP from a release prior to VxWorks 6.8 Update Pack 1, your BSP may have code that assumes that the logical CPU index and the physical CPU index are the same for an SMP system. If this is the case, your BSP will not work in the current release if the SMP image starts on a physical CPU index greater than 0. Backward compatibility is ensured only if the system is still configured such that the SMP image starts on physical CPU index 0.
The \texttt{vxCpuIndexGet()} routine returns the logical index of the CPU that it is called from. If there is a standard method that an architecture can use to determine the logical CPU index, the architecture code provides an implementation of this routine.

\begin{verbatim}
unsigned int vxCpuIndexGet(void)
\end{verbatim}

The \texttt{vxCpuIndexGet()} routine returns the architecture-based ID of the CPU that it is called from.

\begin{verbatim}
unsigned int vxCpuIdGet(void)
\end{verbatim}

\begin{verbatim}
void sysHwInit(void);
\end{verbatim}

If your hardware requires specific initialization to support SMP, then add the necessary calls to hardware initialization routines to \texttt{sysHwInit()}. Examples include setting cache coherency modes, placing secondary CPUs in a quiescent state and ready to be enabled, and so forth.

\begin{verbatim}
STATUS vxCpuStateInit
{
    unsigned int cpuIndex, /* Logical index of CPU to enable */
    WIND_CPU_STATE *cpuState, /* Arch-specific CPU state structure */
    char *pIdleStackBase, /* Idle task stack base */
    void (*kernelCpuEntry) (void) /* OS Entry function for CPU */
}
\end{verbatim}

This routine initializes the \texttt{WIND\_CPU\_STATE} structure for the CPU designated by the logical index \texttt{cpuIndex}. The \texttt{WIND\_CPU\_STATE} structure is later passed to \texttt{sysCpuEnable()} to determine the initial state of the secondary CPU. Normally, this routine should already be provided by the architecture code. However, in some cases (most notably the IA-32 architecture), this routine must be provided by the BSP. This allows the initial state of secondary CPUs to be fine-tuned at a board level.

This routine is somewhat similar to the architecture specific routine \texttt{taskRegsInit()} except that \texttt{taskRegsInit()} only needs to initialize a \texttt{REG\_SET} structure for the initial state of a VxWorks task. This routine is initializing the state for a CPU.

The calling routine in the core OS specifies the entry point (\texttt{kernelCpuEntry}) that should be called once the secondary CPU is initialized, and the base of the idle task's exception stack (\texttt{pIdleStackBase}).

Because the routine pointed to by \texttt{kernelCpuEntry} is a C function, it must be invoked with a proper C context. For example, on architectures that support small data areas, the small data base register(s) in the \texttt{WIND\_CPU\_STATE} structure (or in a \texttt{REG\_SET} structure within \texttt{WIND\_CPU\_STATE}) must be initialized to the base of the small data area(s).
sysCpuEnable()

```c
#ifdef _WRS_ARCH_SUPPORTS_VX_CPU_STATE_INIT
int sysCpuEnable
{
    unsigned int cpuIndex, /* Logical CPU Index of CPU to enable */
    WIND_CPU_STATE *cpuState /* Arch-specific CPU state structure */
};
#else
int sysCpuEnable(
    unsigned int cpuIndex, /* Index of CPU to enable */
    void (*kernelCpuEntry) (void), /* OS Entry function for CPU */
    char *pIdleStackBase /* Idle task stack base */
);
#endif
```

This routine is called from the core OS code to start a secondary CPU. The CPU is designated by logical CPU index, `cpuIndex`. Depending on whether the architecture code for your processor supports the `WIND_CPU_STATE` structure (as designated by `_WRS_ARCH_SUPPORTS_VX_CPU_STATE_INIT`, defined in `installDir/vxworks-6.x/target/h/arch/`), the parameters following `cpuIndex` are different.

If `_WRS_ARCH_SUPPORTS_VX_CPU_STATE_INIT` is defined, then the architecture or BSP initializes the passed `WIND_CPU_STATE` structure. The format of this structure is architecture-dependent. However, at a minimum, it contains an OS entry point and an initial stack pointer, as well as the initial value of other registers. This is typically encapsulated in a `REG_SET` structure.

This routine must “wake up” the designated secondary CPU into the context provided by `WIND_CPU_STATE`. The method for waking up the secondary CPU depends on the secondary CPU boot method selected. (See Booting the Secondary CPUs, p.131). Once the secondary CPU is awakened it begins executing in `sysCpuInit()`. In order for `sysCpuInit()` to have the required information (like where to enter the kernel and how to set up the stack), you generally need to pass some information to `sysCpuInit()` using a global data structure prior to enabling the CPU. For instance, this is the code for the MIPS RMI XLR BSP:

```c
sysCpuInitTable[cpuIndex] = cpuState->regs.pc
sysCpuArgsTable[cpuIndex].sp = cpuState->regs.spReg
sysCpuArgsTable[cpuIndex].gp = cpuState->regs.gpReg
sysCpuEnableFlag[cpuIndex] = 1;
return OK;
```

Here, the program counter, stack pointer, and global pointer registers are copied into the `sysCpuArgsTable`, and the CPU is released for execution by setting the `sysCpuEnableFlag`.

If `_WRS_ARCH_SUPPORTS_VX_CPU_STATE_INIT` is not defined, then rather than a `WIND_CPU_STATE` structure, you are simply given a starting function address (`kernelCpuEntry`) and a stack pointer (`pIdleStackBase`). In this case, a similar function is performed. Save `kernelCpuEntry` and `pIdleStackBase` to a known location and enable the given secondary CPU. The secondary CPU must then initialize the stack pointer register accordingly and branch to `kernelCpuEntry`.

This routine returns `OK` on success or `ERROR` on failure.

Changes to sysCpuEnable() in VxWorks 6.8 Update Pack 1 and Beyond

In VxWorks 6.8 Update Pack 1 and beyond, the `cpuIndex` parameter of the `sysCpuEnable()` routine is a physical CPU index. For more information, see Logical and Physical CPU Index in VxWorks 6.8 Update Pack 1 and Beyond, p.136.
sysCpuInit()  

The `sysCpuInit()` routine is the BSP-specific entry point for secondary CPUs. The interface to this routine is system-dependent. It depends on the contract between the boot loader code (`romInit`) and the runtime code. The ROM reset routine (`romInit`) calls `sysCpuInit()` when executing on a secondary CPU. The function of `sysCpuInit()` is analogous to `sysInit()`, the entry point for the bootstrap CPU.

The `sysCpuInit()` routine performs CPU-specific initialization such as setting the stack pointer, and initializing the MMU and cache. It then branches to the OS entry point that is passed to `sysCpuEnable()`.

sysCpuDisable()  

```c
STATUS sysCpuDisable
{
    unsigned int cpuIndex,       /* Logical index of CPU to disable */
}
```

This routine disables the given secondary CPU and clears the `vxCpuEnabled` mask bit for the CPU designated by `cpuIndex`. The disabled CPU is no longer available for scheduling. The steps involved in disabling a CPU are hardware dependent. For an example of how this routine is implemented on the IA-32 architecture, see `installDir/vxworks-6.x/target/config/pcPentium/sysLib.c`.

In VxWorks 6.8 Update Pack 1 and earlier releases, this routine is optional. It is not actually used by any entities outside the BSP. This routine may be used by the BSP itself in the implementation of `sysToMonitor()`.

Changes to `sysCpuDisable()` in VxWorks 6.8 Update Pack 1 and Beyond  

In VxWorks 6.8 Update Pack 1 and beyond, the `cpuIndex` parameter of the `sysCpuDisable()` routine is a physical CPU index. For more information, see *Logical and Physical CPU Index in VxWorks 6.8 Update Pack 1 and Beyond*, p.136.

sysCpuAvailableGet()  

```c
UINT32 sysCpuAvailableGet(void)
```

This routine returns the number of physical CPUs available on the processor. Note that this may be different than `VX_SMP_NUM_CPUS`, which is the number of CPUs allocated for the SMP kernel. It may also be different from the number of enabled CPUs.

sysToMonitor()  

```c
STATUS sysToMonitor
{
    int startType  /* parameter passed to ROM to tell it how to boot */
}
```

This routine performs a reboot of the VxWorks SMP image. A reboot may be initiated with a `^X` or `reboot` command entered at the system console, or programmatically. In either case, some general VxWorks requirements must be taken into account:
Reboot processing must be able to execute on any of the running CPUs. You may not be able to predict which CPU happens to be initiating the reboot.

All secondary CPUs configured in the SMP image must be brought to a quiescent state, ready to be started again. No critical transactions should be left unfinished.

If the bootstrap CPU is configured into the SMP image being rebooted, then it must be made to jump to the reset vector. On some hardware platforms, this can be accomplished with a global processor reset. However, if warm boot is to be supported, it is necessary to ensure that it is the bootstrap CPU that jumps to the warm start entry in `romInit()` (not a secondary CPU). In the pcPentium4 BSP, this is accomplished by first checking whether the CPU running the routine is the bootstrap CPU. If it is not, then `sysToMonitor()` is invoked on the bootstrap CPU using an IPI. Then, the secondary CPU goes into a loop waiting to be rebooted.

If the bootstrap CPU is not configured into the SMP image being rebooted (which is possible in VxWorks 6.8 Update Pack 1 and beyond), then this routine must not reset the entire processor. Rather, it must only reboot the secondary CPUs that are participating in the image. When `sysToMonitor()` finishes execution, all of the CPUs that are owned by the OS instance must be in a state where it is possible to reload them using `wrload`. In other words, if you do not own the bootstrap CPU, do not perform a full processor reset, just reboot your own OS instance.

Regardless of which CPU is running `sysToMonitor()`, the following may need to be accomplished on all CPUs that are configured into the SMP image:

- Under ideal circumstances, you should notify any applications of the pending shutdown. (There is no standard Wind River-supplied API to do this.)
- Disable interrupts.
- Set or verify any sentinels or other reboot states to keep the secondary CPUs in the hold state.
- Disable and flush caches, flush write buffers, and perform anything else needed to complete any outstanding transactions with other CPUs or devices.

**NOTE:** Frequently, there are board and CPU-specific timing issues to address in order to allow buffers time to flush, or devices time to become quiescent.

Put all CPUs configured in the SMP image back into appropriate board-dependent reset states, whatever corresponds to your boot strategy. (This is hardware specific.)

`sysToMonitor()` should not attempt to use any blocking operating system services to assist with the reboot process. Synchronous CPCs cannot be trusted to work reliably in this situation, because the reason for calling `sysToMonitor()` might in fact be because of some type of lock-up or other problem within the collection of SMP processors. A BSP can attempt to use direct IPI operations or asynchronous CPC operations, but the BSP should ensure that a failure of any (or all) CPUs to respond to the CPC or IPI operations does not prevent the system from rebooting.

Assuming the bootstrap CPU is included in the SMP image being rebooted, the code that is executed after `sysToMonitor()` may be configured for UP operation (as is the case for VxWorks boot loaders). Because of this, `sysToMonitor()` should
ensure that the hardware is appropriately reset so that the bootstrap processor
does not have to contend with activities of any non-bootstrap processor.

6.6.9 Configuring VxBus (hwconf.c)

The hardware devices available to a VxWorks image, such as serial ports, Ethernet
ports, timers, interrupt controllers, and so on, are listed in the `hcfDeviceList[]`
array in the BSP `hwconf.c` file. During system initialization, VxBus iterates through
the devices listed in `hcfDeviceList[]` and attempts to pair each device with a
corresponding device driver (for more information on VxBus and system
initialization, see the *VxBus Device Driver Developer’s Guide: Device Driver
Fundamentals*). If a driver is found that supports a listed device, the device and
driver are paired together to form a device instance that can be used within the
system.

Configuring Interrupt Controller Drivers

Multiprocessing hardware generally supports a hierarchy of interrupt controllers.
At the CPU level, there is generally support for determining what interrupts need
servicing, masking individual interrupt sources, and associating interrupt service
routines with different interrupt sources (usually through a vector table). In
addition, there is generally one or more interrupt controllers that perform similar
functions, but at a global, inter-CPU level. A typical code path for servicing an
interrupt on a given CPU might be:

1. Code at the vector associated with the interrupt source is executed.
2. The CPU is queried to determine which source needs servicing. The CPU
interrupt controller driver ISR is called to handle the interrupt.
3. The CPU interrupt controller driver ISR looks up the handler that is associated
with the given interrupt.
   For some CPU-specific devices, like internal timers, this will likely be the
   handler that ultimately services the device. For processor-global devices, such
   as network controllers, there is often an additional interrupt controller that is
global to all CPUs that handles the device. For the purposes of this
documentation, this additional interrupt controller is called the *global interrupt
controller*. For processor global devices, the global interrupt controller driver
ISR is called.
4. The global interrupt controller driver ISR looks up the handler that it has
registered to service the interrupt, and calls that routine. This is likely the
routine that ultimately services the device.

The interconnections between CPU and global interrupt controller drivers and
associated handlers is accomplished through tables defined in `hwconf.c`, and is
described in the following sections.

Interrupt Signal Routing and Processing Illustration

*Figure 6-1* can help you visualize the effect of the interrupt controller configuration
tables described in the following sections.
In Figure 6-1, you see that global Device A interrupts on input pin 1 of the global interrupt controller, which routes the interrupt signal to pin 5 of CPU 1. Global Device B interrupts on input pin 2 of the global interrupt controller, which routes the interrupt signal to pin 3 of CPU 0. Each CPU has a local timer device, which interrupts on pin 7 of the CPU. When an interrupt fires on a given CPU, the CPU interrupt entry code is executed. It calls the CPU interrupt controller driver ISR instance for the CPU. If it is one of the local CPU timers that is interrupting, the timer driver ISR instance for the given CPU is called to handle it. Otherwise, the global interrupt controller driver ISR called, which in turn finally calls the driver ISR for Device A or B.

For more information from the VxBus and driver perspective, see the VxBus Device Driver Developer’s Guide: Interrupt Controller Drivers.

**Interrupt Controller CPU Routing**

In SMP, each interrupt must be routed to a single CPU. This routing may be changed at runtime with the `vxbIntReroute()` routine. But the initial routing is provided by the CPU routing table. In this example, device interrupts are hardwired to global interrupt controller input pins by hardware.

```
#define DEVICE_A_INT_PIN 1 /* Device A interrupts on global int ctrlr pin 1 */
#define DEVICE_B_INT_PIN 2 /* Device B interrupts on global int ctrlr pin 2 */

const struct intrCtlrCpu globalIntCtlrCPURoute[] =
{
  {DEVICE_A_INT_PIN, 1}, /* Route Device A interrupts to CPU 1 */
  {DEVICE_B_INT_PIN, 0}, /* Route Device B interrupts to CPU 0 */
};
```
Here you see that the global interrupt controller driver is being configured to set up the global interrupt controller device to route Device A interrupts to logical CPU index 1 and Device B interrupts to logical CPU index 0.

### Interrupt Controller Cross Bar

The interrupt controller cross bar configures signal routing from the global interrupt controller's input pins to the CPU interrupt controller input pins.

```c
const struct intrCtlrXBar globalIntCtlrXBar[] = {
    {DEVICE_A_INT_PIN, 5}, /* Assign Device A to CPU input pin 5 /*
    {DEVICE_B_INT_PIN, 3}, /* Assign Device B to CPU input pin 3 /*
};
```

In this example, the global interrupt controller driver is being configured to set up the interrupt controller device to route Device A interrupt signals to pin 5 of whatever CPU they are assigned to by the CPU routing table. Device B interrupt signals are to be sent to CPU pin 3.

### CPU Interrupt Controller Inputs

Every CPU interrupt controller driver must be configured to associate each of the input pins it handles to a driver that handles the interrupting device. This is done with the an interrupt controller inputs table. For example:

```c
const struct intrCtlrInputs cpu0IntCtlrInputs[] = {
    /* inputPin drvName drvUnit drvIndex */
    {2, "globalIntCtlr", 0, 2},
    {3, "globalIntCtlr", 0, 3},
    {4, "globalIntCtlr", 0, 4},
    {5, "globalIntCtlr", 0, 5},
    {6, "globalIntCtlr", 0, 6},
    {7, "timerDriver", 0, 0},
    [etc]
};
```

This is an interrupt controller input table for a CPU-level interrupt controller driver. Here you see that CPU input pins 2-6 are routed to a downstream global interrupt controller driver: "globalIntCtlr". CPU input pin 7 is routed directly to a device driver ("timerDriver") that services the device.

**drvUnit** refers to the device unit number, and is used to distinguish multiple instances of the same driver. In this case, you want to route interrupts on pin 7 of CPU 0 to the unit 0 instance of the "timerDriver" driver.

**drvIndex** is used to uniquely identify the interrupt to the downstream handler of the interrupt (in this case, "globalIntCtlr"). The "globalIntCtlr" driver ISR must field interrupt calls from multiple CPUs, each of which have the same set of input pin numbers. So the **drvIndex** is required to uniquely identify the interrupt source. Assuming that each CPU has 32 input pins, the CPU interrupt controller input table for CPU 1 might look like this:

```c
const struct intrCtlrInputs cpu1IntCtlrInputs[] = {
    /* inputPin drvName drvUnit drvIndex */
    {2, "globalIntCtlr", 0, 34},
    {3, "globalIntCtlr", 0, 35},
    {4, "globalIntCtlr", 0, 36},
    {5, "globalIntCtlr", 0, 37},
    {6, "globalIntCtlr", 0, 38},
    {7, "timerDriver", 1, 0},
    [etc]
};
```
Using `drvIndex`, the global "globalIntCtrl" (and VxBus) can distinguish an interrupt that comes in on pin 2 on CPU 0 (`drvIndex 2`) from an interrupt that comes in on pin 2 on CPU 1 (`drvIndex 34`).

Note that in the case of the "timerDriver", `drvIndex` is set to 0 for both CPU 0 and CPU 1. In this case, a separate instance of the driver is created for each CPU because there is, in fact, a separate timer device for each CPU. Therefore, no index is required to distinguish the one interrupt source that will be serviced by each instance of the "timerDriver" driver.

Again, `drvUnit` refers to the device unit number, and is used to distinguish multiple instances of the same driver. You want to route interrupts on pin 7 of CPU 1 to the unit 1 instance of the "timerDriver" driver.

### Global Interrupt Controller Inputs

Global interrupt controller inputs must be specified to the global interrupt controller driver. The principal is the same as for CPU interrupt controller inputs, except there is normally a one to many relationship between global and CPU interrupt controllers. A global interrupt controller input table might look like this:

```c
const struct intrCtlrInputs globalIntCtlrInputs[] = {
    /* inputPin  drvName    drvUnit  drvIndex */
    {DEVICE_A_INT_PIN,  "deviceADriver",  0,  0},
    {DEVICE_B_INT_PIN,  "deviceBDriver",  0,  0},
    {etc}
};
```

In this case, "inputPin" refers to the global interrupt controller input pin, not the CPU input pin. Rather than passing the interrupt handling to another interrupt controller (as was done with the CPU interrupt controller inputs), the interrupt is passed to a device driver that ultimately services the interrupting device.

### Interrupt Controller Resources

The interrupt controller resources table pulls together all of the configuration settings for a given interrupt controller. Each CPU interrupt controller and each global interrupt controller defines its own table. For instance, the global interrupt controller driver resource table might look like this:

```c
const struct hcfResource globalIntCtlrResources[] = {
    "regBase",  HCF_RES_INT,  (void *)PIC_REG_BASE},
    "input",    HCF_RES_ADDR,  (void *)&globalIntCtlrInputs[0],
    "inputTableSize",  HCF_RES_INT,  (void *)NELEMENTS(globalIntCtlrInputs)},
    "cpuRoute",  HCF_RES_ADDR,  (void *)&globalIntCtlrCpuRoute[0]},
    "crossBar",  HCF_RES_ADDR,  (void *)&globalIntCtlrXBar[0]},
    "xlrCpuNum",  HCF_RES_INT,  (void *)VX_SMP_NUM_CPUS)},
};
```

#define globalIntCtlrResourcesSize  NELEMENTS(globalIntCtlrResources)

And the CPU interrupt controller resource table for a given CPU might look like this:

```c
const struct hcfResource cpu0IntCtlrResources[] = {
    "regBase",  HCF_RES_INT,  (void *)TRUE},
    "input",    HCF_RES_ADDR,  (void *)&cpuIntCtlr0Inputs[0]},
    "inputTableSize",  HCF_RES_INT,  (void *)NELEMENTS(cpu0IntCtlrInputs)},
};
```

#define cpu0IntCtlrResourcesSize  NELEMENTS(cpu0IntCtlrResources)
The above CPU interrupt controller resource table would need to be repeated for each CPU in the processor.

I/O Device Configuration

Configuring VxBus I/O device drivers in an SMP BSP is generally no different than configuring those same drivers in a UP BSP. Each device driver requires a resource table (const struct hcfResource). There may be some additional parameters required in the driver’s resource table that are SMP-specific.

Keep in mind that in a VxWorks SMP configuration, hardware devices that are external to the CPU can be shared by all CPUs running the SMP image. Therefore, SMP-safe device drivers need to include provisions for handling parallel requests from different CPUs. Shared resources must be protected with appropriate multi-processor mutual exclusion such as atomic operators, memory barriers, and spinlocks. Simply disabling interrupts does not suffice. Disabling interrupts is CPU-specific. In addition, the interrupt contexts within the driver may not assume that no tasks are running. In fact, tasks could be running on other CPUs. For this reason, it is important to verify that every device driver you are using is SMP safe. When evaluating a driver for use in an SMP system, read the driver documentation or source code carefully to be sure that the driver is SMP safe.

CPU-specific devices such as the timers mentioned in the above example are not shared by more than one CPU. Therefore, the SMP multiprocessing constraints do not apply to such drivers.

For more information on multi-processor mutual exclusion, see the VxWorks Kernel Programmer’s Guide. For more information on developing drivers in SMP, see the VxBus Device Driver Developer’s Guide.

Hardware Configuration Device List

Finally, each of the device resource tables (interrupt controllers and other devices) are included in the hardware configuration device list (hcfDeviceList[]), which is scanned by VxBus during system initialization.

    const struct hcfDevice hcfDeviceList[] = {
        { "globalIntCtlr", 0, VXB_BUSID_PLB, 0, globalIntCtlrResourcesSize, globalIntCtlrResources },
        { "cpuIntCtlr", 0, VXB_BUSID_PLB, 0, cpu0IntCtlrResourcesSize, cpu0IntCtlrResources },
        { "timerDriver", 0, VXB_BUSID_PLB, 0, timerDriverResourcesSize, timerDriverResources },
    #ifdef _WRS_CONFIG_SMP
    #if (VX_SMP_NUM_CPUS > 1)
        { "cpuIntCtlr", 1, VXB_BUSID_PLB, 0, cpu1IntCtlrResourcesSize, cpu1IntCtlrResources },
        { "timerDriver", 1, VXB_BUSID_PLB, 0, timerDriverResourcesSize, timerDriverResources },
    #if (VX_SMP_NUM_CPUS > 2)
        { "cpuIntCtlr", 2, VXB_BUSID_PLB, 0, cpu2IntCtlrResourcesSize, cpu2IntCtlrResources },
        { "timerDriver", 2, VXB_BUSID_PLB, 0, timerDriverResourcesSize, timerDriverResources },
    
...
const int hcfDeviceNum = NELEMENTS(hcfDeviceList);

Configuring VxBus Memory Allocation

VxBus drivers and core utilities need to allocate dynamic memory before the kernel dynamic memory services (malloc) are available. To accommodate this need, VxBus defines a special memory pool for its own use, as well as for drivers that require it. The size of this memory pool is determined by the HWMEM_POOL_SIZE preprocessor define.

The default for HWMEM_POOL_SIZE is 50,000 bytes. This is generally plenty for UP kernels, but when allocating additional device instances for SMP, this may not be enough. If the hardware memory pool is exhausted, the system is rebooted. If you find that your SMP kernel is immediately rebooting early in the initialization process, try increasing the size of the hardware memory pool by overriding the HWMEM_POOL_SIZE parameter. For example, you can increase the memory pool size to 100,000 bytes using the vxprj command-line utility:

```
% vxprj parameter set HWMEM_POOL_SIZE 100000
```

You can also increase the size of the parameter in the Workbench using the Kernel Configuration Editor (Hardware > Hardware Interface Modules > Pre-Kernel memory Allocation Pool Size).

As an example, the rmi_xlr_mipsi64sf BSP supports 32 SMP CPUs, each having their own CPU interrupt controller and timer driver instances. It defines HWMEM_POOL_SIZE to be 300000.

6.6.10 Debugging Techniques

Obtaining hardware debug support for your multicore system can be challenging. Simulators, ICEs, logic analyzers, and similar applications for multicore parts are not universally available and can evolve rapidly. The following techniques may help you in your debugging efforts:

- Create a low-level print routine that writes directly to the serial output device using polling methods. This is useful for seeing what is happening early in the bootstrap process. Dealing with inter-mixed messages from asynchronous events on multiple CPUs and similar issues are magnified by multicore parts. Serial output is not ideal but can sometimes be the only tool available.

- The VxWorks kprintf() routine may be useful for low-level printing. To use kprintf(), do the following:
  a. Implement a low-level, polled buffer write routine (call it fooBspWrite()) with the following signature:
     ```c
     STATUS fooBspWrite(char * pBuf, size_t len);
     ```
  b. Add the INCLUDE_DEBUG_KPRINTF and INCLUDE_DEBUG_KWRITE_USER components to your VIP project. For instance:
     ```
     % vxprj component add INCLUDE_DEBUG_KPRINTF INCLUDE_DEBUG_KWRITE_USER
     ```
c. Set the `DEBUG_KWRITE_USR_RTN` parameter to the name of your routine (`fooBspWrite()`). For instance:
   ```bash
   % vxprj parameter set DEBUG_KWRITE_USR_RTN fooBspWrite
   ```

d. Now, you should be able to call `kprintf()` (a `printf()`-style interface) in your BSP.

- When bringing up an SMP BSP for the first time, use of a `vxWorks.st` image reduces the possible implications of network interactions until the startup has been debugged.
- Another useful technique while debugging secondary CPU startups is to prevent `usrRoot()` from starting them by setting the `ENABLE_ALL_CPUS` project parameter to `FALSE`. You can enable the secondary CPUs manually from the target shell command line with `kernelCpuEnable`.

### 6.6.11 Building your BSP

SMP BSPs are built with the VxWorks project facility using either the `vxprj` command-line utility or Wind River Workbench.

#### Using the `vxprj` Command

**32-Bit**

To build an SMP BSP using the `vxprj` command-line utility, specify the `-smp` switch in addition to all other required parameters. This causes the SMP runtime libraries to be selected during the link process. For example, to create a VxWorks Image Project (VIP) from the BSP `mySMPBsp` with software floating point using the Wind River Compiler:

```bash
% vxprj create -smp mySMPBsp sfdiab
```

This creates the project in `installDir/vxworks-6.x/target/proj/mySMPBsp_sfdiab`.

To build that project, change your working directory to the project directory and build:

```bash
% cd {installDir}/vxworks-6.x/target/proj/mySMPBsp_sfdiab
% vxprj build
```

Alternately, create a VSB and enable SMP in the VSB configuration. When creating the VIP, specify the path to the VSB directory using the `-vsb` switch. For more information on VSBs, see the `VxWorks Kernel Programmer’s Guide: VxWorks Configuration`.

**64-Bit**

As in the case of SMP BSPs, 64-bit BSP `bootApp` and VxWorks images can only be built using the VxWorks project facility (`vxprj`). BSP directory command line builds are not supported in 64-bits.

All 64-bit BSP use of the `vxprj` command-line utility requires the `-lp64` switch.

To create and build a 64-bit `bootApp` project (builds a `vxWorks_romCompress.bin` image in this example):

```bash
% vxprj create -profile PROFILE_BOOTAPP_BASIC -lp64 my64bitBsp gnu my64bitBsp_bootApp
% cd my64bitBsp_bootApp
```
To create and build a 64-bit vxWorks image project:

```bash
% vxprj build set default_romCompress
% vxprj build vxWorks_romCompress.bin
```

To create and build a 64-bit vxWorks SMP image project:

```bash
% vxprj create -lp64 my64bitBsp gnu my64bitBsp_vip
% cd my64bitBsp_vip
% vxprj build
```

**Using Wind River Workbench**

To create an SMP BSP project for your SMP BSP using Workbench:

1. Create a VxWorks Image Project (File > New > Project)
2. In the Project Setup dialog, click the **Browse** button next to the BSP list and browse to your BSP directory.
3. In the Options dialog, select **SMP support in kernel**.

```bash
% vxprj create -lp64 -smp my64bitBsp gnu my64bitBsp_vipSmp
% cd my64bitBsp_vipSmp
% vxprj build
```

**NOTE:** If you do not see this option, edit the BSP CDF file and make sure that the Bsp statement contains the **MP_OPTIONS SMP** clause. Then re-create the VxWorks Image Project.

4. Click **Finish**.

To build the project, right-click the project folder in the Project Explorer pane, and select **Build Project**.

VSBS may also be created, built, and used by VIPs in Workbench. For more information on VSBS, see the *VxWorks Kernel Programmer’s Guide: VxWorks Configuration*. 
7.1 Introduction

This chapter describes how to extend an existing uniprocessor BSP for VxWorks AMP. It also provides information on extending an SMP BSP to work in an AMP configuration\(^1\). It is primarily intended for users who want to use a processor that is supported by AMP with a board for which there is no pre-existing multicore BSP.

\(^1\) This functionality is available for MIPS processors in VxWorks 6.8 Update Pack 1 and later. This functionality is available for PowerPC processors in VxWorks 6.8 Update Pack 2 and later. This functionality is available for IA-32 processors in VxWorks 6.9 and later. For more information, see your Platform release notes.
For information on which target hardware platforms support VxWorks AMP in this release, see your Platform release notes.

7.2 Overview of BSP Development for AMP

In developing a BSP for AMP, you need to address the following issues that do not arise for uniprocessor BSPs:

- Review and verify VxWorks AMP development prerequisites (see 7.3 Development Prerequisites, p.151).
- Create a baseline BSP and perform initial build configuration (see 7.4.1 Creating and Building a Baseline BSP, p.155).
- Determine your boot loader support and strategy (see 7.4.2 Choosing a Boot Loader Strategy, p.155).
- Allocate address space across CPUs (see 7.4.3 Configuring Memory, p.156).
- Add multicore exception handling (see 7.4.4 Configuring Exception Vectors, p.156).
- Add multicore interrupt delivery (see 7.4.5 Adding General Exception and Interrupt Handling, p.157).
- Provide initial (static) device allocation (see 7.4.6 Allocating Devices, p.157).
- Manage secondary cores during reset and reboot (see 7.4.7 Planning Secondary CPU Handling During Reboot and Restart, p.158).
- Add wrload utility support routines (see 7.4.8 Implementing Support for the wrload Utility, p.158).
- Verify correctness of miscellaneous utility routines (see 7.4.9 Performing Miscellaneous Development Tasks, p.159).
- Add dynamic allocation of devices (see 7.4.6 Allocating Devices, p.157). This is optional but highly recommended.
- Add MIPC support for interprocessor communication (see 7.14 Adding Support for MIPC, p.185). This is optional but highly recommended.
- Update documentation to reflect new features (see 7.4.10 Documenting Your Work, p.159).
- Validate and test your BSP (see 7.4.11 Performing Incremental Development and Testing, p.160).

It is possible to develop multiple BSPs for a single AMP system. For example, for the PowerPC 8572, you can create a BSP for the primary CPU and a separate BSP for the secondary CPU. The approach taken in this chapter is to provide a single BSP that applies to all CPUs on the processor.
7.3 Development Prerequisites

Before beginning your VxWorks AMP BSP development, be sure to have the following items available:

- complete CPU and board documentation (see 7.3.1 Hardware Documentation, p.151)
- VxWorks AMP support for your target CPU (see 7.3.2 VxWorks AMP Architecture Support, p.151)
- a working and validated uniprocessor (UP) BSP for your target hardware (see 7.3.3 Existing BSP Support, p.152)
- a working SMP BSP for your target board (optional for UP AMP configurations but strongly recommended, see SMP BSP Support, p.152)

In addition, you should be sure to implement your code in a way that is consistent with existing VxWorks AMP BSPs. This generally involves creating custom VxWorks components to control VxWorks AMP features. For more information on custom components and general AMP coding conventions, see 7.3.4 Custom Components for VxWorks AMP, p.153.

7.3.1 Hardware Documentation

Before beginning your VxWorks AMP BSP development, you should be sure to have complete documentation for your target board and processor. This documentation is typically available from the processor or board vendor. Key information to have available includes:

- target RAM and memory map information
- primary and secondary CPU start and stop mechanisms and states

**NOTE:** VxWorks boot loaders typically do little or nothing with secondary CPUs (basically, keep them quiescent). This can pose challenges for BSP development. AMP requires configuring, starting, and stopping secondary CPUs in a controlled manner.

- CPU interrupt controller and interrupt routing information

**NOTE:** Implementation of the interrupt controller for AMP is an architecture-level issue and beyond the scope of this guide, but the BSP is responsible for the routing and prioritization of interrupts.

If you do not have this information, contact your hardware vendor.

7.3.2 VxWorks AMP Architecture Support

Typically, adding VxWorks AMP support for a specific CPU requires some architecture-level OS changes, especially in core interrupt and exception management, which is beyond the scope of this guide. You must be sure that your VxWorks release includes AMP support for your target architecture. For information on target architecture support for VxWorks AMP, see your Platform release notes or the Wind River Online Support Web site.
7.3.3 Existing BSP Support

This document assumes that your AMP BSP development builds on a base uniprocessor (UP) BSP or a base SMP BSP if your AMP configuration includes SMP images. You must develop and validate the base BSP before attempting to add AMP support. Your base BSP should be fully tested and validated before you begin your VxWorks AMP development.

You must be sure to have complete Workbench or \texttt{vxprj} build support available in your base BSP, with basic component (CDF) configuration in place before beginning your AMP development. AMP must be built with Workbench or \texttt{vxprj}, and cannot be built by typing make in the BSP directory.

In addition, your BSP should implement VxBus-compliant device drivers (see \textit{Other Required Support}, p.152).

SMP BSP Support

Although it is not strictly required for most configurations, Wind River recommends having a working SMP BSP available for your target hardware before beginning VxWorks AMP development (if using SMP in an AMP configuration, you must start with a working SMP BSP). It is generally an easier transition to move from an SMP BSP to an AMP BSP than to move from a uniprocessor BSP to an AMP BSP. This is because AMP reuses and modifies many BSP features and facilities required by SMP. Within Wind River, AMP BSPs have always been developed on top of a pre-existing SMP BSP. The instructions in this chapter may not be adequate to develop an AMP BSP directly from a UP BSP base. Separate validation of the SMP BSP is a tremendous help in isolating and debugging faults that are independent of AMP.

\begin{itemize}
  \item \textbf{NOTE:} If your AMP configuration includes SMP images, you must base your development for the SMP AMP BSP on a working SMP BSP.
\end{itemize}

\textbf{Other Required Support}

Before beginning your VxWorks AMP BSP development, you should also consider the following:

\begin{itemize}
  \item You must use VxBus-compliant device drivers.
\end{itemize}

Support for non-VxBus drivers in AMP is ad-hoc and beyond the scope of this guide. If you develop a new device driver for an AMP BSP, the driver must follow the VxBus device driver design model and use the VxBus interrupt-configuration API to set up the interrupt delivery for a device (see the \textit{VxBus Device Driver Developer’s Guide}). In addition, the driver must be
specifically designed for multiple CPUs (see C. Writing Device Drivers for VxWorks AMP).

- PCI-bus sharing is not supported.

The current release of VxWorks AMP does not support sharing a PCI bus between CPUs. If you add support for PCI devices to a VxWorks AMP image, you need to ensure that it is not present in any other AMP images.

### 7.3.4 Custom Components for VxWorks AMP

Code that is specific to a feature (whether a sub-feature of VxWorks AMP or not) should be made into a Custom VxWorks component. VxWorks components provide features (such as file systems or networking protocols) that can be configured into the operating system using the standard VxWorks configuration facilities. These components are described in component description files (CDFs) and typically appear in the configuration facility in the form of INCLUDE_FOO. (For more information on VxWorks components, see VxWorks Custom Component and CDF Developer’s Guide.)

Ideally, most of your BSP implementation should be written in such a way that the code can be expressed as custom components. However, this is not easily achieved for low-level initialization (BSP development) and you must often insert code fragments into existing functions. When you add new code that is specific to a particular VxWorks component to existing routines, the code should use ifdef statements that include the component label. For example:

```c
...  
  common code  
...  
  ifdef INCLUDE_COMPONENT_FOO  
  ...  
  special code  
  ...  
  endif /* INCLUDE_COMPONENT_FOO */  
```

In VxWorks, some BSPs use the `_WRS_VX_AMP` token to detect AMP builds and others use the INCLUDE_AMP_CPU token.

See the Makefile, 20bsp.cdf, and target.ref in the reference BSP to determine which token you need to use for your BSP.

When you are writing your AMP code, it is important that the code is written to first determine if the build is for AMP, and then check for a primary CPU (#ifdef INCLUDE_AMP_CPU_00).

You should have two classes of AMP code, no matter how many CPUs are supported: primary and secondary.

For AMP code use the following tokens:

- If the code is for a specific component, use (or create) the correct component tag. For example, if the code is specific to (and only needed by) the wrload utility, use:
  ```c
  ifdef INCLUDE_WRLOAD  
  ```

- If the code is used only by the primary AMP CPU configuration, use:
  ```c
  ifdef INCLUDE_AMP_CPU_00  
  ```
This is defined in an AMP BSP component description file (such as, \texttt{20bsp.cdf}).

\begin{itemize}
  \item If the code is specific to AMP—but no other component—and is always needed by AMP, use one of the following (depending on the BSP):
    \begin{verbatim}
    #ifdef _WRS_VX_AMP
    or
    #ifdef INCLUDE_AMP_CPU
    \end{verbatim}
    This is defined in the AMP BSP makefile (\texttt{Makefile}).
    \end{itemize}

\begin{itemize}
  \item If the code is specific to exactly one image build other than the primary CPU, use the following:
    \begin{verbatim}
    #ifdef INCLUDE_AMP_CPU_nnn
    \end{verbatim}
    \end{itemize}

\begin{itemize}
  \item If the code is used only by the secondary CPU(s), use one the following (depending on the BSP):
    \begin{verbatim}
    #ifdef defined(_WRS_VX_AMP)
    or
    #ifdef INCLUDE_AMP_CPU
    \end{verbatim}
    Together with:
    \begin{verbatim}
    #ifdef INCLUDE_AMP_CPU_00
    \end{verbatim}
    Do not use the token INCLUDE_AMP with \texttt{ifdef} statements in the code. INCLUDE_AMP is an almost empty container component and may be deprecated in a future release.
\end{itemize}

\section*{7.4 Implementation Overview}

This section reviews some of the design and implementation choices that need to be considered when developing a VxWorks AMP BSP. This section provides a detailed overview of the BSP development process. Additional discussion on certain topics is provided in the remaining sections.
7.4.1 Creating and Building a Baseline BSP

Keeping in mind the development prerequisites in 7.3 Development Prerequisites, p.151, one of the first steps in VxWorks AMP BSP development is to establish a baseline BSP. Do this by creating a new BSP based on your reference BSP (the BSP you are basing your work on) then making small changes to create your baseline AMP BSP. This provides you with a stable base for the remaining development work. For more specific information on creating a baseline BSP, see 7.5 Creating an Initial BSP for AMP, p.161.

7.4.2 Choosing a Boot Loader Strategy

Booting multicore CPUs immediately raises all the primitive AMP issues: getting secondary CPUs to a known state, establishing control and communication, and so forth. The vendor-provided boot loader may not support VxWorks, and some mix of vendor boot loader and VxWorks boot loader/boot application may prove to be the fastest and most reliable development or deployment option. Your implementation choices are:

- Use the VxWorks boot loader.
  
  The standard VxWorks boot loader is configured only to start a single CPU of a multicore part, and to leave the other CPUs as nearly off as is possible (this is architecture dependent). Using a VxWorks boot loader generally requires modifications (romInit.s, sysALib.s) to add initialization and startup of secondary CPUs. SMP BSPs typically solve many of these problems, although the code includes ifdef statements for _WRS_VX_SMP and will require some additional porting to work correctly for both SMP and AMP.

- Use a vendor boot loader.
  
  A vendor-supplied boot loader may be a better choice than using a VxWorks boot loader for AMP, depending on a variety of factors. Vendor boot loaders may have richer solutions for CPU-variant and secondary CPU initialization, board configuration, and so forth. Using a vendor boot loader to boot VxWorks in a UP, SMP, AMP UP, or AMP SMP configuration requires an additional learning curve for the vendor boot loader, and additional design and implementation effort to integrate booting VxWorks images. However, the total effort, reliability and long-term maintenance may be better than a custom built VxWorks boot loader, especially if mixing different OSs on a multicore part.

- Mix boot loaders.
  
  One possible development strategy is to use the vendor boot loader to launch a minimal VxWorks boot application which consists of a VxWorks boot loader shell and interface built as a small VxWorks image, then use the VxWorks boot application to launch a full VxWorks image. This provides a short path to getting VxWorks image running on the target, and then provides the standard VxWorks boot prompt mechanism for launching the full VxWorks image. Sometime later in the development process, when the system is up and development is more stable, a deployable version of the boot loader can be developed.
For more information on boot applications, see 4.3 PROFILE_BOOTAPP, p.101 and the VxWorks Kernel Programmer’s Guide: Boot Loader. For more information on boot loader changes for AMP, see 7.6 Changing Boot Behavior for AMP, p.162.

7.4.3 Configuring Memory

The ROM and RAM for your target board must be preconfigured and allocated for each of the OS images that will run concurrently. When considering memory configuration, consider the following points:

- Each VxWorks image requires a contiguous, non-overlapping chunk of RAM. The entire image text, data, heap, and stack must have RAM allocated to it at all times. The total run-time RAM area (LOCAL_MEM_LOCAL_ADRS to LOCAL_MEM_LOCAL_ADRS + LOCAL_MEM_SIZE) is built into the VxWorks image. The capability of using the same binary image on separate CPUs is architecture and configuration dependent. Additional details are discussed below. It is possible to use the same binary for some PowerPC variants, but generally not possible to use the same binary for MIPS or certain PowerPC variants because physical addresses are hardcoded into the ELF file.

- These regions typically have to also avoid existing boot loaders, boot ROMs, and other board-specific restricted regions, along with any other OS that may have additional memory region requirements.

- Separate shared-memory regions, apart from any given image, may also be required for cross-OS data.

All these memory configuration decisions must be made and enforced during design and configuration time. There are few tools to prevent separate images from colliding with each other at run time. Because each image typically runs with highest privilege levels, it is not difficult for one kernel to read or write any memory location, even a memory location owned by some other image.

This stage of development typically involves modifying BSP CDF files for several configuration parameters—for example, LOCAL_MEM_LOCAL_ADRS and so forth—and modifying romInit.s and sysALib.s to guarantee that memory bounds are correctly implemented.

For more information on memory configuration, see 7.12 Allocating Address Space, p.177.

7.4.4 Configuring Exception Vectors

Each VxWorks image must provide its own exception handling. Typically, each image has a separate exception vector area, and the exception vector area is within the image’s local address space.

While it may be technically possible to share or multiplex exception handling between CPUs, this is a poor design choice. The shared vector approach requires each image to have additional dependencies on the behavior of another image, which in general cannot be verified or enforced for independent AMP images. Each AMP image must be allowed to modify its own vectors, through use of VxWorks utilities like excVecSet() for debugging or additional features. Shared exceptions may be modified and can cause other AMP images to fail in
unpredictable ways. Avoid a design which shares exception vectors, as this requires an extensive porting effort and may alter kernel features.

This development stage typically involves changes to `hwconf.c`, `romInit.s`, and `sysALib.s`.

### 7.4.5 Adding General Exception and Interrupt Handling

In a UP or SMP kernel, one run-time image owns everything including managing all exception and interrupt processing (including, possibly, dynamically routing interrupts to different CPUs). When working with VxWorks AMP, all images are logically independent of each other. External interrupts must be routed to, and handled by, a specific CPU with no intervention from any other CPU. Even if interrupts are distributed round-robin, each CPU is acting independently unless some higher-level interprocessor mechanism is overlaid on top of standard OS behavior. This means that AMP introduces issues of device contention, race conditions, and the potential for lost or misdirected interrupts that are not commonly seen on UP, and have different effects on SMP. For more information, see 7.8 Managing Interrupts, p.167.

This stage of development typically involves changes to `hwconf.c`, `romInit.s`, and `sysALib.s` and then implementing/porting the IPI subsystem (several BSP-dependant files).

### 7.4.6 Allocating Devices

As with memory and exceptions, any unique device—for example, the serial line or network interface—can only be “owned” by a single VxWorks image. If several running kernels try to initialize and use the same unique device, behavior is likely to be undefined or potentially fatal to the entire board. For example, each CPU could try to own and reconfigure the chip-wide interrupt controller which could break the CPU interrupt configuration for every other CPU.

This stage typically involves modifications to `hwconf.c` and `sysLib.c`. The `sysHwInit()` and `sysHwInit2()` routines typically need to treat device initialization for physical CPU index 0 (bootstrap CPU) as a special case. Specific device driver changes may also be required by AMP (see C. Writing Device Drivers for VxWorks AMP).

For more information on device allocation, see 7.13 Assigning Devices, p.181.

**Device Driver Limitations and Design Considerations**

VxWorks AMP device drivers may not support all potential configurations (for example, allocating network ports among multiple CPUs). Such device or driver limitations or restrictions must be clearly understood and documented. You must make design choices about device allocation to specific CPUs.

**NOTE:** Note that modifying device drivers for complex AMP configurations is beyond the scope of this guide. For more information on device driver development, see the VxBus Device Driver Developer’s Guide.
Incremental Implementation

Device allocation can be implemented incrementally. During the initial implementation of an AMP BSP you may want to hardcode device allocation to specific CPUs for development speed and simplicity. Most devices are correctly encapsulated as build-time selectable components (for example, DRV_SIO_NS16550 serial port) so that during early development each image can be built with specific components to allocate devices. Later in your implementation, after the basic BSP is stable, you can implement dynamic device allocation. To simplify configuration and build, VxWorks provides a mechanism for run-time allocation of devices based on CPU ID and device ID (see The sysBspDevFilter() Routine for Run-Time Device Assignment, p.182).

Special Cases for Development and Debug

Providing a serial console for secondary CPUs during development and debug is highly desirable in most development scenarios. A simple direct-to-serial polling print routine (such as xprintf() or kprintf()) is also useful. This allows you to see debug output before interrupts are enabled on the secondary CPU.

7.4.7 Planning Secondary CPU Handling During Reboot and Restart

The final fundamental design issue for AMP is the highly hardware-specific secondary CPU behavior on reset from the running VxWorks image. That is, what happens with ^X or reboot from the secondary CPU console. These details depend on decisions discussed above and require deep understanding of the hardware. However, some general VxWorks requirements must be taken into account:

- Reboot of the VxWorks image running on physical CPU index 0 (the bootstrap CPU) resets the entire board—all other CPUs shut down.
- Reboot of a VxWorks image not running on physical CPU index 0 reboots only the CPU(s) assigned to that image, leaving no critical transactions unfinished or resources locked that will cause other CPUs to deadlock or halt.

This development stage typically involves creating utility routines that can be plugged in to sysToMonitor().

For more information, see 7.6.2 Booting a Secondary CPU, p.163.

7.4.8 Implementing Support for the wrload Utility

Once you understand all of the above design considerations, you must add two routines to enable the wrload utility to download and start the secondary CPUs:

- sysAmpCpuPrep() puts a secondary CPU into a state that allows the wrload utility to download and initialize an image.
- sysAmpCpuEnable() starts the secondary CPU running the downloaded image.

In some BSPs, these routines have been added to sysLib.c but better practice is to create a separate module. You can also specify within the BSP how the wrload utility should handle memory accesses to the secondary CPU.
7.4 Implementation Overview

For more information on adding support for the wrload utility, see 7.11 Adding Support for the wrload Utility, p.170.

7.4.9 Performing Miscellaneous Development Tasks

Although the tasks in this section are not specific to BSP development for VxWorks AMP, they are important development considerations. You should review these routines and verify that they have been implemented properly.

sysMemTop()

sysMemTop() defines the first address at the top of memory that is not usable by the kernel. Depending on the implementation in your base BSP, the sysMemTop() routine may require changes. sysMemTop() is implemented in some BSPs with hardcoded addresses or fixed offsets from physical memory limits. If this is the case in your reference BSP, the implementation for this routine must be changed to honor LOCAL_MEM_LOCAL_ADRS + LOCAL_MEM_SIZE (sysPhysMemDesc), USER_RESERVED_MEM, and PM_RESERVED_MEM. Otherwise, the routine could point into another image space entirely. Usually, sysMemTop() will equal sysPhysMemTop - USER_RESERVED_MEM - PM_RESERVED_MEM.

sysProcNumSet() and sysProcNumGet()

The sysProcNumSet() routine must change the sysProcNum variable. The sysProcNumSet() routine is called by bootLineParse() with the bootline. The sysProcNumGet() routine must return sysProcNum.

The most common VxWorks AMP error is to hardcode sysProcNum, sysProcNumSet(), or sysProcNumGet() to use the CPU index rather than being variable as set by the bootline. Such an implementation fails in subtle ways. For example, MIPC will map its nodes incorrectly.

7.4.10 Documenting Your Work

Part of the overall BSP development process involves documentation (see 3.3.8 Updating BSP-Specific Documentation, p.96). You should verify that manual pages are generated for new or significantly modified BSP routines. You should also update the BSP target.ref file with AMP-specific details. This usually involves a detailed discussion of the following items:

- Boot mechanism changes.
- Additional or modified configuration parameters.
- AMP-specific device allocation and restrictions or constraints on device allocation.
- Optional feature and product support (for example, MIPC).
### 7.4.11 Performing Incremental Development and Testing

Developing an AMP BSP is challenging. In one sense, it is as simple as replicating a UP image on another CPU (making it nominally “easier” than SMP development). However, it is difficult to see what a secondary CPU is doing as it comes up. This adds a layer of complexity to the development process.

#### Planning Your Development Strategy

When planning your overall development strategy, you should note that some stages of development are difficult to decompose into incremental tasks. That is, several different parts are required to work together or nothing goes on in the system. Even so, basic engineering principles apply:

- Design first.
- Tackle the implementation in small steps.
- Incorporate peer review and conduct unit testing often during development.
- Update your overall design as issues are resolved.

Obtaining hardware debug support for your multicore system can also be challenging. Simulators, ICEs, logic analyzers, and similar applications for multicore parts are not universally available and can evolve rapidly.

**NOTE:** Describing available hardware debug support for multicore boards is beyond the scope of this guide.

You may find it helpful to create your own low-level debug print routines. Creating a `xprintf()` or similar routine that writes directly to serial output is useful for seeing what is happening early in the bootstrap process for secondary CPUs. Dealing with inter-mixed messages from asynchronous events on multiple CPUs and similar issues are magnified by multicore parts. Serial output is not ideal but can sometimes be the only tool available.

#### Planning for Testing and Validation

Validation and testing of AMP images requires many manual steps. The BSP Validation Test Suite (BSP VTS) provides a broad suite of BSP functional tests, but is configured only for UP or SMP on the primary CPU by default. Also, the BSP VTS does not address several AMP-specific test cases. For example:

- As mentioned previously, the VxWorks BSP VTS is not automatically configured for AMP. However, it can be configured and run manually on secondary CPUs.

- Device allocation, especially multiple network interfaces spread over multiple CPUs (if supported), requires manual configuration and verification. There is no support for this type of testing in the BSP VTS.

- Stress-testing a VxWorks AMP system, IPI delivery, and other cross-CPU interactions is ad-hoc and manual. There is no support for this type of testing in the BSP VTS.
You should also note that optional technologies such as Wind River MIPC may require additional testing or validation beyond what is supported in the BSP VTS. For more information on testing optional technologies, see the related product guides.

### 7.5 Creating an Initial BSP for AMP

BSPs generally share as much code as possible with their UP and SMP equivalents. In some cases, the BSP code changes involved with porting to VxWorks AMP are limited to a few dozen lines of code in the entire BSP. However, the makefile and configuration files are nearly always different for VxWorks AMP regardless of the base BSP. For this reason, you can typically start the initial BSP creation by making changes to those files.

You can begin your initial BSP development for AMP by doing the following:

1. Create a new BSP directory.
   
   For AMP BSPs, the recommended naming convention is to use the base BSP name plus an AMP designation (for example, `ads8572_AMP` or `cav_cn3xxx_AMP_mipsi64r2sf`).

2. Link source files from the base BSP to the AMP BSP.
   
   A symbolic link of the original source files will make it easier to incrementally test both the original UP or SMP BSP and the new AMP BSP in parallel. Several source files will change during development. However, the original BSP should always build and run without any configuration changes. This guarantees that the AMP modifications are properly separated. Symbolic linking can be done by entering the following command from a VxWorks development shell:
   
   ```bash
   -> ln -s src_file target_link
   ```

3. Replace the following files with local copies:
   
   - Makefile
   - 20bsp.cdf

4. Edit `Makefile` and add the following line:
   
   ```make
   EXTRA_DEFINE +=-D_WRS_VX_AMP
   ```

5. Edit the `20bsp.cdf` and change the `Bsp` component (typically the first component in the file) to match the new directory name for the AMP BSP.

   Once you have made the above changes, refresh your environment and build a VIP with the newly created BSP. It should compile exactly the same as the original. Verify that the newly created image boots on a target. This establishes your baseline for making further modifications.

**NOTE:** This makefile change should not be applied to BSPs that use `#ifdef INCLUDE_AMP_CPU`. For more information, see the `target.ref` file for your reference BSP.
7.6 Changing Boot Behavior for AMP

One of the greatest obstacles you must overcome when developing an AMP BSP is getting a multicore board out of reset with all of the CPUs and CPU-interconnects initialized correctly for VxWorks. If the processor and board are not sufficiently initialized, it may not be possible to start additional VxWorks AMP images. Conversely, if too much has been initialized, or things have been initialized in such a way that all hardware resources are completely allocated, the VxWorks images may not boot correctly.

Consider that on multicore boards, one CPU is considered to be the master. Typically, the master CPU is the only CPU enabled out of reset and all other CPUs do not execute any instructions until explicit steps are taken by the master. By convention, the master CPU is called “physical CPU index 0”, the bootstrap CPU, or the primary CPU, and all other CPUs are referred to as secondary.

With those terms in mind, the VxWorks bootstrap process out of hardware reset must achieve the following state:

- The primary CPU must be running VxWorks (only on the primary CPU).
- The secondary CPUs must be quiescent and must be able to remain in that state indefinitely without any adverse affects on the system.
- The secondary CPUs are ready to respond to wrload requests from the primary CPU.
- All system resources are either initialized, allocated and managed by the primary CPU, or are available for secondary CPUs to initialize and manage when the secondary CPU boots. In unsupervised AMP, nothing—other than careful system design—precludes secondary CPUs from trying to claim any or all system resources.

The remainder of this section expands on the strategies and details of how to achieve this state using either a VxWorks boot loader or existing vendor support. It also discusses AMP-specific boot details for both primary and secondary CPUs.

7.6.1 Booting the Primary CPU

In VxWorks, the primary CPU has ownership of, and responsibility for, several critical chip-wide and board-wide resources. For example, the primary CPU must configure the system interrupt controller and at least all of the resources it will use directly. On multicore systems, the primary CPU typically must also configure—either minimally or completely, depending on hardware—any multicore interconnect busses, mailboxes, or similar. For example, the RMI (MIPS) XLR primary CPU must partially configure and start the chip-wide Fast Messaging Network (FMN) bus. Therefore, the primary CPU almost always has some unique functionality that is not shared by secondary CPUs.

When considering a boot strategy using the VxWorks boot loader or a strategy using the vendor boot loader to launch VxWorks, note that only the primary CPU executes romInit()—the low-level boot loader board initialization (see romInit.s, p.22). After that, there are parallel but slightly different paths through sysInit() (see sysALib.s, p.22) for each of the CPUs. As discussed previously, the primary CPU has hardware-specific requirements and responsibilities for a multicore part.
After that, the primary CPU is assumed to continue booting to the multitasking state.

7.6.2 Booting a Secondary CPU

For VxWorks AMP, secondary CPUs must be held relatively quiescent out of reset until requested to start using the \texttt{wrload} utility. Note that “relatively quiescent” means the secondary CPU is most likely spinning somewhere early in its own bootstrap process (see below). Once started using \texttt{wrload}, secondary CPUs essentially run through the same bootstrap process as uniprocessor VxWorks would on a single CPU board. That is, the secondary CPU bootstrap process will make the CPU interrupts quiescent, set the local memory map and initialize memory, initialize exception vectors, enable CPU-specific clocks and devices, and so on. When VxWorks AMP is running on a secondary CPU, it treats most hardware resources as if it has exclusive access. Therefore, secondary images must be built to explicitly limit their resource usage.

VxWorks AMP requires secondary CPUs to \texttt{wait}—that is, not complete booting of VxWorks, even to the end of \texttt{sysInit}()—until explicitly started using the \texttt{wrload()} routine. This can be delicate because in most cases, the secondary CPU is running some code from the boot loader and must be held back from proceeding. While it is possible to start secondary CPUs programmatically (for example, by calling \texttt{wrload()} from \texttt{usrAppInit()} or similar), secondary CPUs must not be allowed to simply jump to \texttt{sysInit()} out of reset and come up to full multitasking or application state.

Holding the Secondary CPU on Startup or Reset

The strategy used for holding a secondary CPU is hardware-dependent but generally falls into one of three models:

- **Option 1**: Spin in the VxWorks boot loader, typically in \texttt{sysInit}(), waiting for updates to well-known memory locations, shared semaphores, or something similar that then informs the secondary CPU on how it should proceed. This strategy is used by the PowerPC \texttt{hpcNet8641d}.

- **Option 2**: Spin in a vendor boot loader and then use the vendor-supplied mechanism to jump to and launch a VxWorks image. This strategy is used by the MIPS RMI XLR/XLS boards.

- **Option 3**: Leave the secondary CPU suspended (literally doing nothing) until it is enabled with information on how to boot VxWorks. MIPS Cavium boards can use this mechanism.

Each of these strategies is expanded below. Note that regardless of what mechanism you choose, you must provide the same functionality in the BSP. That is, you must always hold the secondary CPU at the earliest possible state.

- **Option 1: Spin in the VxWorks Boot Loader**

  In this scenario, a secondary CPU branches to code that spins reading some well-known memory locations waiting for the primary CPU to write a known-good set of values. This happens as early as possible in \texttt{sysInit}(). In a typical situation, 3 to 4 words in shared memory are used to encode how and where the secondary CPU should start—the entry point and state. This shared
memory is also used to enforce synchronization. For example, to avoid a secondary CPU starting with incomplete or corrupt data, the words are written by the primary CPU in reverse order from the order that they are read by the secondary CPU. (For an example of this, see the `hpcNet8641d BSP waitForInitStartSet` in `romInit()`.)

The secondary CPU should also modify the encoded locations in shared memory as soon as possible to a different well-known state, so that failure to launch does not simply loop back to `sysInit()` and retry with bad values. Optionally, those memory locations can be used to share state information with the primary CPU as the secondary CPU boots. (Wind River does not provide an API for this.)

This mechanism is already common in many BSPs that support VxWorks SMP. In most cases, the same or only slightly modified code should work for VxWorks AMP as well.

- **Option 2: Spin in the Vendor Boot Loader**

  This option is logically similar to option 1, but depends on a third-party API for reset and start details. Generally, the vendor API requires information about the entry point and any flags passed to the image, CPU number, and possibly other options. This mechanism is used by the RMI XLR/XLS family with the RMI boot loader.

  Vendor boot loaders may include more sophisticated auto-detection and configuration of the hardware when compared to VxWorks boot loaders. In addition, vendor-supplied solutions tend to offer better support and maintenance because of their focus on, and vested interest in, specific hardware. However, vendor-supplied solutions may also be substantially more complex and may not offer the same level of backward compatibility between hardware releases that is found in VxWorks boot loaders. You should consider these issues when choosing your boot strategy.

- **Option 3: Leave the Secondary CPU Suspended or Not Running**

  There are CPU-specific halt and start function available for some hardware and firmware. For example, MIPS Cavium CNxxxx families can be halted by writing a bit to a system register. In this case, restart of an individual CPU usually involves significant work to set up the CPU correctly and the vendor’s SDK provides a public API that hides the details.

  **NOTE:** Although this option is available, it is usually not enough to simply shut a CPU down, especially if it is participating in a running system with potentially unwritten data, outstanding inter-CPU transactions, and similar unfinished work. Depending on the vendor API, the BSP may still be responsible for flushing data, yielding certain hardware resources, and other mandatory clean up, and also for any VxWorks or application-specific clean up.

For all three of the above cases, the same mechanism is also used by `sysToMonitor()` to handle halting/rebooting the secondary CPU, and by the `wrload` utility to prepare and start the secondary CPU.

At some point, the secondary CPU is awakened by a `wrload()` call from the primary CPU. The time is indeterminate. During development, the secondary CPUs are often started from the primary CPU console shell minutes or hours after the primary CPU boots. Regardless of how the CPU comes out of waiting, the next
VxWorks code to be run is `sysInit()`, perhaps first completing a loop in the VxWorks boot loader, depending on the wait strategy.

**Booting the Secondary Image**

The secondary image boot is nearly the same as the primary image but there are some modifications that may be required to port the BSP to AMP. The cache, initial interrupt state, and so forth are likely to be identical for all CPUs, but memory and exceptions can require changes for secondary CPUs. The remainder of this section describes how the boot process for the secondary image differs from the boot process for the primary image.

**Initializing Secondary Image Memory**

VxWorks requires a contiguous chunk of RAM for the image text, data, stack, and heap. This memory region must not overlap with any other image space. The VxWorks memory region must depend entirely on the values of the BSP configuration parameters, `LOCAL_MEM_LOCAL_ADRS` and `LOCAL_MEM_SIZE`. (For more information on these parameters, see the *VxWorks Kernel Programmer’s Guide: Memory Management* and 2.3.5 Required Macros, p.41.)

`LOCAL_MEM_LOCAL_ADRS` defines the bottom of memory used by the image. This points to a location below exception vectors, saved boot line, (optional) error detection and reporting logs, and so forth. `LOCAL_MEM_LOCAL_ADRS + LOCAL_MEM_SIZE` (zero-based) defines the top of memory accessible by the current image. `RAM_LOW_ADRS` points to the entry point, typically `sysInit()`, and must lie within the region described above (`LOCAL_MEM_LOCAL_ADRS + LOCAL_MEM_SIZE`), as must any other constant addresses. Generally, each concurrent VxWorks AMP image must have separate values for `LOCAL_MEM_LOCAL_ADRS` and `RAM_LOW_ADRS` that preserve these requirements. This means that the same binary VxWorks AMP image cannot run on multiple CPUs concurrently.

**NOTE:** Some hardware supports hard-wired remapping of low memory addresses for each CPU. For example, the PowerPC 8641d “low memory offset mode”—a mode which is typically wired to a jumper on the board—makes each CPU see a separate chunk of RAM at base address 0. In this case, the configuration of `LOCAL_MEM_LOCAL_ADRS` and `RAM_LOW_ADRS` can be identical for two or more concurrent images—and may appear to violate the restriction above—but in fact each image is still using a non-overlapping contiguous chunk of RAM. The advantage of such hardware is that (with some additional steps for devices, described in The sysBspDevFilter() Routine for Run-Time Device Assignment, p.182) the same binary VxWorks AMP image can run on multiple CPUs concurrently.

When porting an existing BSP to AMP, the only work typically required to set the memory region definition is to verify that nothing in the existing BSP incorrectly assumes that the image area starts from address 0 or some fixed offset from 0. Any such assumption must be changed to depend on the value of `LOCAL_MEM_LOCAL_ADRS`. Otherwise, it is usually sufficient to verify `sysInit()` and closely related startup code (such as the jump to `RAM_LOW_ADRS`) and, in some cases, `sysPhysMemDesc[]` in `sysLib.c`. No code changes are likely. Unfortunately, a bug here can go undetected unless all memory accesses outside of the defined region are detected and trapped.
The global symbols, `sysAbsSymPhysMemBottom` and `sysAbsSymPhysMemSize`, are also initialized to `LOCAL_MEM_LOCAL_ADRS` and `LOCAL_MEM_SIZE` respectively. This is seen in:

```
installDir/vxworks-6.x/target/config/comps/src/sysComms.c
```

The `wrload` utility uses these two symbols to determine the bounds of the downloaded image. They should never need to be changed by the BSP.

### Initializing Secondary Image Exception Vectors

The requirements and strategies for initializing secondary image exception vectors are hardware dependent and must typically be done early in `sysInit()`, after the `wrload` utility releases the CPU to continue booting and before enabling any exceptions.

Most multicore CPUs support per-CPU (per-thread) exception vectors. For example, MIPS Cavium and RMI parts must initialize the exception base, `EBASE`, to an offset from `LOCAL_MEM_LOCAL_ADRS` early in `sysInit()`. PowerPC must perform a similar configuration of `IPVR`.

Once this is complete, the secondary CPU should be ready to jump to `usrInit()` and higher level AMP initialization.

### 7.7 Modifying `sysPhysMemDesc[ ]`

Use the `sysPhysMemDesc[ ]` structure, in `sysLib.c`, to initialize the page table entry (PTE) array required by the MMU to translate addresses (mapped images only). This structure must be examined. If the mapping only covers RAM and is based on `LOCAL_MEM_LOCAL_ADRS` and `LOCAL_MEM_SIZE`, it may need little or no modification.

You may need to make changes if the mapping is based on physical addresses (typically, physical 0 to maximum memory size) rather than `LOCAL_MEM_LOCAL_ADRS` and `LOCAL_MEM_SIZE`. You may also have to make changes for memory mapped hardware spaces that cannot be safely shared by AMP images. For example:

- Device or I/O space that is not controlled with `ifdef` statements for a component (for example, `INCLUDE_PCI_BUS`\(^2\), USB, and so forth). Wrap such mappings with appropriate component `ifdef` statements.

- Secondary physical memory, such as DRAM, SRAM, flash, and so forth. Generally, you should restrict such devices to the primary CPU, or else they must be divided into specific regions for each CPU to have exclusive access.

Any absolute or non-component-protected (not controlled by `ifdef`) regions cause the same physical space to be mapped into multiple AMP images, which may or may not be the intended design. This is system-design specific, there is no general rule.

---

2. PCI cannot be shared by AMP images (see Other Required Support, p.152). It is used here as a general example.
Additionally, the required changes for this structure are architecture-dependent. For example, PowerPC boards may require that the base region depend on absolute physical memory for the primary CPU, and on hardware “auto-mapped” memory for secondary CPU(s). For an example of this, see the hpcNet8641d_AMP BSP.


7.8 Managing Interrupts

VxWorks AMP provides support for interrupt connection and delivery. As noted earlier in the chapter, VxWorks AMP requires that all required device drivers adhere to the VxBus device driver model and use the VxBus interrupt-configuration API to set up interrupt delivery to devices.

**NOTE:** As of VxWorks 6.8, Wind River MIPC and its associated drivers (such as MSD and MND) no longer use the VxBus device driver model. However, you must continue to use the VxBus device driver model for other device drivers.

The most important task in configuring interrupt delivery in a VxWorks AMP environment is ensuring that device interrupts are delivered to the correct CPU. This is straightforward when each device has its own dedicated interrupt signal, as illustrated in Figure 7-1.

The interrupt controllers for the BSPs supported in the current release provide pins for dedicated interrupt routing. When interrupts from separate devices need to go to separate CPUs, interrupt controllers that use wire-or interrupt routing are not suitable for multicore processors, as illustrated in Figure 7-2.
In Figure 7-2, two serial ports send wire-or-ed interrupts to the interrupt controller. The interrupt controller then delivers the merged interrupt signal from the two serial ports to the two CPUs in the system. When the interrupt is asserted, both CPUs start interrupt processing. However, only one of the two CPUs has an interrupt service routine attached to the interrupt that can properly respond to the hardware's interrupt request. The other CPU cannot perform any action to service the interrupt.

### 7.8.1 Supported Interrupts

**CPU-Generated Interrupts**

Some interrupt-generating devices reside within the CPU, for example, the PowerPC decrementer. Such interrupts do not pass through the interrupt controller and are therefore safe to use in a VxWorks AMP system.

**PLB Device Interrupts**

Some devices reside on the processor local bus (PLB) and generate interrupts that are hardwired directly to the system interrupt controller. Such devices are supported by VxWorks AMP, as long as the devices generate interrupts that can be uniquely delivered through the system interrupt controller. This is the case, for example, if multiple PLB devices share an interrupt line in a wire-or configuration, but only one of the devices is used.

### 7.8.2 Unsupported Interrupts

The current release of VxWorks AMP does not support the following types of interrupts:

- PCI-device interrupts.

  In addition, the current release does not support PCI-bus sharing (see Other Required Support, p.152).
7.9 Modifying the sysHwInit() Routine

If you are using a VxWorks boot loader to hold your secondary CPUs (see *Holding the Secondary CPU on Startup or Reset*, p.163), you must update the shared memory locations used to pass the boot information. You must also exclude any non-sharable hardware devices (for example, UART and network devices) from initialization by the secondary CPUs.

You should make these modifications using VxWorks components so that each image can include or exclude the appropriate components at build time. However, this is not always feasible. In these cases, you may need to either `ifdef` your code directly (see *7.3.4 Custom Components for VxWorks AMP*, p.153) or dynamically detect the CPU number at run time (if the same binary can be used on multiple CPUs).

For more information on `sysHwInit()`, see `sysLib.c: sysHwInit()`, p.64.

7.10 Modifying sysToMonitor()

VxWorks AMP requires new logic to handle shutdown and reboot of secondary CPUs. For UP, reboot or `^X` causes the system to go back through the boot loader and normal boot sequence either to the boot prompt or, if auto-boot flags are set, to reboot to multitasking. For VxWorks AMP, you must make the modifications described in this section.

7.10.1 Changes for the Primary CPU (Bootstrap CPU)

Stopping or rebooting the primary CPU must stop all secondary CPUs first, and then reboot the primary CPU only. The secondary CPUs are left in the wait state defined by the reboot strategy (see *Holding the Secondary CPU on Startup or Reset*, p.163). Generally, you can implement primary CPU reboot as a board-wide reset. The only (optional) change for the primary CPU path is to add some explicit code to shut down the secondary CPUs if it is unsafe to assume that they will stop otherwise.
7.10.2 Changes for the Secondary CPUs

Stopping a secondary CPU or CPUs for AMP SMP configurations—for example, using \(^X\) or a reboot command from its console—must stop only the specific CPU(s) specified and must leave the primary and all other secondary CPUs running and stable. This implementation typically involves the following details:

- Under ideal circumstances, you should notify any applications of the pending shutdown. (There is no standard Wind River-supplied API to do this.)
- Set or verify any sentinels or other reboot states to keep the CPU in the hold state while waiting for the `wrload` utility to restart it.
- Disable interrupts to and from other CPUs.
- Disable and flush caches, flush write buffers, and perform anything else needed to complete any outstanding transactions with other CPUs or devices.
  
  Frequently, there are board and CPU-specific timing issues to address in order to allow buffers time to flush or devices time to become quiescent.
- Optionally, release board and hardware resources. This is good practice but not a critical change.
- Put the CPU back into the appropriate board-dependent reset state, according to your boot strategy. This is hardware specific. Some examples are as follows:
  
  - PowerPC targets typically jump back to the boot loader, and the secondary CPU proceeds to its loop waiting for sentinels to tell it how to boot.
  - MIPS Cavium targets can simply shut the CPU down.
  - Current Wind River-supplied implementations for MIPS RMI targets just spin but this is not correct behavior. Instead, the CPU should return either to the VxWorks boot loader wait code in `romInit()`, or to the RMI boot loader (depending on the chosen boot strategy). This will be addressed in a future release.

  For MIPS RMI targets in VxWorks 6.8 Update Pack 1 and later, rebooting any VxWorks image in the system, reboots the entire system regardless of which CPU the image is running on (primary or secondary).

7.11 Adding Support for the wrload Utility

This section describes work that must be done in a BSP to support the VxWorks `wrload` utility. For more information on the `wrload` utility (such as information on the available command-line switches for `wrload`), see VxWorks Kernel Programmer’s Guide: Booting VxWorks for AMP.

The `wrload` utility is responsible for the following tasks:

- Downloading a VxWorks image from the primary CPU into a secondary or higher CPU’s memory.
- Optionally, modifying the bootline or other initialized data in the downloaded image prior to boot.
Typically, the BSP implements the following two routines to support `wrload`:

```c
sysAmpCpuPrep()
```

The prototype for this routine is as follows:

```c
STATUS sysAmpCpuPrep(UINT32 cpuId, void * arg)
```

This routine verifies the secondary CPU is ready for download, and optionally performs any tasks necessary to make it so. This typically involves putting the secondary CPU into a quiescent state where interrupts are disabled. Depending on your architecture, it may stop the CPU near the end of the process.

```c
sysAmpCpuEnable()
```

The prototype for this routine is as follows:

```c
void sysAmpCpuEnable(FUNCPTR entryPt, UINT32 cpuId)
```

This routine provides hardware-specific code to start (boot) the secondary CPU. As the routine executes, the secondary CPU jumps to a specific entry point and starts running.

**NOTE:** Support for the `wrload` utility is only required on the primary CPU.

By default, `wrload` assumes that the secondary core memory is available for use without any special operations (that is, `wrload` performs direct memory reads and writes using the `read()` routine). If access to a core’s memory requires special processing—for example, to map and unmap the secondary core memory into the primary core memory space (it is generally a good practice to isolate each core’s memory from the others), appropriate BSP memory access routines must be registered for use by `wrload`.

If required, the `sysAmpCpuPrep()` and `sysAmpCpuEnable()` routines can also be overridden. For more information, see Registering the `wrload` BSP-Specific Routines, p.175.

For information on `wrload` support (including limitations and the specific implementation), see the reference entry (`target.ref` file) for your BSP.

### 7.11.1 `sysAmpCpuPrep()`

The purpose of the `sysAmpCpuPrep()` routine (or any custom routine that replaces it, see `wrloadCpuRtnSet()` usage in Registering the `wrload` BSP-Specific Routines, p.175) is to inform the `wrload` utility that a CPU is available and ready to download and start running a new image.

**NOTE:** The `sysAmpCpuPrep()` routine is not called when `wrload()` is called with the `-nostart (-ns)` flag.

If the routine returns `ERROR`, `wrload()` returns `ERROR` and does not attempt to download a new image or take any other action. If the routine returns `OK`, the `wrload` utility is allowed to proceed with downloading a new image, configuring the CPU for boot, and starting it running.

**NOTE:** If some other image is currently running on the secondary CPU, `wrload` destructively overwrites the currently running image (text and data) without any further warning.
There is no strict requirement that \texttt{sysAmpCpuPrep()} must stop the secondary CPU. The routine can be implemented to check whether the CPU is running or not, or anything else the system designer requires. That said, the typical implementation of \texttt{sysAmpCpuPrep()} causes the secondary CPU to stop and return to its wait-for-boot state. The implementation generally calls or duplicates \texttt{sysToMonitor()} on the secondary CPU.

The \texttt{sysAmpCpuPrep()} routine returns \texttt{ERROR} if the CPU index passed to the routine is that of the primary CPU. \texttt{sysAmpCpuPrep()} cannot halt the primary CPU, nor any CPU that is part of an SMP system that includes the primary CPU. If your BSP also supports VxWorks SMP, then \texttt{sysAmpCpuPrep()} must return \texttt{ERROR} if it is targeted at one of the CPUs used by SMP if the SMP image is running on a set of CPUs that includes the bootstrap CPU. If the SMP AMP image is running on a set of CPUs that does not include the bootstrap CPU, the image can be reloaded with a new image. The \texttt{sysAmpCpuPrep()} routine should also return \texttt{ERROR} if the state of the secondary CPU cannot be determined (that is, if the target CPU may not be ready to start a new image).

\begin{itemize}
\item If the CPU is spinning in the VxWorks boot loader waiting for a sentinel, take only those steps strictly required to update that sentinel with boot information so that the secondary CPU boots.
\item If the CPU is spinning in the vendor boot loader, take only those steps strictly required to have the vendor boot loader start the CPU at the provided entry point.
\item If the CPU is stopped or halted, take only those steps strictly required to start the CPU executing at the provided entry point.
\end{itemize}

In each of the above cases, “strictly required” implies that \texttt{sysAmpCpuEnable()} must not be doing anything that is properly in the domain of \texttt{sysInit()} or later code in VxWorks. Do not configure memory, interrupts, exceptions, or anything else in \texttt{sysAmpCpuEnable()} unless it is strictly required by the secondary CPU and impossible to do during the normal flow of the VxWorks bootstrap process. Any steps that prepare the CPU should be performed as part of the \texttt{sysAmpCpuPrep()} routine (or its equivalent). The \texttt{sysAmpCpuEnable()} routine should include only those instructions required to release the CPU from the idle or waiting state.

\begin{itemize}
\item The SMP AMP image can only be run on a set of CPUs that does not include the bootstrap CPU for MIPS processors in VxWorks 6.8 Update Pack 1 and beyond and for PowerPC processors in VxWorks 6.8 Update Pack 2 and beyond. This functionality is not available for other architectures.
\end{itemize}
This section describes various facilities that `wrload` provides for use with BSPs.

**wrload Symbol Passing Facility**

The `wrload` symbol passing facility is a way for the primary CPU to retrieve information located within the secondary ELF image that is loaded by `wrload`. This is useful for cases such as loading above the 32-bit boundary, where the 36-bit or 64-bit physical address for placing the image into memory must be provided in addition to the image's embedded 32-bit virtual program addresses.

The mechanism works as follows:

1. The BSP encodes the information it wants to pass by means of absolute symbols whose name begin with the `_wrload_` prefix.
2. `wrload` retrieves these symbols while parsing the ELF image. It then stores them for later retrieval by the primary CPU.
3. The primary CPU can then call the `wrloadSymValueGet()` routine with the symbol name as an argument, to retrieve the value it needs. This is generally done within the `xxxCpuPrep()` and `xxxCpuEnable()` routines. If the symbol value is also needed by the secondary CPU, such as for its startup code, the primary CPU can place the value into a shared memory location which the secondary CPU is programmed to check.

For example, you could use the following code in your BSP to define RAM size and memory offset values for later retrieval within the `wrload`-registered routines. The actual values to encode are contained within the `ramSize` and `memOffset` integers:

```c
_WRS_ABSOLUTE_BEGIN (wrloadSymbolBlock)
_WRS_ABSOLUTE (_wrload_ramSize, ramSize);
_WRS_ABSOLUTE (_wrload_memOffset, memOffset);
_WRS_ABSOLUTE_END
```

The `_WRS_ABSOLUTE_BEGIN` and `_WRS_ABSOLUTE_END` markers are compiler macros that ensure the appropriate instructions are generated for the compiler to create absolute symbols.

This code can then be used to retrieve the RAM size value. `id` is the opaque `wrload` ID which is passed to the hook routine that was registered with `wrload` either from `usrWrloadInit()`, by defining corresponding `WRLOAD_CPU_xxx_FUNC` parameters, or with a manual call to `wrloadCpuRtnSet()`. If you are using the standard `sysAmpCpuPrep()` and `sysAmpCpuEnable()` routines, you should give `wrloadSymValueGet()` the CPU index that is passed to these routines in place of a real `wrload` ID.

```c
UINT32 ramSize;
status = wrloadSymValueGet (id, "_wrload_ramSize", (void *) &ramSize,
                              sizeof (ramSize));
```

If `status` is `ERROR`, the symbol cannot be retrieved. Doing a symbol dump on the ELF image being loaded by `wrload` and searching for symbols beginning with `_wrload_` will allow you to check whether the symbol was correctly embedded within the image. You can also check if the specified value width is correctly supported by `wrloadSymValueGet()`. It currently supports 1, 4 and 8 bytes only.
The physical load bias is an example of information which is passed in this way and is available for use by the primary and secondary CPU code.

**Loading an Image Above 32 Bits**

In VxWorks 6.9 Update Pack 2 and later, the `wrload` utility can be used to load a 32-bit VxWorks image above the 4 GB memory region (32-bit address space). This feature is in support of 36-bit addressing on targets such as the Freescale QorIQ P4080, and 64-bit addressing on targets such as the Cavium Octeon CN58xx. The `wrload` utility accomplishes this by dynamically allocating a "scratch" memory region within the loading CPU’s own 32-bit virtual address space, that will map to a physical address greater than 32-bits, to actually place the incoming ELF image above the 4 GB boundary.

Typically, the dynamic memory allocation is provided via an optional, architecture-independent pair of map and unmap routines. These are selected by copying `target/config/comps/vxWorks/10wrload.cdf` to the local BSP directory and defining the `WRLOAD_CPU_MEM_MAPFUNC` and `WRLOAD_CPU_MEM_UNMAP_FUNC` parameters’ DEFAULT field to `usrWrloadMap` and `usrWrloadUnmap` respectively. 4

For processors and targets requiring special handling for memory allocation, the map and unmap routines can be set by defining the aforementioned `WRLOAD_CPU_MEM_MAPFUNC` and `WRLOAD_CPU_MEM_UNMAP_FUNC` parameters to any special function the BSP author provides.

The map and unmap routines are registered with `wrload` in `usrWrloadInit( )` as part of `usrToolsInit( )` during the post-kernel initialization functions in `usrRoot( )` by the primary CPU.

**NOTE:** Adding the `INCLUDE_SHOW_ROUTINES` component to VxWorks, allows use of the `usrWrloadPeek( )` function to dump dynamically allocated memory regions from the shell. This is useful for verifying that the secondary ELF image was loaded to the expected location, or for inspecting the exception line of a secondary CPU from the primary CPU. For example:

```c
usrWrloadPeek (0, 0x10004300, 300, 1);
```

**Specifying a Load Bias**

The `wrload` utility provides a mechanism that allows a bias to be applied to the load memory addresses (LMA) of the ELF image being loaded. You can use this functionality to change the placement in memory of the segments contained in the ELF image. For example, when working with target hardware with 36-bit address support, the target image may need to be loaded above 32-bits, while still built with 32-bit addresses and written in ELF32 file format. You accomplish this using the load bias (the load bias is a 64-bit quantity).

You specify the load bias in the secondary CPU’s VxWorks Image Project or BSP by setting the parameter `WRLOAD_IMAGE_BUILD_PHYS_BIAS`. The `wrload load`
bias is embedded in the ELF image using the symbol passing convention described in *wrload Symbol Passing Facility*, p.173. In addition to being available for retrieval using the `wrloadSymValueGet()` routine, a `wrload` bias embedded with the `WRLOAD_IMAGE_BUILD_PHYS_BIAS` parameter is automatically used by the `wrload` utility.

The bias is a 64-bit value. Therefore, in an ELF32 image file it must be encoded using two separate 32-bit integers. The associated symbol names are `_wrload_bias_high_32` and `_wrload_bias_low_32`.

The following code is used to define a full 64-bit physical bias. The actual high and low bits of the bias are contained within the `WRLOAD_IMAGE_BUILD_PHYS_BIAS` macro in the following example:

```c
_WRS_ABSOLUTE_BEGIN(wrloadBiasSymbols)
  _WRS_ABSOLUTE(_wrload_bias_high_32, ((UINT64)WRLOAD_IMAGE_BUILD_PHYS_BIAS >> 32);
  _WRS_ABSOLUTE(_wrload_bias_low_32, WRLOAD_IMAGE_BUILD_PHYS_BIAS);
_WRS_ABSOLUTE_END
```

If you need to use the `wrloadSymValueGet()` routine to retrieve the bias value for passing to secondary core startup code, be aware the symbol name `_wrload_bias` should be used. It contains the full 64-bit bias computed from the high and low parts of the bias by `wrload`, and is forward compatible with ELF64 file support. In fact, the `wrload` utility will not reply to queries for `_wrload_bias[high,low]_32` symbols.

The `wrload` utility also provides a command line argument to specify a load bias. When making the call to `wrload` from the primary CPU, add `-loadbias` followed by the desired value of the physical load bias. The value may be up to 64 bits in length. The value will be implicitly modulus by `LOCAL_MEM_SIZE`. For more information on the `wrload` utility, see *VxWorks Kernel Programmer’s Guide: Booting VxWorks for AMP*.

**NOTE:** If you use the `-loadbias` option on the command line, the specified bias value overrides the value provided by the BSP.

**NOTE:** The `-loadbias` command-line option is only supported by MIPS or PPC BSPs in conjunction with the `-nostart` option. This is because the physical address of the secondary image must be known by the MMU handling software; a facility for replacing it (post-build) based on the command line `-loadbias` has not been implemented at this time. The physical offsets must be specified at build time by the `WRLOAD_IMAGE_BUILD_PHYS_BIAS` CDF parameter.

Further, MIPS images that support the large memory model (LMM) contain a memory section that is unmapped and cannot be biased. This section is used for vectors, kernel entry point, text, and data that must be executed in unmapped memory. The address of this section is given by the `LOCAL_UNMAPPED_BASE_ADRS` CDF parameter. See *VxWorks Architecture Supplement: MIPS* for information about LMM.

### Registering the `wrload` BSP-Specific Routines

A set of parameters is available (`WRLOAD_CPU_MAP_FUNC`, `WRLOAD_CPU_UNMAP_FUNC`, `WRLOAD_CPU_MEM_READ_FUNC`, and so on), each of which corresponds to a `wrload` hook routine. By defining the parameter to
a real function name, the function will become registered with the `wrload` utility and invoked by the `wrload` core when appropriate. Registration occurs in the routine `usrWrloadInit()`, which is automatically called from `usrToolsInit()`, within `usrRoot()`, whenever the vxWorks kernel includes the `wrload` utility.

All of the parameters are defined in `target/config/comps/vxWorks/10wrload.cdf`. To enable a hook routine, copy the file to the local BSP and edit it to set the `DEFAULT` field of the parameter definition for the desired hook routine to a real function name. Note, if enabling a hook routine, all the hook routines within a given set must be defined in total. There are four sets with hook routines grouped as follows:

- `WRLOAD_CPU_MEM_MAP_FUNC / WRLOAD_CPU_MEM_UNMAP_FUNC`
- `WRLOAD_CPU_MEM_READ_FUNC / WRLOAD_CPU_MEM_WRITE_FUNC / WRLOAD_CPU_MEM_FILL_FUNC`
- `WRLOAD_CPU_MEM_OPEN_FUNC / WRLOAD_CPU_MEM_CLOSE_FUNC`
- `WRLOAD_CPU_PREPARE_FUNC / WRLOAD_CPU_ENABLE_FUNC`

Use of the `WRLOAD_..._FUNC` routines allows generic architecture-dependent and architecture-independent implementations where possible, thus relieving the amount of customization required in the BSP.

If a hook routine is not registered, `wrload` will run with the legacy functionality. The `wrload` utility’s legacy functionality relies on the `sysAmpCpuPrep()` and `sysAmpCpuEnable()` routines, and assumes the secondary core memory is already mapped within the primary core memory space. However, this is not a best practice.

Best practice is to provide isolation between the memory for each core so that they are protected from stray accesses by other cores. This is usually done by making the memory of the secondary core(s) invisible or unmapped when seen from the primary core. In this situation, the `wrload` map/unmap or read/write/fill routines are used to provide access to the memory of the secondary core(s) during the secondary core loading process.

The memory access model to choose (either map/unmap or read/write/fill) typically depends on the underlying facility that the BSP hardware uses to manage memory.

The memory open and close routines are not mandatory and are available for performance reasons. Some BSPs, such as the VxWorks simulator BSP, call the `open()` routine on a shared memory file to access the secondary core’s memory. Using the memory open routine in this situation avoids opening and closing that file each time it is accessed.

Hook routines must retrieve the CPU index from the `WRLOAD_ID` that is passed to the routine. This is done using the `WRLOAD_ID_VALUE_GET()` macro on the `wrload` ID as follows:

```c
    cpuId = WRLOAD_ID_VALUE_GET(id);
```

Wind River strongly advises that BSP routines registered for use by the `wrload` utility check the type of the `wrload` ID using the `WRLOAD_ID_TYPE_GET` macro. This protects your BSP code from future changes in the `wrload` utility implementation. Currently, only the default ID type, which conveys a physical CPU index, is supported. An example of this check is as follows:

```c
    if (WRLOAD_ID_TYPE_GET(id) != WRLOAD_ID_DEFAULT_TYPE)
    {
```
In the above example, \textit{id} is the wrload ID passed to the routine and \texttt{WRLOAD_ID_DEFAULT_TYPE} is defined within \texttt{wrload.h}.

An erroneous return status from any of the registered memory access routines indicates that memory from a given core is unreachable and immediately terminates the secondary core load process.

The CPU prepare and enable routines, set with the parameters \texttt{WRLOAD_CPU_PREPARE_FUNC} and \texttt{WRLOAD_CPU_ENABLE_FUNC}, allow you to override \texttt{sysAmpCpuPrep()} and \texttt{sysAmpCpuEnable()}. This avoids code duplication if, for instance, the routines are not BSP-dependent but architecture-dependent. The \texttt{arg} argument of the initialization routine is currently unused and set to \texttt{NULL}. Again, if the hook routines are not registered, \texttt{wrload} falls back to making direct calls to the \texttt{sysAmpCpuXXX()} routines.

For more information on \texttt{wrload} support for specific target hardware, see the reference entry (\texttt{target.ref} file) for your BSP.

### 7.12 Allocating Address Space

One of the challenges in setting up a VxWorks AMP environment is partitioning hardware resources among the different VxWorks run-time images. This section discusses the partitioning of system RAM between VxWorks AMP images.

#### 7.12.1 Overview of Uniprocessor Address Space Allocation for VxWorks

When a single VxWorks image runs, it typically claims all available memory for its use. This block of memory is configured into the run-time image by setting the following three VxWorks configuration parameters:

- **LOCAL_MEM_LOCAL_ADRS** (system memory start address)
  The address of the lowest addressable byte of memory
- **RAM_LOW_ADRS** (Run-time kernel load address)
  The architecture-dependent address of the text segment.
- **LOCAL_MEM_SIZE** (system memory size)
  The size, in bytes, of local memory.

The address space from **LOCAL_MEM_LOCAL_ADRS** to the sum of **LOCAL_MEM_LOCAL_ADRS** plus **LOCAL_MEM_SIZE** is subdivided into the regions shown in Figure 7-3.

Figure 7-3 is a generalized diagram that illustrates a typical VxWorks memory layout. For details about the memory layout for a particular architecture, see the \textit{VxWorks Architecture Supplement}. In Figure 7-3, note that:
The VxWorks image uses all of the memory from \( \text{LOCAL\_MEM\_LOCAL\_ADRS} \) to the sum of \( \text{LOCAL\_MEM\_LOCAL\_ADRS} \) plus \( \text{LOCAL\_MEM\_SIZE} \).

- All memory allocations in the VxWorks local memory range are computed relative to \( \text{LOCAL\_MEM\_LOCAL\_ADRS} \). It is unsafe to assume that \( \text{LOCAL\_MEM\_LOCAL\_ADRS} \) has a predictable value (such as 0).
- There is an architecture-dependent gap between \( \text{LOCAL\_MEM\_LOCAL\_ADRS} \) and \( \text{RAM\_LOW\_ADRS} \).
- \( \text{sysMemTop()} \) and \( \text{sysPhysMemTop()} \) are routines that need to be implemented in an AMP BSP (see \( \text{sysMemTop()} \), p.159).

\( \text{sysPhysMemTop()} \) returns a pointer to the top of physical memory, \( \text{sysMemTop()} \) returns a pointer to the memory location at the top of the memory area allocated to the heap, system stacks, and the RTP memory pool.

### 7.12.2 Allocating Address Space for AMP

In VxWorks AMP, the address map described in Figure 7-3 needs to be replicated for each run-time image that is loaded into the system. In addition, if you are going to use MIPC for communication between CPUs, you need to configure separate blocks of address space for MIPC shared-memory usage (see 7.14 Adding Support for MIPC, p.185 and Figure 7-4).

The following section covers allocation of address space for VxWorks in individual CPUs. For information on allocating address space for MIPC, see 7.14.1 Allocation of Address Space for MIPC Shared Memory, p.185.
Allocating Address Space for VxWorks Run-time Images

For VxWorks AMP, each run-time image in a processor needs to have its own unique value for the following memory parameters:

- LOCAL_MEM_LOCAL_ADRS
- RAM_LOW_ADRS
- LOCAL_MEM_SIZE

You declare these parameters and assign them default values in the 20bsp.cdf file for an AMP BSP. See, for example, the 20bsp.cdf file for ads8572_AMP or cav_cn3xxx_mipsi64r2sf. The path to the file is:

`installDir/vxworks-6.x/target/config/bspName/20bsp.cdf`

The memory allocation parameters for a CPU are always assigned to a component that has the following format:

```
INCLUDE_AMP_CPU_nn
```

where `nn` is a two-digit CPU number. For example, memory allocation parameters for physical CPU index 1 are assigned to the component `INCLUDE_AMP_CPU_01`. In VxWorks 6.8 Update Pack 1 and beyond, `INCLUDE_AMP_CPU_0n` is not tied to a specific physical CPU. Instead, it defines a range of addresses the image will execute in and some default components the image will include. The physical CPU the image executes on is determined by the `wrload` utility. For more information, see the VxWorks Kernel Programmer's Guide: Overview of VxWorks AMP.

Before you can assign these values, you need to decide how much system RAM to allocate to each image. For an AMP system without MIPC, you can simply divide system RAM into equal parts. When MIPC is included, you need to allocate space for it and then subdivide the remaining memory. RAM does not need to be divided equally; you can allocate more RAM to some CPUs than others.

In allocating address space to individual CPUs, it is important to make sure that none of the CPUs have overlapping address spaces, because this may prevent VxWorks from loading or cause it to fail once it does. In addition, to avoid unused RAM, make sure that there are no gaps in allocations from one CPU to another.

**NOTE:** The `wrload` utility must have sufficient system memory on the primary CPU to read (copy) the entire secondary image, before moving it to its own space. Also, the bootstrap CPU often has more initialization and optional components than the secondary CPUs. Therefore, the primary CPU typically needs a minimum of 4 to 8 MB more memory than a secondary CPU.

**Special Syntax for Default Values in AMP CDF Files**

In a CDF file for a uniprocessor BSP, enter a single default value for a parameter in the `DEFAULT` field of a parameter block:

```
Parameter LOCAL_MEM_LOCAL_ADRS {
    NAME Runtime kernel lower memory address
    DEFAULT 0x00000000
}
```

For AMP, in order to specify separate default values for each configuration, the syntax for specifying a default value has been extended, as follows:

```
<contextA>::(<contextA_default_value) \n<contextB>::(<contextB_default_value) \n```
... (contextN)::(contextN_default_value) \n  default_value

• All lines are optional:
  • You can specify all default values in terms of contexts, without a context-free default value
  • You can specify contexts for default values followed by a context-free default value
  • As in the past, you can specify a context-free default value on its own.

• If you enter a context-free default as the last line of the DEFAULT field, it applies to all contexts other than those previously specified.

The following parameter statement illustrates the new syntax:

Parameter RAM_HIGH_ADRS {
  NAME Example parameter
  DEFAULT (CONTEXT_A)::(5) \n  (CONTEXT_B)::(10) \n  0
}

The parameter statement can be interpreted as follows:

• If CONTEXT_A is defined, set the default for SOME_PARAM to 5.
• If CONTEXT_B is defined, set the default for SOME_PARAM to 10.
• In all other cases, set the default for SOME_PARAM to 0.

Setting the LOCAL_MEM_LOCAL_ADRS Parameter

If a system has 64 megabytes of RAM equally divided across four CPUs, each CPU will have 16 megabytes (0x01000000) of address space. The following example assigns default values to LOCAL_MEM_LOCAL_ADRS for this configuration:

Parameter LOCAL_MEM_LOCAL_ADRS {
  NAME Runtime kernel local memory address
  DEFAULT (INCLUDE_AMP_CPU_03)::(0x03000000) \n  (INCLUDE_AMP_CPU_02)::(0x02000000) \n  (INCLUDE_AMP_CPU_01)::(0x01000000) \n  (INCLUDE_AMP_CPU_00)::(0x00000000) \n  0
}

Setting the RAM_LOW_ADRS Parameter

There is always an architecture-dependent offset between LOCAL_MEM_LOCAL_ADRS and RAM_LOW_ADRS. Given the system in the preceding section and an offset of 128 KB, you would assign default values to RAM_LOW_ADRS as follows:

Parameter RAM_LOW_ADRS {
  NAME Runtime kernel load address
  DEFAULT (INCLUDE_AMP_CPU_03)::(0x03020000) \n  (INCLUDE_AMP_CPU_02)::(0x02020000) \n  (INCLUDE_AMP_CPU_01)::(0x01020000) \n  (INCLUDE_AMP_CPU_00)::(0x00020000) \n  0
}
7 Extending a BSP to an AMP System

7.13 Assigning Devices

Setting the LOCAL_MEM_SIZE Parameter

When the same amount of RAM is allocated to each CPU, it is not necessary to use the special conditional syntax for default values of the preceding examples. The following example sets LOCAL_MEM_SIZE to 16 megabytes:

```plaintext
Parameter LOCAL_MEM_SIZE {
    NAME local memory size
    DEFAULT 0x01000000
}
```

It is possible to assign individual CPUs differing amounts of RAM, as in the following example, which assigns 32 megabytes of RAM to CPU0, and 16 megabytes of RAM to all other CPUs:

```plaintext
Parameter LOCAL_MEM_SIZE {
    NAME local memory size
    DEFAULT (INCLUDE_AMP_CPU_00)::(0x02000000) \ 0x01000000
}
```

Summary

Each VxWorks AMP image needs its own memory space, and the memory spaces allocated to separate images may not overlap. You must ensure this through the settings you assign in each image to the following parameters:

- LOCAL_MEM_LOCAL_ADRS
- RAM_LOW_ADRS
- LOCAL_MEM_LOCAL_SIZE

7.13 Assigning Devices

In a BSP, the hardware devices available to a VxWorks image, such as serial ports, Ethernet ports, timers, interrupt controllers, and so on, are listed in the hcfDeviceList[] array in the BSP’s hwconf.c file. During system initialization, VxBus checks to see if BSP device filtering is in effect (see The sysBspDevFilter() Routine for Run-Time Device Assignment, p.182). If device filtering is not in effect, VxBus iterates through the devices listed in hcfDeviceList[] and attempts to pair each device with a corresponding device driver (for information on VxBus and system initialization, see VxBus Device Driver Developer’s Guide: Device Driver Fundamentals). If a driver is found that supports a listed device, the device and driver are paired together to form a device instance that can be used within the system.

For VxWorks AMP, the devices listed in hcfDeviceList[] need to be specifically assigned to individual VxWorks images. For example, if there are two serial ports, the system designer needs to determine which port to allocate to which CPU and modify the BSP accordingly. For the current release, it is recommended that CPU0 always have a serial port so that it can be used as a console.
Assigning Devices to Individual CPUs

When a single BSP is used in creating all of the VxWorks images in an AMP system, as assumed here, you can assign hardware devices to images and ensure that no two images attempt to claim the same device in the following ways:

- **Run-Time Configuration**
  - Implement the `sysBspDevFilter()` BSP driver method to assign devices to CPUs at run time (see *The sysBspDevFilter() Routine for Run-Time Device Assignment*, p.182).

- **Build-Time Configuration**
  - Remove support for a device driver from a VxWorks Image Project (VIP) (see *Removing a Device Driver from a VxWorks Image Project*, p.184).
  - Use `ifdef` statements in `hwconf.c` to control access to devices (see *Using ifdef Statements to Control Access to Devices*, p.184).

Devices Shared by Multiple CPUs

In a VxWorks AMP configuration, some hardware devices need to be shared by multiple CPUs. For example, on most AMP systems, all CPUs need to share a single interrupt controller. In such cases, the driver for the device must be able to route interrupts to the appropriate CPU. For information on implementing such drivers, see *C. Writing Device Drivers for VxWorks AMP*.

7.13.1 Run-time Configuration

Wind River recommends that you use the run-time configuration method described in this section for device allocation. However, build-time configuration is still supported and is discussed in 7.13.2 Build-Time Configuration, p.183.

The `sysBspDevFilter()` Routine for Run-Time Device Assignment

You can allocate devices at run time through the driver method `sysBspDevFilter()`.

To do this, you need to:

1. Publish `sysBspDevFilter()` to VxBus (see *Publishing the sysBspDevFilter() Driver Method*, p.183).
   
   During system initialization, VxBus checks to see if `sysBspDevFilter()` is published. If it is, VxBus first makes device allocations based on the device assignments in `sysBspDevFilter()`, then it makes any remaining allocations based on the devices listed in `hwconf.c`.

2. Implement `sysBspDevFilter()` (see *Implementing a sysBspDevFilter() Routine*, p.183).

---

5. Serial lines/UARTs are an exception to this; they cannot be dynamically assigned at run time.
Publishing the \texttt{sysBspDevFilter()} Driver Method

To publish \texttt{sysBspDevFilter()}, enter lines in the \texttt{sysLib.c} file for your BSP, as follows:

1. At the top of the file, enter the following lines:

   \begin{verbatim}
   LOCAL STATUS sysBspDevFilter (struct vxbDev *);
   METHOD_DECL(sysBspDevFilter);
   IMPORT device_method_t * pSysPlbMethods;
   LOCAL struct vxbDeviceMethod plbMethods[] =
       {        
           DEVMETHOD(sysBspDevFilter, sysBspDevFilter),
           { 0, 0 } 
       };
   \end{verbatim}

2. Within the \texttt{sysHwInit()} routine, find \texttt{hardWareInterFaceInit()} and set the line before it to \texttt{pSysPlbMethods = plbMethods}, as in the following example:

   \begin{verbatim}
   pSysPlbMethods = plbMethods; /* advertise device filter callback */
   hardWareInterFaceInit();
   \end{verbatim}

3. Implement \texttt{sysBspDevFilter()} (see \textit{Implementing a sysBspDevFilter()} Routine, p.183)

Implementing a \texttt{sysBspDevFilter()} Routine

You can implement the logic necessary for allocating devices to CPUs in \texttt{sysBspDevFilter()}. The following example illustrates how to implement \texttt{sysBspDevFilter()} to allocate one Ethernet port to each CPU in an AMP system:

\begin{verbatim}
LOCAL STATUS sysBspDevFilter
    (struct vxbDev *pDev)
    
    if (!strcmp(pDev->pName, "ethernet"))
        
          /* allocate one Ethernet device per CPU */
          if (pDev->unitNumber == vxCpuIdGet())
            return OK;
          else
            return ERROR;
          
        /* for all non-Ethernet devices, allow VxBus to use them */
        
        return OK;
    }
\end{verbatim}

7.13.2 Build-Time Configuration

On processors other than MIPS, Wind River recommends that you use the run-time configuration method for device allocation when possible (see 7.13.1 Run-time Configuration, p.182). However, a build-time configuration method is also supported and is described in this section. (For MIPS processors, the build-time configuration described here is preferred.)
Removing a Device Driver from a VxWorks Image Project

In a VxWorks Image Project (VIP), device drivers are build components that can be selectively added to or removed from a project. To prevent a device from being included in an image project, you can remove the component for its device driver from the project. For example, the component for the NS16550 serial driver is DRV_SIO_NS16550. To use the vxprj command-line utility to remove the NS16550 serial driver, you enter:

```
-> vxprj component remove DRV_SIO_NS16550
```

Removing the device driver for a device is a straightforward way to prevent the device from being used in an individual VxWorks AMP configuration. However, this approach is not feasible if:

- You have multiple devices that use the same driver, and you want to use the driver to control a subset of the devices, but not all of them.
- You want to avoid creating a custom VxWorks configuration for each of the CPUs in your system.

In these cases, you need to use one of the following approaches:

- Control access to devices using `ifdef` statements (see Using `ifdef` Statements to Control Access to Devices, p.184)
- Allocate devices at run time using `sysBspDevFilter()`

This is the recommended approach (see 7.13.1 Run-time Configuration, p.182).

Using `ifdef` Statements to Control Access to Devices

You can use `ifdef` statements to control the assignment of devices to VxWorks images.

When you configure a VxWorks Image Project for AMP, the project must always apply to a specific CPU. You designate the CPU by including the build component `INCLUDED_AMP_CPU_"mn"` (AMP kernel configured for core mn) in the project, where mn is a two digit number for the CPU. For example, to build a project for CPU1, you need to include the build component `INCLUDED_AMP_CPU_01` (AMP kernel configured for core 01). When you build the project, the build process automatically defines the C preprocessor symbol `INCLUDED_AMP_CPU_mn` for your project’s CPU.

In the `hwconf.c` file for a BSP, you can control the assignment of devices to CPUs by using `ifdef` statements with `INCLUDED_AMP_CPU_mn` definitions. For example, suppose four Ethernet devices are entered in a `hwconf.c` file, as follows:

```
{ "mottsec", 0, VXB_BUSID_PLB, 0, tsecVxbEnd0Num, tsecVxbEnd0Resources },
{ "mottsec", 1, VXB_BUSID_PLB, 0, tsecVxbEnd1Num, tsecVxbEnd1Resources },
{ "mottsec", 2, VXB_BUSID_PLB, 0, tsecVxbEnd2Num, tsecVxbEnd2Resources },
{ "mottsec", 3, VXB_BUSID_PLB, 0, tsecVxbEnd3Num, tsecVxbEnd3Resources },
```

To assign a different Ethernet device to each of four CPUs, you can use the following `ifdef` logic:

```
#if defined(INCLUDED_AMP_CPU_00)
{ "mottsec", 0, VXB_BUSID_PLB, 0, tsecVxbEnd0Num, tsecVxbEnd0Resources },
#endif
```

```c
#elif defined(INCLUDED_AMP_CPU_01)
{ "mottsec", 1, VXB_BUSID_PLB, 0, tsecVxbEnd1Num, tsecVxbEnd1Resources },
#endif
```

```c
#elif defined(INCLUDED_AMP_CPU_02)
{ "mottsec", 2, VXB_BUSID_PLB, 0, tsecVxbEnd2Num, tsecVxbEnd2Resources },
#endif
```

```c
#elif defined(INCLUDED_AMP_CPU_03)
{ "mottsec", 3, VXB_BUSID_PLB, 0, tsecVxbEnd3Num, tsecVxbEnd3Resources },
#endif
```

This is the recommended approach (see 7.13.1 Run-time Configuration, p.182).
7.14 Adding Support for MIPC

Wind River MIPC provides communication between CPUs in a multicore environment. It uses a dedicated area of shared memory to implement one or more virtual buses, each of which can be used for messaging (see the Wind River MIPC Programmer’s Guide). Each virtual bus can use interrupts to detect bus traffic. Alternatively, the bus can be polled for activity. All MIPC buses that are interrupt-driven share the same interrupt which may also be shared by other VxWorks services.

To add support for MIPC to an AMP BSP, you need to:

- Set your address map to reserve memory for the MIPC shared memory region (see 7.14.1 Allocation of Address Space for MIPC Shared Memory, p. 185).
- Configure the VxWorks components that utilize the address map (see 7.14.2 Configuring MIPC Build Components in the 22comp_mipc.cdf File, p. 187).
- Configure the VxWorks components that specify the interrupt used by MIPC buses (see Adding Parameters for the Interrupt Used by MIPC Buses, p. 187).

In addition to configuring these items, you may also need to change other MIPC configuration settings from their default values, based on your intended use of MIPC. For example, you may need to increase the number of MIPC buses (default is 1), the number of MIPC nodes per bus (default is 2), the number of message buffers per node per bus (default is 64), or a combination of these. For more information on the various configuration settings provided by MIPC, see the Wind River MIPC Programmer’s Guide.

7.14.1 Allocation of Address Space for MIPC Shared Memory

When you include MIPC in AMP, you need to configure address space for MIPC in shared-memory, as illustrated in Figure 7-4.
The starting address of the MIPC shared memory region should be aligned to a page boundary so that it can be safely mapped by your MMU when running in mapped mode.

The size of the shared memory region required to run MIPC is influenced by a number of factors, including:

- the number of MIPC buses in the system
- the number of MIPC nodes on each bus
- the number of MIPC ports each node has on a bus
- the number of transmit buffers each port uses
- the total number of MIPC sockets each node has

Of these, the most significant factor is typically the memory required for transmit buffers. A rough order-of-magnitude estimate for the amount of shared memory required in a MIPC system can be found by determining the total number of transmit buffers required by all ports on all nodes for a given bus, multiplying this by the buffer size used by the bus, and considering this to be the memory requirements for that bus. Repeat this calculation for each bus in the system and then use the sum of the results to estimate the total amount of shared memory required for MIPC.

You should initially overestimate the amount of memory needed by MIPC and then use the mipcShow() routine on a running system to determine how much unused memory remains, after which you can adjust the size of the MIPC shared memory region.
7.14 Adding Support for MIPC

7.14.2 Configuring MIPC Build Components in the 22comp_mipc.cdf File

The 22comp_mipc.cdf BSP file contains build components and parameters for configuring MIPC. The directory location of the file is:

\[\text{installDir/vxworks-6.x/target/config/bsp_name/22comp_mipc.cdf}\]

The following sections apply to entries in this file.

Enabling MIPC

By default, MIPC build components are not visible within Workbench because the majority of VxWorks BSPs do not support MIPC. To make MIPC components visible, add the following lines to 22comp_mipc.cdf:

```
Folder FOLDER_MIPC {
    _CHILDREN FOLDER_MULTIOS
}
Component INCLUDE_MIPC_UNSUPPORTED {
    _CHILDREN FOLDER_NOT_VISIBLE
}
```

Adding Parameters for the Address and Size of the MIPC Shared-Memory Region

After the INCLUDE_MIPC_UNSUPPORTED component in 22comp_mipc.cdf, enter the parameters that specify the address and size of the MIPC shared-memory region. The following example shows how to do this:

```
Parameter MIPC_SM_SYSTEM_POOL_BASE {
    DEFAULT 0x84000000
}
Parameter MIPC_SM_SYSTEM_POOL_SIZE {
    DEFAULT 0x200000
}
```

For determining the default values to enter, see 7.14.1 Allocation of Address Space for MIPC Shared Memory, p.185.

Adding Parameters for the Interrupt Used by MIPC Buses

If you want to have one or more interrupt-driven MIPC buses in your system, enter the parameter that specifies the interrupt to use after the

INCLUDE_MIPC_UNSUPPORTED component in 22comp_mipc.cdf. The following example shows how to do this:

```
Parameter MIPC_SM_NODE_IRQ {
    DEFAULT 0
}
```
8.1 Introduction

VxWorks can be configured using a small-footprint profile. This configuration is designed for systems in which minimal kernel functionality is necessary and a small memory footprint is required. You must make modifications to your BSP to support this configuration. This chapter describes the BSP modifications that are required to support a small-footprint configuration. For more information on using this configuration in your VxWorks system, see the VxWorks Kernel Programmer’s Guide: VxWorks Configuration.

NOTE: The small-footprint VxWorks configuration is not supported for all processor architectures. For information on currently supported features for your target architecture, see the VxWorks Architecture Supplement.

8.1.1 Prerequisites

The start point for your BSP development should be an existing BSP that supports the small-footprint VxWorks configuration (if one is available for your target processor) or a working uniprocessor BSP. Wind River does not recommend writing a small-footprint enabled BSP from scratch.

If you are starting with a uniprocessor BSP (not small-footprint enabled), you can use any standard BSP provided by Wind River or a third-party, or you can use a custom BSP that you have created. In either case, you must be sure that the small-footprint configuration is supported by your architecture and that the BSP is working and validated prior to beginning you small-footprint configuration work.
8.1.2 Workflow

If you are making minor modifications to an existing BSP that already includes support for the small-footprint VxWorks configuration, you can study the source code for the BSP and make modifications as needed.

If you are modifying a standard uniprocessor BSP to support the small-footprint configuration, you need to consider the following tasks:

- **Design and Plan**—Your BSP can be designed to support both standard and small-footprint VxWorks configurations or it can support the small-footprint configuration alone. When developing a dual purpose BSP, you must develop a `common.vxconfig` file to control how the BSP is built (with or without support for the small footprint configuration).

- **Modify Standard BSP Files**—This includes updating `hwconf.c` to remove unneeded drivers and adjusting ram addresses in `Makefile`, `config.h`, `20bsp.cdf`, and `sysLib.c`.

- **Implement Cache and MMU Support**—The small-footprint configuration does not provide cache and MMU support in the kernel. If you require this support, you may need to make changes to your BSP to provide basic cache and MMU functionality without pulling in the standard cache and MMU-related libraries.

- **Debug the BSP**—Small-footprint configuration BSPs do not show any output at boot time. You may want to implement additional code to help with initial debugging and boot up. This may increase the size of your small-footprint VxWorks image.

- **Test and Validate**—The VxWorks BSP validation test suite (BSP VTS) provides support for testing small-footprint configuration BSPs.

8.2 Planning Your BSP

The first step in planning your BSP is to choose an existing BSP to use as a start point. Wind River recommends that you always start with an existing validated uniprocessor BSP. In some cases, you may be able to use an existing BSP that already includes support for the small-footprint VxWorks configuration. In this case, modifications are generally limited to typical BSP adaptations such as driver support or changes to the memory map.

In many cases, you will need to choose a standard uniprocessor (UP) BSP as your start point because there is no small-footprint BSP available for your target architecture. In this situation, you may still find it useful to have a small-footprint BSP available for reference.

Once you have chosen your reference BSP, you must decide if the new BSP will support both small-footprint and standard VxWorks configurations or if it will support the small-footprint configuration only. The small-footprint VxWorks configuration is enabled and built using the small-footprint profile in a VxWorks Source Build (VSB). For a dual purpose BSP, you create a `common.vxconfig` file that is used to define certain macros that indicate if the BSP should be built with
support for the small-footprint configuration. The common.vxconfig file is not required for BSPs that only support the small-footprint configuration. However, you must still create a VSB project to enable the small-footprint configuration.

### 8.3 Modifying Your BSP

This section generally assumes that you are starting with a working (validated) UP BSP as a start point. This section provides a specific example BSP, as well as guidelines for general development. Keep in mind that the specific development steps required for your project will vary depending on your target processor and how closely matched your design is to your reference BSP.

The examples provided are taken from the ti_omap137evm BSP for the ARM architecture. This BSP is a dual purpose BSP that includes a default configuration with full standard BSP capabilities as well as a small-footprint configuration.

**Step 1: Create the common.vxconfig File**

**NOTE:** The common.vxconfig file is only required if you are creating a dual purpose BSP. If you are creating a BSP that will be used for the small-footprint configuration only, you can skip this step.

The common.vxconfig file is located in the BSP directory:

```
installDir/vxworks-6.x/target/config/bspName
```

This file includes a macro that is defined when the BSP is being built to support a small-footprint configuration. The macro is then used to enable or disable code throughout the BSP. The macro takes the form:

```
_WRS_CONFIG_bspName_VSB_BUILD
```

where `bspName` is the name of your BSP. For example:

```
_WRS_CONFIG_TI_OMAPL137EVM_VSB_BUILD
```

You can create your common vxconfig file by copying an example from an existing small-footprint BSP (provided with your installation) and renaming the macro for your BSP.

The following is a code snippet taken from the ti_omap137evm BSP:

```
... config TI_OMAPL137EVM_VSB_BUILD
bool
default y if PROFILE_SMALL_FOOTPRINT
help
This feature configures bsp to use VSB configuration options.
```

**Step 2: Modify the hwconf.c and config.h Files.**

The hwconf.c and config.h files are located in:

```
installDir/vxworks-6.x/target/config/bspName
```

In some cases, you will need to modify these files to remove unneeded drivers when the BSP is built for the small-footprint configuration. The only drivers
required for the small-footprint configuration are an interrupt controller and a
timer driver. You may also want to include a serial driver for debugging (optional).
Other drivers, such as network drivers, should be removed or conditionalized
with the necessary define statements. In particular, you must be sure to properly
set any VSB build configuration options.

The following example is taken from the config.h file for the ti_omap137evm BSP.
This examples shows the VSB build configuration settings specific to the
small-footprint configuration for this BSP.

```c
/*
 * VSB build configuration options
 * If PROFILE_SMALL_FOOTPRINT, the following four features would not be defined.
 */

#ifndef _WRS_CONFIG_TI_OMAPL137EVM_VSB_BUILD
  # define _WRS_CONFIG_TI_OMAPL137EVM_SPIFLASH_SUPPORT
  # define _WRS_CONFIG_TI_OMAPL137EVM_I2C_SUPPORT
  # define _WRS_CONFIG_TI_OMAPL137EVM_I2C_EEPROM_SUPPORT
#else
  # define _WRS_CONFIG_TI_OMAPL137EVM_STATIC_MMU
#endif /* _WRS_CONFIG_TI_OMAPL137EVM_VSB_BUILD */

#ifdef _WRS_CONFIG_TI_OMAPL137EVM_SPIFLASH_SUPPORT
  # define INCLUDE_OMAPL137_SPIFLASH
#endif /* _WRS_CONFIG_TI_OMAPL137EVM_SPIFLASH_SUPPORT */

#undef INCLUDE_OMAPL137_RTC

#ifdef _WRS_CONFIG_TI_OMAPL137EVM_I2C_SUPPORT
  # define INCLUDE_OMAPL137_I2C
#endif /* _WRS_CONFIG_TI_OMAPL137EVM_I2C_SUPPORT */

Step 3: Adjust the RAM Addresses as Needed

You may need to make modifications to Makefile, config.h, and 20bsp.cdf to
adjust RAM_LOW_ADRS for a narrower memory address range. sysLib.c must
also be modified to reflect the proper settings for your hardware.

8.3.1 Implementing Cache and MMU Support

You must give careful consideration to cache and MMU support when developing
for a small-footprint VxWorks configuration. In a standard VxWorks
configuration, the libraries associated with cache and MMU (vmLib and mmuLib)
bring in a considerable amount of code and therefore should not be enabled as-is
in a small-footprint configuration. However, for most hardware platforms, you
must enable some level of cache support in order to get reasonable performance.
To provide this functionality without substantially increasing the memory
footprint, you must enable cache locally in the BSP.

MMU also increases the memory footprint but does not have the performance
ramifications associated with cache support. For this reason, Wind River
recommends that MMU not be enabled in any capacity. However, on some
architectures (such as ARM), cache cannot be enabled without some level of MMU
support. In these cases, you must enable the MMU locally in the BSP as well.

The following section discusses how this support is implemented in the
ti_omap137evm BSP.
Example: ti_omap137evm BSP

This section describes the cache and MMU implementation used in the ti_omap137evm BSP.

Setting Up the Static MMU Page Tables

In order to insure a reasonable amount of performance, cache must be enabled. On the ARM architecture the MMU must be enabled in order to enable data cache. The full vmLib and mmuLib pull in a fairly large amount of code. In order to bypass this system and architecture code, a minimal MMU (and therefore cache) configuration is implemented locally in the BSP.

For ARM, this solution requires the following:

- INCLUDE_CACHE_SUPPORT is not defined so that cacheArmxxxxLibInstall is not called from sysHwInit0().
- INCLUDE_MMU is not defined so that mmuArmxxxxLibInstall is not called from sysHwInit0().
- sysPhysMemDesc[ ] is protected with #ifdef INCLUDE_MMU so that the structure is not part of the VxWorks image.
- sysToMonitor() does not call VM_ENABLE. Instead, this routine calls mmuADisable().

sysMmuInit()

In the ti_omap137evm BSP, the sysMmuInit() routine is implemented and called from sysHwInit0(). This routine sets up the static MMU page table. The example BSP uses section page table entries to map all required addresses. You could also choose to implement coarse second-level entries for faster operation (but a larger footprint).

NOTE: This routine is specific to 32-bit BSPs.

```c
LOCAL void sysMmuInit(void)
{
    int i;
    unsigned int * mmuPageTable = (unsigned int *)(LOCAL_MEM_END_ADRS - 
        USER_RESERVED_MEM);

    /* Interrupt vector is cachable */
    mmuPageTable[0] = LOCAL_MEM_BUS_ADRS | (MMU_SECTION_DES_DEFAULT_VALUE | 
        MMU_SECTION_DES_CACHE_MODE);

    /* RAM is cachable, others is uncachable */
    for (i = 1; i < MMU_PAGE_ENTRIES; i++)
    {
        mmuPageTable[i] = ( i << 20) | (MMU_SECTION_DES_DEFAULT_VALUE);

        if (((LOCAL_MEM_LOCAL_ADRS >> 20) <= i) &&
            (i < (LOCAL_MEM_END_ADRS >> 20)))
            mmuPageTable[i] |= (MMU_SECTION_DES_CACHE_MODE);
    }
}
```
8.4 Debugging Your BSP

When working with a small-footprint configuration BSP, you should note that no output is generated when the BSP boots and depending on your configuration, WDB is not guaranteed to work. One way to prove that your system is booting and running correctly is to add \texttt{kprintf()} statements. \texttt{kprintf()} statements can be inserted anywhere after \texttt{sysHwInit()} to show the latter stages of the boot process.

Another option would be to use an in-circuit emulator (ICE). For example, the OMAP-L137 EVM board from Texas Instruments includes an on-board ICE. You can debug this board by connecting a USB cable to the J201 connection and running Code Composer Studio on the host. Once the system is up and running, you can send a \texttt{stop} command to the on-board ICE which should stop in the idle loop thus indicating a functional system.

8.5 Testing and Validating Your BSP

Note that you should always begin your small-footprint BSP development by starting with a tested and validated uniprocessor BSP. This gives you a baseline test to start with and should ease the overall development and testing process.

Wind River recommends that you use the VxWorks BSP Validation Test Suite (BSP VTS) to test and validate your small-footprint configuration BSP. To do this, you must build and run your tests with the small-footprint test harness driver. This section provides instructions on how to build and run tests with the small-footprint test harness driver and also provides some general usage caveats for BSP testing using a small-footprint configuration. For more information and general instructions for working with the BSP VTS, see the \textit{VxWorks BSP Validation Test Suite User's Guide}.

8.5.1 Enabling the Small-Footprint Test in a VSB

To enable small-footprint BSP testing in the BSP VTS, you must enable the following options in your VSB (in addition to the \texttt{PROFILE_SMALL_FOOTPRINT} option):

\texttt{COMPONENT_VXTESTV2_PSF}

Enables the building of BSP VTS test cases into the library.
SFOOTPRINT_VXTESTV2
Enables specialized vxTestV2 testing that is designed to validate a small-footprint VxWorks configuration. SFOOTPRINT_VXTESTV2 configures the test cases to avoid references to routines that are not available in a small-footprint image. This option is only available when PROFILE_SMALL_FOOTPRINT is enabled.

For more information on enabling and working with VSBs and the small-footprint VxWorks configuration, see the VxWorks Kernel Programmer’s Guide: VxWorks Configuration.

8.5.2 Building and Running Tests with the Small-Footprint Test Harness Driver

To build and run the test suite with the small-footprint driver enabled, execute the suiteRun command with the -sfDriver option switch. This should be executed from the VxWorks Development Shell (or a command shell on the host with a properly set environment) as follows:

```bash
% vxtest suiteRun -sfDriver otherRequiredOptions
```

This -sfDriver switch selects the small-footprint driver. The small footprint driver resolves all vxTestV2 routines used by the test cases. The driver requires no externals except a character output routine and VX_TASK_INSTANTIATE() (see installDir/vxworks-6.x/target/h/taskLib.h). The entry point for the character output routine defaults to sfPutChar() (which in turn calls kprintf() to print a single character).

This may be overridden with the use of the following CDF parameters:

- VXTESTV2_SF_DRIVER_PUTCHAR
- VXTESTV2_SF_DRIVER_PUTCHAR_IS_RAW

To override the character output function used by the small footprint driver:

1. Add a CDL to the BSP CDF file (20bsp.cdf). For example, if your BSP defines a low-level character output routine ns16550PutChar(), add the following:

   ```
   Parameter VXTESTV2_SF_DRIVER_PUTCHAR {
     DEFAULT ns16550PutChar
   }
   ```

2. Add a prototype declaration of your character output function to config.h. For example:

   ```
   #ifndef _ASMLANGUAGE
   IMPORT int ns16550PutChar (int c);
   #endif
   ```

3. If your character output routine enhances the output (adds carriage returns to line feeds), then there is nothing more to do. If it is a raw character output routine, you need to also add the following CDL to 20bsp.cdf:

   ```
   Parameter VXTESTV2_SF_DRIVER_PUTCHAR_IS_RAW {
     DEFAULT 1
   }
   ```

**NOTE:** The return value of the routine is ignored. It is maintained in the function pointer prototype in order to be compatible with sfPutChar().

For example:

```c
#ifndef _ASMLANGUAGE
IMPORT int ns16550PutChar (int c);
#endif
```
Examples

The examples provided below show sample commands for running certain test suite scenarios for small-footprint BSPs. The output from these commands is similar to the output from similar test runs using standard BSPs. For output examples and more information on working with the BSP VTS in general, see the VxWorks BSP Validation Test Suite User’s Guide.

Example 8-1  Run the Pre-Defined Test Suite for Small-Footprint Configuration Images

To run the pre-defined test suite for small-footprint configuration images (installDir/vxworks-6.x/vxtest/src/config/smallfootprint.prm), do the following:

1. Build a VSB for small footprint in:

   $ cd $WIND_BASE/target/vxworks-6.x/target/vsb_ bspName

   Do the following:

   $ vxprj vsb create -bsp bspName
   $ cd vsb_ bspName
   $ make

2. Create a board configuration (board.prm) file for your board:

   $ vxtest boardConfig -template boardName

   Then, edit the resulting file to add specifics about the board.

3. Run the small footprint test suite with your BSP:

   $ vxtest suiteRun -suite smallfootprint -vsb $WIND_BASE/target/vsb_ bspName -profile PROFILE_SMALL_FOOTPRINT -sfDriver -libBuild -bspToolBoard bspName toolchain boardName

Example 8-2  Running a Single Test Module

This example assumes that you want to run a single test module—in this case, installDir/vxworks-6.x/vxtest/src/tests/bsp/sysClock/tmSysClock.c—and you have built a VSB for small footprint in installDir/vxworks-6.x/target/vsb_ bspName.

To run this scenario, enter the following command:

$ vxtest suiteRun -suite sysClock -test tmSysClock.c -vsb $WIND_BASE/target/vsb_ bspName -profile PROFILE_SMALL_FOOTPRINT -sfDriver -libBuild -bspToolBoard bspName toolchain boardName

Example 8-3  Building a Test Module Image Only

This example is similar to Example 8-2, except you only want to build the image (not run it using the test harness). For example, you may need to load and run the image using an ICE.

To build the image only, enter the following command:

$ vxtest suiteBuild -suite sysClock -test tmSysClock.c -vsb $WIND_BASE/target/vsb_ bspName -profile PROFILE_SMALL_FOOTPRINT -sfDriver -libBuild -bspToolBoard bspName toolchain boardName

Output for this command is sent to the shell where the command was issued. The output gives the build result (pass/fail) and either the path to the build image (if the build is successful) or a path to a build log that provides additional information about any build failures.
8.5.3 Usage Caveats

Theoretically, most any existing \texttt{vxTestV2} test module can be run with the small-footprint test harness driver. However, when a test module is intended for use with the small-footprint test harness driver, you must keep the following points in mind:

- The test module should not pull in any components nor rely on any externals that are not already part of a small footprint \texttt{vxWorks} image.
- Verbosity of prints are assumed max (level 4).
- No memory leak calculations are performed.
- No test case timing is performed.
- All test cases are run sequentially by one task, which is spawned at priority 100 from the root task during system initialization.
- No test case time-outs are detected. If a test case hangs, then the entire test suite will hang.
- There are no tests provided with support for running SMP test cases with CPU affinity. However, nothing prevents you from running the small footprint driver with an SMP kernel and the test cases will have whatever affinity (if any) is set for the root task.
- No RTP support is provided. Test cases cannot be run as RTPs.
- The \texttt{vxTestMsg()} routine only supports basic format specifiers: \%d, \%x, \%s, with prefix modifiers for \%d and \%x; 0 for leading zeros and an integer literal for print width, with - signifying left justified: (for example, \%10d, \%-30s, \%08x). No floating point conversions are supported. Specifying any unsupported format specifier results in a partial message being printed, consisting of the format string and data up until the point that the unsupported format specifier is encountered, along with a warning that the print was truncated.

8.5.4 Stubbed Driver Routines

When working with the small-footprint driver, note that the test harness driver routines shown in Table 8-1 are stubbed.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Purpose</th>
</tr>
</thead>
</table>
| char * \texttt{vxTestWindBaseGet} (void); | Get $\texttt{WIND\_BASE}$.
| int \texttt{vxTestMdlFdGet} (char * moduleName); | Get a file descriptor to read a test module object file. |
| char * \texttt{vxTestParamGet} (char * paramName); | Get a test case parameter. (deprecated) |
| void \texttt{vxTestBenchmark} (char *benchmarkID, double benchmarkValue, char *units, long iterations); | Print a benchmark result. |
| void \texttt{vxTestUsDelay} (int uSecs); | Software-based microsecond delay. |
If any of the stubbed routines are called, the small-footprint driver prints a driver error message indicating which stubbed routine was called and then continues.
9

Migrating BSPs from 32-Bit to 64-Bit

9.1 Introduction 199
9.2 BSP Requirements for 64-Bit Systems 200
9.3 Modifying a 32-Bit BSP for 64-Bit Hardware Platforms 204

9.1 Introduction

This chapter provides a general discussion of the requirements of BSPs for 64-bit systems, and an outline of the steps used to migrate the 32-bit itl_core2 BSP into the 64-bit itl_core2_64 BSP, which can be used as an example. Review the files that make up the itl_core2_64 64-bit BSP and use them as your point of reference.

NOTE: For VxWorks 6.9, the symmetric I/O mode is the default for both the uniprocessor (UP) and symmetric multiprocessor (SMP) variants of VxWorks. See VxWorks Architecture Supplement: Intel 64 for additional information about interrupt modes.

Before you consider BSP migration, you should be familiar with the coding guidelines for 64-bit hardware platforms, with changes introduced with VxWorks 64-bit memory management.

For additional information, see the VxWorks Kernel Programmer’s Guide.

NOTE: Code built for variants of VxWorks or for different releases of VxWorks is not binary compatible between variants or releases. Code must be built specifically for uniprocessor (UP) VxWorks, VxWorks SMP, 32-bit VxWorks, 64-bit VxWorks, or for a VxWorks system based on variant libraries produced with a VxWorks source build (VSB) project—or for any supported combination of these variants. The only exception to this rule is that RTP application executables can be run on both UP VxWorks and VxWorks SMP (when all else is the same).
9.2 BSP Requirements for 64-Bit Systems

To support the more complex address maps that are used in a 64-bit environment, the BSP’s interface between the kernel and the BSP must take into account the following aspects of 64-bit VxWorks:

- VxWorks always uses an MMU when running in 64-bit mode. The kernel uses the MMU to map the memory regions identified by the BSP according to the needs of the operating system. For example, the x86-64 ABI requires that memory be available at the upper 2 GB of the 64-bit address space in order to implement the kernel code model. Since Intel RAM typically is located starting at physical address 0, the MMU must be used to place the RAM at the correct virtual address.

- 64-bit hardware platforms do not always place the available RAM in a single contiguous block of the physical address space. For example, on the Intel platform, the first 3 GB of RAM is found in the lower 3 GB of the physical address space, and any additional RAM in the system is found above the 4 GB physical address boundary.

- 64-bit VxWorks does not map all of the available RAM into a single contiguous block of virtual memory.

- 64-bit VxWorks is more likely to be booted from a vendor-provided boot loader or a hypervisor environment, and therefore needs to determine its available resources by performing run-time queries, rather than referring to precompiled data structures.

- The initialization order for 64-bit VxWorks systems delays the initialization of VxBus until after the core features of the operating system (memory allocation, spinlocks, semaphores, and so on) are available.

- Not all RAM can be treated equally in a 64-bit platform. For example, some hardware devices have limitations that prevent them from performing DMA to memory located above the 4 GB boundary.

Changes to the 32-Bit BSP Model for 64-Bit BSPs

To address the differences in the 64-bit environment, the role of the BSP with regard to memory mapping is different: it must only provide the physical address, size, and memory attributes for each physical region to be used by the system. There are only two exceptions to this rule: the BSP must provide an initial static MMU map for a subset of the RAM in order to bootstrap the kernel; in addition, the BSP has the option to request virtual address space from the kernel during system startup, and to use that allocated virtual address space to create any required address maps.

The following changes to the 32-bit BSP model are required for 64-bit BSPs:

- The BSP does not implement the `sysMemTop()` and `sysPhysMemTop()` routines. The RAM used by the kernel in 64-bit systems is not located in a contiguous block that ends at `sysPhysMemTop()`.

- The BSP does not provide `sysPhysMemDesc[]` to describe the virtual memory layout and the physical memory layout to the kernel. The memory mapping is considerably more complex for 64-bit systems (not identity mapping or a
constant offset between each virtual address and its corresponding physical address), and the task of assigning virtual addresses to physical addresses is performed by the kernel itself. A `sysPhysRamDesc[]` array may be implemented in a BSP to simplify implementation of the required `sysMemDescGet()` routine, but it must be defined as `LOCAL`.

- The BSP must provide `sysMemDescInit()` and `sysMemDescGet()` routines to describe the physical address map of the hardware to the kernel— the 64-bit kernel does not reference `sysPhysMemDesc[]`. This allows for booting VxWorks in more complex environments (hypervisor, device-tree-based boot loaders, and so on). The BSP has run-time control of the query process so it can implement any required translation from the configuration provided by the boot loader to the configuration requirements of the kernel.

- The BSP’s `sysMemDescGet()` routine does not provide support for mapping the I/O regions of the physical address space. This routine is provided by VxBus. This simplifies the BSP, as it does not have to perform any device address space management, and it only has to provide information about the location of devices in the physical address space.

- The BSP’s `sysMemDescGet()` routine does not have to provide any information about the placement of memory areas into the virtual address space. For 64-bit VxWorks, the kernel is solely responsible for managing the virtual address space.

`sysMemDescInit()` and `sysMemDescGet()`

BSPs for 64-bit systems must implement `sysMemDescInit()` and `sysMemDescGet()` routines to describe the physical address map of the hardware to the kernel.

`sysMemDescInit()` Routine

The `sysMemDescInit()` routine has the following prototype:

```c
void sysMemDescInit(void)
```

The kernel calls `sysMemDescInit()` once during system startup to allow the BSP to perform any initialization that needs to be performed before `sysMemDescGet()` is called.

`sysMemDescGet()` Routine

There is considerable flexibility allowed for the implementation of `sysMemDescGet()`—no specific implementation is required within the body of the routine. This flexibility is intentional, as future BSPs will need to interact with more sophisticated run-time environments (such as being hosted by a hypervisor, or learning about the underlying hardware memory layout by traversing a vendor-provided device tree).

When VxWorks boots, the kernel makes calls into the BSP’s `sysMemDescGet()` routine to determine the BSP’s description of the physical memory map. The kernel calls this routine very early in the boot process, even before `sysHwInit()` is invoked. The BSP therefore must be structured so that it can execute `sysMemDescGet()` arbitrarily early in the boot process. The BSP must not make any assumptions about the order in which the various calls to `sysMemDescGet()` are made.
The `sysMemDescGet()` routine has the following prototype:

```c
STATUS sysMemDescGet
{
  MEM_DESC_TYPE memDescType,  /* type of memory being queried */
  int index,  /* index of memory block of requested type */
  PHYS_MEM_DESC *pMemDesc  /* used to return the description */
}
```

**memDescType Parameter**

The `memDescType` parameter to `sysMemDescGet()` can be one of the following enumeration values:

**MEM_DESC_RAM**

A region of RAM owned by the kernel. The BSP must provide at least one region (for `index = 0`), but it can provide an arbitrary number of additional regions. By supporting multiple memory regions, the BSP can provide an arbitrary amount of memory for the kernel. The kernel and any optional components included in the VxWorks configuration are responsible for partitioning the memory into the user-configured regions (kernel common heap, kernel proximity heap, and so on).

**MEM_DESC_USER_RESERVED_RAM**

A region of RAM owned by an application. The kernel does not perform any mapping operations on this memory region. If a user application wants to map this RAM, it must allocate the required virtual address space and perform the mapping operation itself.

**MEM_DESC_PM_RAM**

A region of RAM that is reserved for persistent memory. Descriptors of this type must provide a physical address, but do not provide a virtual address. This memory should be reserved by the BSP if `INCLUDE_EDR_PM` is included in the system. Depending on the `INCLUDE_EDR_PM` component include, the 0th memory segment (`sysPhysRamDesc[0]`) is reduced by `PM_RESERVED_MEM` bytes. The actual size and location is specified toward the end of the `sysMemDescInit()` function. See `sysMemDescInit()` (64-bit only), p.34 for more information on the `sysMemDescInit()` function and D.2 Example `sysMemDescInit()` Code, p.252 for an example of how it is used.

**MEM_DESC_DMA32_RAM**

A region of RAM that is reserved for 32-bit DMA operations. The BSP must ensure that the physical addresses for this RAM fall below the 4 GB address space boundary. However, some hardware architectures with IOMMU capabilities may allow this memory to be placed elsewhere in the physical address space. The BSP developer must ensure that the memory described by this descriptor can be accessed by devices that are only capable of providing a 32-bit address for DMA operations. Descriptors of this type must provide a physical address; they do not provide a virtual address. This memory should be reserved by the BSP only if `INCLUDE_DMA32_MEM` is included in the system. See `sysMemDescInit()` (64-bit only), p.34 for more information on the `sysMemDescInit()` function and D.2 Example `sysMemDescInit()` Code, p.252 for an example of how it is used.

**index Parameter**

The `index` parameter to `sysMemDescGet()` is used to determine which block of memory of the requested type to return. For each type of descriptor, the kernel
makes successive calls to the BSP’s `sysMemDescGet()` with increasing index values until `sysMemDescGet()` returns ERROR.

**pMemDesc Parameter**

The `pMemDesc` parameter to `sysMemDescGet()` is a pointer to a data structure of type `PHYS_MEM_DESC`, which is the same data structure as used in the `sysPhysMemDesc[]` array in 32-bit BSPs. The structure is defined as follows:

```c
typedef struct phys_mem_desc {
    VIRT_ADDR virtualAddr; /* virtual address */
    PHYS_ADDR physicalAddr; /* physical address */
    size_t len; /* valid if > 0 */
    UINT initialStateMask; /* mask to vmStateSet */
    UINT initialState; /* state to vmStateSet */
} PHYS_MEM_DESC;
```

The BSP is required to provide the physical address for each block of memory that it returns to the kernel. However, the BSP is only required to provide a virtual address for the 0th `MEM_DESC_RAM` block (for more information about the 0th descriptor, see Static MMU Map Requirements, p.203). This frees the BSP from having to understand the memory layout details of the kernel, since the layout of memory in virtual address space is substantially more complex in 64-bit VxWorks than in 32-bit VxWorks. For every descriptor other than the 0th `MEM_DESC_RAM` descriptor, it sets the `virtualAddr` structure element to the value `MEM_DESC_ADDR_KERNEL_ASSIGNED`.

**Example Code for sysMemDescInit() and sysMemDescGet()**

For an example of `sysMemDescInit()` and `sysMemDescGet()`, see D.2 Example `sysMemDescInit()` Code, p.252.

**Static MMU Map Requirements**

When a 64-bit BSP boots, it is initially running with the MMU disabled. This creates a specific initial runtime environment, because the VxWorks image is loaded very low in the 64-bit physical address space, but it is linked as if it was running in the upper 2 GB of the address space. While the MMU is disabled, the BSP cannot perform any CPU instructions that refer to the absolute (virtual) address of any code or data. This restriction is enforced because the virtual addresses of the text and data are not accessible by the CPU until an MMU mapping is created that maps the virtual address space for the kernel to the physical address into which the VxWorks image is loaded.

To address this limitation, one of the first tasks that the BSP must perform is to turn on the MMU so that the VxWorks image becomes “visible” at the correct virtual address in the upper 2 GB of the virtual address space. The BSP must therefore create the appropriate page table structures to support the initial bootstrap of the kernel. At a minimum, this page table structure must contain at least two distinct map areas:

- The static map must provide a mapping of at least all of the RAM described in the 0th `MEM_DESC_RAM` descriptor returned by `sysMemDescGet()`, mapped at the address `LOCAL_MEM_LOCAL_ADRS`. For example, if the 0th descriptor returned by `sysMemDescGet()` describes a 256 megabyte region of RAM, then the static map must map at least this 256 megabytes of RAM at `LOCAL_MEM_LOCAL_ADRS`. The VxWorks image is linked to run within this
memory area, so this part of the memory map is used to execute kernel code once the MMU is enabled.

- The static map must also map a very small portion of the VxWorks image with an identity mapping, to avoid causing a page fault right after the MMU is enabled. For example, if a BSP implements a sysMmuEnable() routine, the program counter points to the physical address of sysMmuEnable() in low memory when this routine is invoked. As the routine executes, the program counter remains in this low memory area. Once sysMmuEnable() turns on the MMU, all subsequent instruction fetches are treated as virtual address references, and are translated using the installed memory map. Since the program counter is not automatically updated to account for the new mapping when the MMU is turned on, it remains in low memory as it tries to execute the instructions within sysMmuEnable() after the MMU is enabled. To keep these instruction fetches from causing an MMU fault, the code for sysMmuEnable() must be identity mapped by the static map.

The BSP is free to map more memory in its static map than is strictly required by the above rules. For example, a BSP could be structured so that it maps 2 GB of its physical memory using a identity mapping, and also at LOCAL_MEM_LOCAL_ADRS.

Again, note that the static map provided by the BSP can map substantially more RAM than is described by the 0th MEM_DESC_RAM descriptor. This is allowed because the VxWorks kernel accesses memory only after it has queried the BSP for its memory descriptors. Under no circumstances does the kernel attempt to access any memory outside of the region(s) described by the descriptor(s) that are returned by sysMemDescGet().

9.3 Modifying a 32-Bit BSP for 64-Bit Hardware Platforms

Changes to Files

Migrating a BSP for 64-bit support involves changes to the following files:

- Makefile
- pc.h
- romInit.s
- sysALib.s
- sysLib.c
- config.h
- 20bsp.cdf
- 00bsp.cdf
- and possibly sysSerial.c and sysNvRam.c

Note that if no modifications were made to a given file since obtaining it from Wind River, you may be able to simply replace it with the one provided in this release.
Carefully examine the files in the `itl_core2_64` BSP directory to clarify the modifications described in this section.

**Makefile**

Your makefile should incorporate the elements contained in the makefile for the `itl_core2_64` BSP:

```makefile
CPU     = CORE
TOOL    = gnu
VXBUILD = LP64

TGT_DIR = $(WIND_BASE)/target

include $(TGT_DIR)/h/make/defs.bsp2.mk

## Only redefine make definitions below this point, or your definitions will
## be overwritten by the makefile stubs above.

TARGET_DIR = itl_core2_64
VENDOR =
BOARD = Core2

# The constants ROM_TEXT_ADRS, ROM_SIZE, and RAM_HIGH_ADRS are defined
# in config.h, MakeSkel, Makefile, and Makefile.*
# All definitions for these constants must be identical.
#
# ifdef BOOTCODE_IN_RAM
ROM_TEXT_ADRS = 00008000# ROM entry address - A: or C:
ROM_SIZE = 00090000# number of bytes of ROM space
else
ROM_TEXT_ADRS = fff20000# ROM entry address - EPROM
ROM_SIZE = 0007fe00# number of bytes of ROM space
endif

EXTRA_DEFINE = -D_WRS_CONFIG_LP64 -fno-zero-initialized-in-bss
MACH_EXTRA =

RELEASE_CMD =
RELEASE_2PRJ = vip_lp64_bootrompXE vxWorks_lp64 vxWorks_lp64_smp

RELEASE = $(RELEASE_PRE) $(RELEASE_CMD) $(RELEASE_2PRJ) $(RELEASE_POST)
RELEASE_TOOLS = gnu icc

%.pxe:  $(BSP_DIR)/pxeBoot.bin %.bin
cat $+ > $@

## Only redefine make definitions above this point, or the expansion of
## makefile target dependencies may be incorrect.

include $(TGT_DIR)/h/make/rules.bsp2.mk
```

**NOTE:** `vxWorks_lp64_smp` is only added for BSPs that support SMP in addition to LP64.

**pc.h**

Add the following definitions for `sysALib.s` to `pc.h`:

```c
#define WARMBOOT_COPY_ADRS   0x500
#define IA32_EFER       0xc0000080
#define IA32_EFER_SCE   0x00000001      /* SysCall enable (bit 0) */
#define IA32_EFER_LME   0x00000100      /* IA-32e mode enable (bit 8) */
#define IA32_EFER_LMA   0x00000400      /* IA-32e mode active (bit 10) */
```
#define IA32_EFER_NX 0x00000800 /* Execute disable bit enable (bit 11) */

#define CR0_ET 0x00000010

#define CR0_INIT_VALUE (CR0_PE | CR0_MP | CR0_ET | CR0_NE | CR0_WP | \ CR0_AM | CR0_PG)

#define CODE_SEGMENT 0x08
#define DATA_SEGMENT 0x10

#define STATIC_MMU_TBL_PAGE_ATTR_OFF (0x83 | 0x18)
#define STATIC_MMU_TBL_PAGE_ATTR_WT (0x83 | 0x08)
#define STATIC_MMU_TBL_PAGE_ATTR_CB (0x83 | 0x00)
#define STATIC_MMU_TBL_PAGE_ATTR_READONLY (0x81 | 0x18)
#define STATIC_MMU_TBL_PAGE_ATTR_INVALID (0x80 | 0x18)
#define STATIC_MMU_TBL_PAGE_ATTR_CB_4K (0x03 | 0x00)
#define STATIC_MMU_TBL_PAGE_ATTR_READONLY_4K (0x01 | 0x18)
#define STATIC_MMU_TBL_PAGE_ATTR_INVALID_4K (0x00 | 0x18)
#define STATIC_MMU_TBL_PAGE_ATTR_RAM_STATIC_MMU_TBL_PAGE_ATTR_CB
#define STATIC_MMU_TBL_PAGE_ATTR_IO_STATIC_MMU_TBL_PAGE_ATTR_OFF
#define STATIC_MMU_TBL_PAGE_ATTR_PROTECTED_STATIC_MMU_TBL_PAGE_ATTR_READONLY
#define STATIC_MMU_TBL_PAGE_ATTR_RAM_4K STATIC_MMU_TBL_PAGE_ATTR_CB_4K
#define STATIC_MMU_TBL_PAGE_ATTR_PROTECTED_4K STATIC_MMU_TBL_PAGE_ATTR_INVALID_4K

/* Static MMU defines */

#define PAGE_ENTRIES_MAX 512
#define PAGE_OFFSET_PD 21 /* PD offset bits 21-29 */
#define PAGE_OFFSET PAGE_SIZE_4KB

#define PAGE_P 0x0000000000000001 /* Present bit */
#define PAGE_RW 0x0000000000000002 /* Read/Write bit */
#define PAGE_US 0x0000000000000004 /* User/Supervisor bit */

#define PAGE_PML4_NAME(label) page_table_ ## label
#define PAGE_BEGIN_MMU_VARS \ $page = 0;
#define PAGE_BEGIN_PT_VARS \ index = 0;
#define PAGE_BEGIN_PT(label,base) \ $page = $page + 1; \ .org $page * PAGE_OFFSET ; \ pd_ ## label = $page * PAGE_OFFSET + base;
#define PAGE_CREATE_PT(label,properties) \ .quad pt_ ## label | properties; \ index = index + 1;
#define PAGE_CREATE_PTS(count,properties) \ .rept count; \ .quad ((index << PAGE_OFFSET_PD) | properties); \ index = index + 1; \ .endr;

#define PAGE_BEGIN_PD_VARS \ index = 0;
#define PAGE_BEGIN_PD(label,base) \ $page = $page + 1; \ .org $page * PAGE_OFFSET ; \ pdp_ ## label = $page * PAGE_OFFSET + base;
#define PAGE_CREATE_PD(label,properties) \ .quad pd_ ## label | properties; \ index = index + 1;
#define PAGE_CREATE_PDS(count,properties) \ .rept count; \ .quad ((index << PAGE_OFFSET_PD) | properties); \ index = index + 1; \ .endr;

#define PAGE_BEGIN_PDP(label,base) \
9 Migrating BSPs from 32-Bit to 64-Bit

9.3 Modifying a 32-Bit BSP for 64-Bit Hardware Platforms

In `romInit.s`, modify the `romInit` routine to go from 32-bit to 64-bit operational mode. To do so, go through the `romInit` routine provided in the `itl_core2_64` BSP and compare it against your BSP line-by-line. Note that you must also use the static MMU table configured for 64 bits that is provided in the `itl_core2_64` BSP.

In `sysALib.s`, modify the `sysInit` routine to go from the 32-bit to 64-bit operational mode. To do so, go through the `sysInit` routine provided in the `itl_core2_64` BSP and compare it against your BSP line-by-line. Note that you must also use the static MMU table configured for 64 bits that is provided in the `itl_core2_64` BSP.

In `sysLib.c`, you must make changes with regard to memory description, as described in 9.2 BSP Requirements for 64-Bit Systems, p.200, including implementation of `sysMemDescInit()` and `sysMemDescGet()`. The `sysMemDescShow()` routine is a useful utility. For examples, see the `sysLib.c` file for the `itl_core2_64` BSP and D. BSP `sysMemDescInit()` and `sysMemDescGet()` Examples.

Additional stream lining of `config.h` has been performed for project builds. Please refer to `config.h` provided in the `itl_core2_64` BSP. Add or change the parameters in the following example, as illustrated:

```
#define LOCAL_MEM_PHYS_ADRS_MASK        0x00000007fffffff
#define LOCAL_MEM_VIRT_TO_PHYS(addr)    ((addr) & (LOCAL_MEM_PHYS_ADRS_MASK))
#define SYSTEM_RAM_SIZE         (0x80000000)
#define LOCAL_MEM_SIZE  (SYSTEM_RAM_SIZE -
LOCAL_MEM_VIRT_TO_PHYS(LOCAL_MEM_LOCAL_ADRS))

#if defined(INCLUDE_BOOT_APP) && !defined(BOOTAPP)
/* This section for project built bootApp */
```
/* VxWorks image entry point */
#define SYS_RAM_LOW_ADRS  LOCAL_MEM_VIRT_TO_PHYS(RAM_HIGH_ADRS)
/* Boot image entry point */
#define SYS_RAM_HIGH_ADRS  LOCAL_MEM_VIRT_TO_PHYS(RAM_LOW_ADRS)
#else
/* VxWorks image entry point */
#define SYS_RAM_LOW_ADRS  LOCAL_MEM_VIRT_TO_PHYS(RAM_LOW_ADRS)
/* Boot image entry point */
#define SYS_RAM_HIGH_ADRS  LOCAL_MEM_VIRT_TO_PHYS(RAM_HIGH_ADRS)
#endif /* INCLUDE_BOOT_APP */

00bsp.cdf
Refer to the 00bsp.cdf file provided in the itl_core2_64 BSP.

20.bsp.cdf
Set the definitions for the following CDF elements as is done in the itl_core2_64 20bsp.cdf:
- CPU
- DATA_MODEL
- RAM_HIGH_ADRS
- RAM_LOW_ADRS
- LOCAL_MEM_LOCAL_ADRS
Refer to the 20bsp.cdf file provided in the itl_core2_64 BSP.

20profiles.cdf
Refer to 20profiles.cdf provided in the itl_core2_64 BSP.

sysSerial.c
The sysSerial.c file may require prototype modifications.

sysNvRam.c
The sysNvRam.c file may require prototype modifications.
A.1 Introduction

This appendix presents a summary of common issues and concerns encountered during BSP development. Many of these problems are discussed in 2.5 Common Problems, p.51, and throughout the main chapters of this document as part of the development process. The information in this appendix provides additional information on these problems and others that you may encounter during the development process.

A.2 The Development Environment

There are a number of problems that can occur due to your choice of development environment. Many of these problems are related to the relationship between the addresses where the linker expects the code to run and the addresses at which the code is actually located. In most situations, these addresses should be identical. However, for a short period of time, the locations can be different.

For example, when the target processor receives a RESET signal, it starts to execute at a specific reset vector address, typically determined by the processor design. This address is usually where flash is located on the board. However, it is often best
if the boot image is executed from RAM. For this reason, many boot loaders have a short section of position-independent code (PIC) that copies the entire image from flash to RAM, and then transfers control to the RAM copy.

When the boot loader loads the actual OS image, the image is placed at a given location in RAM and the loader causes execution to be transferred to the OS image. The RAM address that the image is loaded at must match the address used in the object file. If these addresses do not match, or if the image must be started with some offset from the normal start location, the OS image that is loaded does not run correctly.

For more information on setting up your development environment, see 2.4 The Development Environment, p.45.

A.2.1 Image Locations

There are several locations that are relevant to the image you are debugging:

- the flash reset address for the boot loader
- the RAM address for the boot loader
- the RAM address for the OS image

The boot loader is put into flash at the processor's reset address, and then copied into RAM when the system boots. When the boot loader executes, it reads a VxWorks image file and puts the image into RAM at a third address.

It is important that the two RAM addresses are each large enough to contain the entire image. For example, if the OS image is to be loaded at 0x00040000 (128 KB) and the boot loader's RAM image is at 0x00100000 (1 MB), the OS image must be no larger than 896 KB (1 MB minus 128 KB), or the OS image overwrites the boot loader before the boot loader starts executing the OS image.

If your system becomes unresponsive, verify that the RAM addresses the code is actually loaded at match the addresses used in the code.

A.2.2 Position-Independent Code

One exception to the matching RAM addresses rule is the PIC instructions at the beginning of the flash image.

In VxWorks, there is a short section of PIC that copies the image from flash to RAM and then transfers execution to the copy in RAM. This includes the romInit() and romStart() routines discussed in 2.2.4 Detailed Boot Sequence, p.14. The OS image code must be linked to reside at the RAM address. However, when the image code is programmed into flash, the addresses are different.

When verifying the addresses of the code in flash, the romInit() address must match the address of the processor’s reset vector. This does not match the address in the bootrom image file.
A.3 Exception Handling and Debug Tools

When porting VxWorks to your target board (single or multicore), you must be sure the exception handling continues to work in the manner expected by the Wind River debugging tools. You can verify that your ported BSP does not break the debugging tools by testing the b, c, and s commands in the VxWorks kernel shell for both kernel tasks and RTP tasks. Then, run the WTX connection test from Workbench by right-clicking the target connection and selecting Target Tools > Run Debugger WTX Connection Test. You can also run additional tests by running the Workbench debugger tutorials available by selecting File > New > Example... and running the desired tutorial.

For more information on exception handling in your BSP, see the target.ref file (or BSP reference entry) for your reference BSP.

A.4 Cache and MMU

A BSP that is running correctly without cache often encounters problems when cache is enabled for the first time. In most cases, these problems are not the result of problems with the cache library.

NOTE: In this discussion, the terms device register and register may refer to the actual registers on some peripheral device or they may refer to structures in RAM that are manipulated by both the processor and the device.

A.4.1 Register Access

When accessing device registers or other shared memory, the processor should invalidate the cache line before reading from the register, and it should flush the cache line immediately after writing to the register. Failure to perform these steps in the appropriate places in a driver or BSP can cause cache coherency problems. When reading from a device register, the main processor may find the register data in cache. In this case, it does not actually check the hardware unless the cache line has been invalidated. Therefore, the value being read may no longer be valid. When writing to a device register, the main processor puts information into a cache line, but does not write it to RAM. Meanwhile, before the processor writes the cached information to RAM, some other device can modify the memory that the cache entry points to. In this case, the modification that was made by the device is never visible to the processor.

When cache is enabled, the initial setting should be to write-through mode if possible. In this mode, the processor does not write to cache without modifying RAM.

If problems occur, they are likely caused by failure to invalidate the cache before reading a device register. When the processor is set to write-back mode, problems can occur both from failure to invalidate the cache before reading the device register and from failure to flush the cache after writing to the device register.
Once the system works in write-through mode, the cache can be configured to write-back mode. If problems occur at this point, they are most likely related to failure to flush the cache after writing to a device register.

In general, if the problem cannot be isolated reasonably quickly, a good procedure is to disable all possible devices, and then re-introduce them into the system one at a time. In this way, when the problem occurs, you know which driver is causing it.

As specified previously, problems occurring when the cache is configured as write-through are often related to failure to invalidate cache before reading a device register. Problems occurring when the cache is configured in write-back mode can be caused by a failure to flush the cache immediately after a write operation.

A.4.2 Timing Issues

Another set of problems related to cache are timing issues. These problems are extremely difficult to debug. If a device writes to a register between the time that the processor writes to a register within the same cache line and the time that the processor flushes the write, the device may not function correctly. It is best to insure that the driver does not write to device registers at a time when the device may also be modifying a nearby register.

How you accomplish this depends on the design of the device.

A.5 Non-portable Code

A common problem when creating BSPs is that code for some devices may have been taken from other sources, and that code may not have been written with portability in mind.

When using code from other sources, you can prevent problems by examining the code for portability before including it in your BSP.

A.6 Volatile Variables

Normally, a compiler is allowed to generate code that maintains copies of any variable in a register, so that the variable does not need to be fetched from memory multiple times. For most application code, this is appropriate behavior. However, for many global variables in BSPs and drivers, there are multiple threads that access the variables, and maintaining a local copy in a register may cause problems.

All variables that are used from more than one thread, including software threads on the main processor and threads of execution on external devices, should be declared using the `volatile` keyword of C. At the time of this writing, current
practice for VxWorks is to use a compiler argument indicating that all BSP and driver variables are to be treated as volatile; it is nevertheless recommended that all variables manipulated by multiple threads be explicitly marked as \textit{volatile}.

A.7 Conflicts Between Virtual and Physical Memory

With VxWorks 5.5 and earlier, VxWorks typically used a flat memory model, which meant that physical addresses and virtual addresses were the same. Device drivers and BSP code could ignore the difference between virtual and physical addresses, and the driver or BSP would still work.

However, there are several reasons why this is not good programming practice. In many modern processors, the physical address space has grown from 32 bits to a larger number such as 36 bits or even 64 bits. However, because VxWorks is still essentially a 32-bit operating system, the virtual address is limited to 32 bits. This means that drivers written without regard to the difference between virtual and physical memory cannot be used on these systems.

In addition, with the introduction of real-time processes (RTPs) in VxWorks 6.x, there is no longer a flat memory model. For drivers to work correctly with VxWorks 6.x, the driver must be aware of the difference between physical and virtual addresses.

For more information, see the \textit{VxBus Device Driver Developer's Guide: Device Driver Fundamentals}. 
B

Implementing Documentation Guidelines

B.1 Introduction 215
B.2 Written Style 216
B.3 Sections for Libraries and Subroutines 222
B.4 Sections for target.ref 228
B.5 Format and Style 230
B.6 Directives 240
B.7 Converting target.nr Files 242
B.8 Generating Reference Entries 243

NOTE: The instructions in this chapter are applicable to both VxWorks 5.5 and VxWorks 6.x users unless otherwise noted. However, the documentation tool provided for the VxWorks 5.5 release is called refgen. In most cases, refgen and apigen are used in the same manner and accept the same syntax and commands. In this chapter, “apigen” means both refgen and apigen unless otherwise noted.

B.1 Introduction

Reference documentation for Wind River board support packages (BSPs) consists of UNIX-style reference entries (formerly known as man pages) for the module sysLib.c and the file target.ref. Documentation in HTML format is generated from these files with the Wind River tool apigen. During a BSP build, make runs apigen and places the HTML output in the docs directory of your installation. The resulting reference entries can be displayed online with an HTML browser.

This chapter covers Wind River conventions for style and format, and the procedures for generating BSP documentation. The BSP templates supplied with the VxWorks provide examples of the writing style, text format, module layout, and text commands discussed throughout this chapter.

Modules formatted with the conventions discussed here are compatible with all Wind River documentation markup and formatting scripts. This is a requirement for BSPs that are turned over to Wind River for distribution.
B.2 Written Style

This section describes a few of the general requirements for written style in Wind River technical publications. The items that follow are only a portion of the standards described in Wind River’s style guide, but are chosen for inclusion here based on their frequent misuse.

Specific requirements for BSPs are in B.3 Sections for Libraries and Subroutines, p. 222, and B.4 Sections for target.ref, p. 228.

Sentences

- Keep sentences brief and to the point, presenting information in a simple, straightforward manner.
- Always use complete sentences.
- Keep sentences in present tense. Do not use future or past tense unless they are necessary to convey the idea.
- Do not use abbreviated English—do not exclude articles (the, a, an) for brevity.

Punctuation

- Always use a colon after the phrase or sentence introducing an example, display, itemized list, or table.
- A comma should always precede the conjunction and, or, or nor when it separates the last of a series of three or more words or phrases. This comma is not optional. For example:
  
  apples, oranges, and bananas

- Avoid the use of quotation marks. If they are necessary, form quotations using the straight double-quote (" ) only. Use single quotes only as described in Special Words, p. 233.

Word Usage

- Do not use capital letters to convey emphasis; use italics. For information on how to apply font changes, see Table B-5. In general, avoid applying italics for emphasis—the best way to convey emphasis is a well cast sentence.
- Do not use the word so to mean thus or therefore. However, the construction so that is acceptable.
- Do not use contractions (don’t, doesn’t, can’t, and so on).

Spelling

Table B-1 defines the Wind River standard for terms that are spelled inconsistently, particularly in the computer industry. This table also includes a few words or abbreviations that are commonly misspelled, and words whose spelling is frequently misunderstood because it may depend on context.
Table B-1  **Spelling Conventions**

<table>
<thead>
<tr>
<th>Use...</th>
<th>Not...</th>
</tr>
</thead>
<tbody>
<tr>
<td>and so forth, among others</td>
<td>etc.</td>
</tr>
<tr>
<td>back end</td>
<td>backend</td>
</tr>
<tr>
<td>backward</td>
<td>backwards</td>
</tr>
<tr>
<td>baseline</td>
<td>base line</td>
</tr>
<tr>
<td>basename (of filename)</td>
<td>base name</td>
</tr>
<tr>
<td>bit-field</td>
<td>bit field</td>
</tr>
<tr>
<td>boot line</td>
<td>bootline, boot-line</td>
</tr>
<tr>
<td>boot loader</td>
<td>bootloader</td>
</tr>
<tr>
<td>boot ROM</td>
<td>bootrom, boot rom, bootROM</td>
</tr>
<tr>
<td>bring up (v.)</td>
<td>bringup, bring-up</td>
</tr>
<tr>
<td>bring-up (n., adj.)</td>
<td>bringup, bring up</td>
</tr>
<tr>
<td>bps</td>
<td>BPS, baud</td>
</tr>
<tr>
<td>caching</td>
<td>cacheing</td>
</tr>
<tr>
<td>cacheable</td>
<td>cachable</td>
</tr>
<tr>
<td>callback</td>
<td>call-back</td>
</tr>
<tr>
<td>cannot</td>
<td>can not</td>
</tr>
<tr>
<td>CD-ROM</td>
<td>CDROM, cdrom</td>
</tr>
<tr>
<td>coprocessor</td>
<td>co-processor</td>
</tr>
<tr>
<td>countdown</td>
<td>count-down</td>
</tr>
<tr>
<td>cross-compiler</td>
<td>cross compiler</td>
</tr>
<tr>
<td>cross-development</td>
<td>cross development</td>
</tr>
<tr>
<td>cross-reference</td>
<td>cross reference</td>
</tr>
<tr>
<td>data type</td>
<td>datatype</td>
</tr>
<tr>
<td>dialog</td>
<td>dialogue</td>
</tr>
<tr>
<td>e-mail</td>
<td>email, E-mail, Email</td>
</tr>
<tr>
<td>Ethernet</td>
<td>ethernet</td>
</tr>
<tr>
<td>Excelan</td>
<td>Excellan</td>
</tr>
<tr>
<td>extensible</td>
<td>extendable, extendible</td>
</tr>
<tr>
<td>fax</td>
<td>FAX</td>
</tr>
<tr>
<td><em>fd</em></td>
<td>FD</td>
</tr>
<tr>
<td>Excelan</td>
<td>Excellan</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>extensible</td>
<td>extendable, extendible</td>
</tr>
<tr>
<td>fax</td>
<td>FAX</td>
</tr>
<tr>
<td>$fd$</td>
<td>FD</td>
</tr>
</tbody>
</table>
### Spelling Conventions (cont'd)

<table>
<thead>
<tr>
<th>Use...</th>
<th>Not...</th>
</tr>
</thead>
<tbody>
<tr>
<td>motherboard</td>
<td>mother-board, mother board</td>
</tr>
<tr>
<td>multiprocessor</td>
<td>multi-processor</td>
</tr>
<tr>
<td>multitasking</td>
<td>multi-tasking</td>
</tr>
<tr>
<td>multi-user</td>
<td>multiuser</td>
</tr>
<tr>
<td>nonvolatile</td>
<td>non-volatile</td>
</tr>
<tr>
<td>nonzero</td>
<td>non-zero</td>
</tr>
<tr>
<td>on-board</td>
<td>on board, onboard</td>
</tr>
<tr>
<td>online</td>
<td>on-line</td>
</tr>
<tr>
<td>overrun</td>
<td>over-run, over run</td>
</tr>
<tr>
<td>overwrite</td>
<td>over-write, over write</td>
</tr>
<tr>
<td>PAL</td>
<td>pal</td>
</tr>
<tr>
<td>pathname</td>
<td>path name</td>
</tr>
<tr>
<td>plug-in</td>
<td>plugin</td>
</tr>
<tr>
<td>pop-up</td>
<td>popup</td>
</tr>
<tr>
<td>POSIX</td>
<td>Posix</td>
</tr>
<tr>
<td>preemptive</td>
<td>pre-emptive</td>
</tr>
<tr>
<td>printout</td>
<td>print-out</td>
</tr>
<tr>
<td>real-time, Real-time</td>
<td>realtime, Real-Time (not even in titles)</td>
</tr>
<tr>
<td>reentrant</td>
<td>re-entrant</td>
</tr>
<tr>
<td>RSH</td>
<td>rsh</td>
</tr>
<tr>
<td>run-time, Run-time (adj.), run time (n.)</td>
<td>runtime, Run-Time</td>
</tr>
<tr>
<td>SBus</td>
<td>S-Bus, Sbus</td>
</tr>
<tr>
<td>scalable</td>
<td>scaleable</td>
</tr>
<tr>
<td>SCSI</td>
<td>Scsi, scsi</td>
</tr>
<tr>
<td>set up (v.)</td>
<td>set-up</td>
</tr>
<tr>
<td>setup (n., adj.)</td>
<td>set-up</td>
</tr>
<tr>
<td>shell script</td>
<td>shellscript</td>
</tr>
<tr>
<td>single-stepping</td>
<td>single stepping</td>
</tr>
<tr>
<td>standalone</td>
<td>stand-alone</td>
</tr>
<tr>
<td>start up (v.)</td>
<td>startup, start-up</td>
</tr>
</tbody>
</table>
Table B-1  **Spelling Conventions** (cont’d)

<table>
<thead>
<tr>
<th>Use…</th>
<th>Not…</th>
</tr>
</thead>
<tbody>
<tr>
<td>startup (n., adj.)</td>
<td>start-up</td>
</tr>
<tr>
<td>stdio.h</td>
<td>stdio</td>
</tr>
<tr>
<td>subclass</td>
<td>sub-class</td>
</tr>
<tr>
<td>subdirectory</td>
<td>sub-directory</td>
</tr>
<tr>
<td>SunOS</td>
<td>SUN OS</td>
</tr>
<tr>
<td>superclass</td>
<td>super-class</td>
</tr>
<tr>
<td>task ID</td>
<td>task id</td>
</tr>
<tr>
<td>Tcl</td>
<td>TCL, tcl</td>
</tr>
<tr>
<td>TFTP</td>
<td>tftp</td>
</tr>
<tr>
<td>that is</td>
<td>i.e.</td>
</tr>
<tr>
<td>timeout</td>
<td>time-out</td>
</tr>
<tr>
<td>timestamp</td>
<td>time stamp, time-stamp</td>
</tr>
<tr>
<td>title bar</td>
<td>titlebar</td>
</tr>
<tr>
<td>TTY</td>
<td>tty</td>
</tr>
<tr>
<td>underrun</td>
<td>under-run, under run</td>
</tr>
<tr>
<td>UNIX</td>
<td>Unix</td>
</tr>
<tr>
<td>uppercase</td>
<td>upper-case, upper case</td>
</tr>
<tr>
<td>Users’ Group</td>
<td>User’s Group, Users Group</td>
</tr>
<tr>
<td>VxWorks</td>
<td>VxWORKS, VXWORKS, vxWorks</td>
</tr>
<tr>
<td>Web</td>
<td>web</td>
</tr>
<tr>
<td>Web site</td>
<td>website</td>
</tr>
<tr>
<td>Wind River</td>
<td>WindRiver</td>
</tr>
<tr>
<td>workaround (n.)</td>
<td>work-around</td>
</tr>
<tr>
<td>write-through, write-back</td>
<td>writethrough, writeback</td>
</tr>
</tbody>
</table>

**Acronyms**

Define acronyms at first usage, except for widely recognized acronyms (see **Table B-2**). At first usage, give the full definition, followed by the acronym in parentheses, for example:

Internet Control Message Protocol (ICMP)

Do not use an apostrophe (’ ) to form the plural of an acronym. The plural of CPU is CPUs.
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>API</td>
<td>application programming interface</td>
</tr>
<tr>
<td>bss</td>
<td>segment of memory that holds uninitialized variables</td>
</tr>
<tr>
<td>CPU</td>
<td>central processing unit</td>
</tr>
<tr>
<td>DOS</td>
<td>Disk Operating System</td>
</tr>
<tr>
<td>dosFs</td>
<td>DOS-like file system</td>
</tr>
<tr>
<td>EOF</td>
<td>end-of-file</td>
</tr>
<tr>
<td>FCC</td>
<td>Federal Communications Commission</td>
</tr>
<tr>
<td>fd</td>
<td>file descriptor</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>GUI</td>
<td>graphical user interface</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>I/O</td>
<td>input/output</td>
</tr>
<tr>
<td>IP</td>
<td>Internal Protocol</td>
</tr>
<tr>
<td>LAN</td>
<td>local-area network</td>
</tr>
<tr>
<td>NFS</td>
<td>Network File System</td>
</tr>
<tr>
<td>PDF</td>
<td>Portable Document Format</td>
</tr>
<tr>
<td>PPP</td>
<td>Point-to-Point Protocol</td>
</tr>
<tr>
<td>PTY</td>
<td>pseudo terminal device</td>
</tr>
<tr>
<td>RAM</td>
<td>random access memory</td>
</tr>
<tr>
<td>rawFs</td>
<td>raw file system</td>
</tr>
<tr>
<td>ROM</td>
<td>read-only memory</td>
</tr>
<tr>
<td>RSH</td>
<td>Remote Shell</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TFTP</td>
<td>Trivial File Transfer Protocol</td>
</tr>
<tr>
<td>TTY</td>
<td>teletypewriter (terminal device)</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>WAN</td>
<td>wide-area network</td>
</tr>
</tbody>
</table>
Board Names

Names used for target boards should correspond to the names used by their suppliers; for example, MV5500 is not an acceptable name for the MVME5500.

When multiple board models are covered by the same board support package, and the portion of their names that differs is separated by a slash (\/) or a hyphen (-), these portions can be repeated, each separated by a comma and a space. See the examples below:

Force SYS68K/CPU-21, -29, -32
Heurikon HK68/V2F, V20, V2FA

However:

Motorola MVME147, MVME147S-1

B.3 Sections for Libraries and Subroutines

This section discusses special stylistic considerations for BSP library and subroutine documentation on a section-by-section basis.

In the examples that follow, mfr\&board means the manufacturer’s name plus the full model name of the board, as described in Board Names, p.222.

Reference entries for libraries and subroutines always contain the sections shown in Table B-3 in the order shown; other sections can be included as needed.

<table>
<thead>
<tr>
<th>Section Name</th>
<th>Library Entry</th>
<th>Routine Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>X</td>
<td>X</td>
<td>The title line, containing the name of the element and a short, one-line description.</td>
</tr>
<tr>
<td>ROUTINES</td>
<td>X</td>
<td></td>
<td>The summary of routines provided by this library, generated automatically by apigen.</td>
</tr>
<tr>
<td>SYNOPSIS</td>
<td></td>
<td>X</td>
<td>The routine declaration, generated automatically by apigen.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>X</td>
<td>X</td>
<td>An overall description of the module.</td>
</tr>
<tr>
<td>INCLUDE FILES</td>
<td>X</td>
<td></td>
<td>The relevant .h files.</td>
</tr>
<tr>
<td>RETURNS</td>
<td></td>
<td>X</td>
<td>The values returned.</td>
</tr>
<tr>
<td>ERRNO</td>
<td></td>
<td>X</td>
<td>The list of ERRNO values set.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>X</td>
<td>X</td>
<td>Cross-references to reference entries for other libraries and routines, or other user manuals.</td>
</tr>
</tbody>
</table>

Special considerations for these sections are discussed in the following sections.
NAME Section

This section is generated automatically. The text is taken from the one-line title of the C file or the routine.

- **Libraries**

Describe briefly what this collection of routines does. The hyphen must appear exactly as indicated (space-hyphen-space)—do not use backslashes or double hyphens. The general format is:

```
nameLib.c - the such-and-such library
```

For example:

```
sysALib.s - mfr&board system-dependent assembly routines
sysLib.c - mfr&board system-dependent library
```

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

- **Routines**

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

```
name - do such and such
```

For example:

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

Do not include the subroutine parentheses in the heading; the `apigen` process adds them in so that they appear in the final reference entry.

⚠️ **CAUTION:** The routine heading and definition must be limited to one line only, though wrapping is permitted. An EOL within the NAME causes confusing formatting errors through the library or routine section.

ROUTINES Section

This section is generated automatically and lists all subroutines in the library that are not declared `LOCAL`, `static`, or marked `NOMANUAL`.

SYNOPSIS Section

**C Routines**

For a C routine, this section is the declaration. The section heading is generated automatically and the text is picked up from the declaration in the code, along with the short comments describing each parameter. In unusual cases where the code declaration is not appropriate, a SYNOPSIS section can be typed manually in the routine-header comment block. If `apigen` sees a manually entered SYNOPSIS, it replaces the one encountered in the routine code.
Tcl Procedures, Scripts, Commands

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

- Enter the calling syntax and parameters between the tags `\ss` and `\se`.
- Show parameters that are optional in square brackets.
- Use the bar character (|) to indicate “or”.
- Represent a variable list of arguments with three dots (...).
- Bracket arguments between angle brackets (< and >) when they are placeholders for user-supplied values.
- If angle brackets are meant to indicate redirection of standard input/output, surround them with space characters.

Example command or script input:

```bash
# SYNOPSIS
# \
```

Resulting output:

```
SYNOPSIS
```

DESCRIPTION Section

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

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

Parameter Lists

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

Begin the parameter list with the word “Parameters” followed by a colon. Format the list with the item-list tags `\is`, `\i`, and `\ie` (for more information, see *Item Lists (Definition Lists or Terms Lists)*, p. 237.) Each parameter should be identified with the `\i` tag, which should be followed by sentences explaining what it is, what it does, what sort of values it expects, and how it is used. To keep the input readable,
separate each parameter item with a blank line. For example, consider the routine `unixDiskDevCreate()` with the following declaration:

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

{ }
```

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

* Parameters:
  * `unixFile`  
    * The name of the UNIX file for the disk device.
  * `bytesPerBlk`  
    * The size of each logical block on the disk. If zero, the default is 512.
  * `blksPerTrack`  
    * The number of blocks on each logical track of the disk. If zero, the count of blocks per track is set to `nBlocks`; that is, the disk is defined as having only a single track.
  * `nBlocks`  
    * The size of the disk in blocks. If zero, a default size is used; the default is calculated as the size of the UNIX disk divided by the number of bytes per block.

When generated, the above will appear as follows:

**Parameters:**

* **unixFile**
  
  The name of the UNIX file for the disk device.

* **bytesPerBlk**
  
  The size of each logical block on the disk. If zero, the default is 512.

* **blksPerTrack**
  
  The number of blocks on each logical track of the disk. If zero, the count of blocks per track is set to `nBlocks`; that is, the disk is defined as having only a single track.

* **nBlocks**
  
  The size of the disk in blocks. If zero, a default size is used; the default is calculated as the size of the UNIX disk divided by the number of bytes per block.

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

**CORRECT:** The name of the UNIX file for the disk device.
**INCORRECT:** Specifies the name of the UNIX file for the disk device.
**INCORRECT:** `<unixFile>` specifies the name of the UNIX file for the disk device.
**INCORRECT:** name of UNIX file for disk device.
Any subsequent definition text that follows the sentence fragment must consist of true complete sentences.

**INCLUDE FILES Section**

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

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

**RETURNS Section**

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

```
RETURNS: N/A
```

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

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

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

- Keep return statements in present tense, even if the conditions that cause an ERROR or any other return value may be thought of as “past” once ERROR is returned. In some cases the return value will be “OK, always” and “ERROR, always.”

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

- In STATUS returns, ERROR must be followed by a qualifying statement. Always enter a comma after “OK,” because it must be clear that the qualifier belongs to the ERROR condition and not the OK. For example:

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

- Do not preface lines of text with extra leading spaces. An input line whose first character is a space will cause a line break. In the past, some authors applied this technique in RETURNS sections to force line breaks for separate elements of a return—we do not follow this convention. For example:

```
CORRECT:       * RETURNS: OK, or ERROR if the tick rate is invalid or the
               * timer cannot be set.
INCORRECT:     * RETURNS: OK, or ERROR
               * if the tick rate is invalid or
               * the timer cannot be set.
```
ERRNO or ERRORS Section

For C routines, you must list any `errno` values set directly by the routine. However, the decision about whether to include or exclude specific `errno` values returned by called routines is left to your discretion. As a general guideline, if you feel that listing the `errno` values set by the called routine improves the usability of the documentation, then list the individual `errno` values returned by the called routine in addition to those set directly by the routine.

Format the list with the item-list tags `\is`, `\i`, and `\ie` (for more information, see *Item Lists (Definition Lists or Terms Lists)*, p. 237.) Tag each error name with the `\i` tag. For example:

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

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

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

SEE ALSO Section

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

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

- Do not include manual section numbers using the UNIX parentheses-plus-number scheme; however, do append parentheses to routine names per the documentation standard (see *Routine Names*, p. 234):

  CORRECT:  SEE ALSO: sysLib, vxTas()

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

- Include cross-references to books by using `apigen`’s `\tb` tag. Each `\tb` tag must appear on a separate line. For more information about this tag, see *Special Words*, p. 233. References to chapters of Wind River manuals should take the form *Publication Title: Chapter Name*. Do not include the chapter number or page number. For example:

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

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

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

### B.4 Sections for target.ref

The target-information reference entry is generated from the file `target.ref`, located in the following directory:

```
installDir/vxworks-6.x/target/config/bspname
```

This file contains board-specific information necessary to run VxWorks. Table B-4 lists the subsections included in a typical `target.ref` file.

<table>
<thead>
<tr>
<th>Section Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>The name of the board</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>Summary of scope and assumptions</td>
</tr>
<tr>
<td>FEATURES</td>
<td>Supported and unsupported features of the board</td>
</tr>
<tr>
<td>HARDWARE DETAILS</td>
<td>Driver and hardware details for the board</td>
</tr>
<tr>
<td>SPECIAL CONSIDERATIONS</td>
<td>Special features or restrictions</td>
</tr>
<tr>
<td>BOARD LAYOUT</td>
<td>The board layout in ASCII format</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>Cross-references to Wind River documentation.</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>References to additional documentation</td>
</tr>
</tbody>
</table>

#### NAME Section

The information in the NAME section should all be on a single line and typed in single quotes as in the following example:

```
mfr&board
```

`mfr&board` stands for the manufacturer’s name plus the manufacturer’s name for the board model, as described earlier. For example:

```
'Motorola MVME2603, MVME2604'
```

#### INTRODUCTION Section

This section includes getting-started information, including subsections detailing ROM installation, boot ROM flash instructions, and jumper settings for VxWorks operation.
IMPLEMENTING DOCUMENTATION GUIDELINES

B.4 Sections for target.ref

FEATURES Section

This section describes all the features of the board. Every feature should be identified under either the Supported Features or Unsupported Features subheadings. Each board configuration option should be considered a feature. A third subheading, Feature Interactions, describes how one feature or board configuration affects others.

HARDWARE DETAILS Section

This section discusses hardware elements and device drivers, such as serial, Ethernet, and SCSI devices. It also includes memory maps for each bus and lists of interrupt levels and/or vector numbers for each interrupting source.

SPECIAL CONSIDERATIONS Section

This section identifies the unique characteristics of the board. It includes all information needed by the user that does not fit in any other section.

For customers that run the BSP validation test suite, this section must also address known failures of tests in the test suite. Presumably the board either does not have a special feature or it implements it in a special manner. The BSP writer is responsible for documenting all exceptions noted during testing of the BSP. For more information on the validation test suite, see the VxWorks BSP Validation Test Suite User’s Guide.

BOARD LAYOUT Section

In this section, include some diagrammatic way of notifying users of the locations of serial and Ethernet connectors, jumpers and switches, the reset button, and other items relevant to getting the board working. The preferred method of providing this information is by using a detailed diagram or picture with the relevant connectors labeled. This is especially important for the console and Ethernet connectors. In cases where the connectors are stacked vertically, be sure to indicate which connector is the console; that is, the uppermost or lowermost D-9 connector. In target.nr files, this type of diagram is provided as an ASCII representation of the board. This is still an acceptable format, although a JPEG image is preferred.

Use the \bs and \be tags to display board diagrams. See the template BSP for guidelines on diagramming jumper positions.

SEE ALSO Section

For VxWorks 5.5 BSPs, this section always references the Setup and Startup chapter of the Tornado User’s Guide (VxWorks 6.x BSPs do not include any automatic references). Other Wind River manuals can be referenced as necessary.
Use the `\tb` tag for titles of manuals. For example:

```
\b SEE ALSO:
\tb Tornado User’s Guide: Establishing Your Environment,
\tb VxWorks BSP Developer’s Guide
```

**BIBLIOGRAPHY Section**

This section references any additional technical manuals, data sheets, or supplements that the user should have at hand. Use the `\tb` tag for these references. (See Table B-5.) For example:

```
\b SEE ALSO:
\tb Motorola PowerPC 603 RISC Microprocessor User’s Manual,
\tb Motorola PowerPC Microprocessor Family: The Programming Environments
```

Note the commas at the ends of the first references in the above examples; the commas are necessary because these references are run together on output. Alternatively, you can separate the references with blank lines to keep each book on a line by itself—this approach is preferable when there are three or more books.

### B.5 Format and Style

This section describes *apigen* markup and text-input conventions. The formatting elements are few and straightforward. To work with *apigen*, source modules and their documentation must be laid out in accordance with a few simple principles.

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

Formatting is controlled by special text *markup*, summarized in Table B-5. Some markup consists of format commands called *tags*, which begin with a backslash and are followed by letters. Some markup elements are *inline*; that is, they can appear anywhere in the line of text. Other markup elements must start in text column 1. Format tags, except `\lib`, always start in column 1. See the following note.

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

<table>
<thead>
<tr>
<th>Location</th>
<th>Markup</th>
<th>Description</th>
<th>Mangen Equiv.</th>
</tr>
</thead>
<tbody>
<tr>
<td>blank line</td>
<td>Paragraph separator.</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>col 1 only</td>
<td>initial spaces</td>
<td>Preserve all spaces and line breaks for this input line.</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table B-5: *Apigen Markup*
<table>
<thead>
<tr>
<th>Location</th>
<th>Markup</th>
<th>Description</th>
<th>Mangen Equiv.</th>
</tr>
</thead>
<tbody>
<tr>
<td>col 1 only</td>
<td>all capital letters</td>
<td>Section heading.</td>
<td>N/A</td>
</tr>
<tr>
<td>col 1 only</td>
<td><code>\&amp;all-caps heading</code></td>
<td>Escape that prevents the special interpretation of an all-caps line as a heading. Suppressed when at the start of an input line, it otherwise generates a plain ampersand.</td>
<td>N/A</td>
</tr>
<tr>
<td>col 1 only</td>
<td><code>\h heading</code></td>
<td>Explicit section heading for use when lowercase characters are required.</td>
<td>N/A</td>
</tr>
<tr>
<td>col 1 only</td>
<td><code>\sh heading</code></td>
<td>Subheading, always mixed case with initial caps.</td>
<td>.SS heading</td>
</tr>
<tr>
<td>inline</td>
<td><code>text</code> or <code>text</code></td>
<td>Bold text, for literal names: filenames, commands, keywords, global variables, structure members, and so on. Text can be multiple words if on the same input line.</td>
<td>same</td>
</tr>
<tr>
<td>inline</td>
<td><code>&lt;text&gt;</code></td>
<td>Italic text, for arguments, placeholders, emphasis, special terms. Text can be multiple words if on the same input line.</td>
<td>same</td>
</tr>
<tr>
<td>inline</td>
<td><code>\lib library</code></td>
<td>Explicit markup for a library name if the name is non-standard (not <code>nameLib</code>, <code>nameDrv</code>, <code>nameShow</code>, <code>nameSio</code>, or <code>if_name</code>).</td>
<td>N/A</td>
</tr>
<tr>
<td>col 1 only</td>
<td><code>\tb booktitle</code></td>
<td>The remainder of the line is a book title reference.</td>
<td>.I, .pG, .tG</td>
</tr>
<tr>
<td>inline</td>
<td><code>&lt;</code> <code>&gt;</code> <code>'</code> <code>'</code></td>
<td>Plain characters <code>&lt;</code>, <code>&gt;</code>, <code>'</code>, and <code>.</code></td>
<td>N/A</td>
</tr>
<tr>
<td>inline</td>
<td><code>\</code> <code>\</code> <code>\</code></td>
<td>Plain character <code>\</code> (backslash).</td>
<td>\e</td>
</tr>
<tr>
<td>col 1 only</td>
<td><code>\cs</code> <code>...</code> <code>\ce</code></td>
<td>Code example or terminal session—preformatted display in fixed-width.</td>
<td>.CS ... .CE</td>
</tr>
<tr>
<td>col 1 only</td>
<td><code>\bs</code> <code>...</code> <code>\be</code></td>
<td>Board diagram—preformatted display in reduced fixed-width.</td>
<td>.BS ... .BE</td>
</tr>
<tr>
<td>col 1 only</td>
<td><code>\ss</code> <code>...</code> <code>\se</code></td>
<td>Syntax display—preformatted display in fixed-width font and containing markup.</td>
<td>.TS ... .TE</td>
</tr>
</tbody>
</table>
### Punctuation and Spelling

#### Quotation Marks

If quotation marks really are necessary in text, always type a straight double-quote (" "). Do not form quotation marks with pairs of single quotes (an old troff convention).

#### Dash

Form a dash by typing two successive hyphens. Do not separate the dash from text with space characters.

#### Headings

Headings must be in all uppercase; `apigen` interprets either of the following types of input as a heading:

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

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

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

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

---

### Table B-5  Apigen Markup (cont’d)

<table>
<thead>
<tr>
<th>Location</th>
<th>Markup</th>
<th>Description</th>
<th>Mangen Equiv.</th>
</tr>
</thead>
<tbody>
<tr>
<td>col 1 only</td>
<td>\is</td>
<td>Item list, also known as a definition list. Each item is a word or phrase followed by an explanation starting on the next line.</td>
<td>.iP or .IP</td>
</tr>
<tr>
<td>col 1 only</td>
<td>\i item \i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>col 1 only</td>
<td>\ms \m mark \me</td>
<td>Numbered or dash list (marker list).[^{a}]</td>
<td>.iP or .IP</td>
</tr>
<tr>
<td>col 1 only</td>
<td>\ts</td>
<td>Table</td>
<td>.TS</td>
</tr>
<tr>
<td>inline</td>
<td></td>
<td>Column delimiter in a table.</td>
<td>N/A</td>
</tr>
<tr>
<td>inline</td>
<td>\</td>
<td>The character</td>
<td>N/A</td>
</tr>
<tr>
<td>col 1 only</td>
<td>&quot;</td>
<td>Comment, ignored by <code>apigen</code>.</td>
<td>N/A</td>
</tr>
</tbody>
</table>

\[^{a}\]: Note that the syntax for starting a marker list was formerly documented as \ml. This value continues to work with current releases of `apigen` but the \ms syntax is preferred and should be used in any new development.
In special cases where such input should not be interpreted as a heading, the words can be preceded with `&`. For example:

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

Occasionally, it is necessary to include lowercase letters in a heading. For such exceptions, use the `\h` tag. For example:

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

Longer reference entries sometimes call for subheadings. Type subheadings in mixed-case on a separate line and apply the `\sh` tag. Capitalize words following standard capitalization practices for mixed-case headings.¹ For example:

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

### Special Words

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

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

- words ending in an empty pair of parentheses (routine names)
- words ending in `.c`, `.h`, `.o`, and `.tcl` (file types)
- words ending in `Lib`, `Drv`, `Show`, or `Sio` (library names)
- words beginning with `if_` (network interface library names)
- words in all uppercase with one or more underscore characters (constants)
- words that begin with `S_` and end with an uppercase string (errno names)

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

Example input:

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

Resulting output:

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

Some library names do not end in the standard suffixes listed above. Flag such names with the `\lib` tag. Note that unlike all other tags, the `\lib` tag can be inline. For example:

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

#### Terminal Keys

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

---

¹ Standard practice: always capitalize the first and last word, and capitalize all other words except articles, prepositions, short conjunctions (fewer than five letters), and the word `to`.
Routine Names

Include parentheses with all routine names. Do not separate the parentheses from the name with a space character (unlike the Wind River convention for code). Do not put a space between the parentheses.

**CORRECT:**
```
taskSpawn()
```

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

Even routines generally construed as VxWorks or host shell “shell commands” must include the parentheses.

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

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

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

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

Placeholder Text

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

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

Resulting output:

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

Parameters

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

```
VOID getName
{
    int    tid, /* task ID */
    char   * pName /* task name */
}
```

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

This routine fetches the name associated with the specified task ID `<tid>` and copies it to `<pName>`.
NOTE: Although C routine parameters are variables from the perspective of the code’s author, they are placeholders from the perspective of the user; therefore we format them as any other placeholder and apply angle brackets. Note, however, that global variables and structure members should be treated as literals, not placeholders; see *Literal Names of Commands, Global Variables, Files, and Other Elements*, p.233.

**Book References**

References to books or book chapters should be tagged with `\tb`, which sets them in italics. For more information about standards for book references, see **SEE ALSO Section**, p.227. Example input:

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

Resulting output:

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

**Cross-References to Other Reference Entries**

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

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

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

**Special Terms**

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

**Emphasis**

In general, avoid applying emphasis to words; a well-cast sentence should be sufficient to convey emphasis. However, if emphasizing a word is necessary, bracket it with the angle brackets (`<` and `>`). In output, these words appear in italics. Never use uppercase to convey emphasis.
Lists and Tables

**NOTE:** Do not use the `\cs` and `\ce` tags to build lists or tables.

**CAUTION:** Nesting of lists is not supported.

### Short Word Lists

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

Example input:

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

Resulting output:

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

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

### Table B-6 Examples of Special Word Formatting

<table>
<thead>
<tr>
<th>Component</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>library in title</td>
<td><code>sysLib.c</code></td>
<td><code>sysLib</code></td>
</tr>
<tr>
<td>library in text</td>
<td><code>sysLib</code></td>
<td><code>sysLib</code></td>
</tr>
<tr>
<td>name with <code>.c</code> extension</td>
<td><code>sysLib.c</code></td>
<td><code>sysLib.c</code></td>
</tr>
<tr>
<td>header file</td>
<td><code>objLib.h</code></td>
<td><code>objLib.h</code></td>
</tr>
<tr>
<td>routine in title</td>
<td><code>sysMemTop</code></td>
<td><code>sysMemTop()</code></td>
</tr>
<tr>
<td>routine in text</td>
<td><code>sysMemTop()</code></td>
<td><code>sysMemTop()</code></td>
</tr>
<tr>
<td>constant, option</td>
<td><code>INCLUDE_SCSI</code></td>
<td><code>INCLUDE_SCSI</code></td>
</tr>
<tr>
<td>other bold elements</td>
<td><code>errno</code></td>
<td><code>errno</code></td>
</tr>
<tr>
<td>placeholder, routine parameter</td>
<td><code>&lt;ptid&gt;</code></td>
<td><code>ptid</code></td>
</tr>
<tr>
<td>emphasis, special terms</td>
<td><code>&lt;must&gt;</code></td>
<td><code>must</code></td>
</tr>
</tbody>
</table>
Item Lists (Definition Lists or Terms Lists)

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

\is
\i 'FIODISKFORMAT'
Formats the entire disk with appropriate hardware track and sector marks. No file system is initialized on the disk by this request.
\i 'FIODISKINIT'
Initializes a DOS file system on the disk volume.
\ie

Resulting output:

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

FIODISKINIT
Initializes a DOS file system on the disk volume.

Marker Lists (Dash or Numbered Lists)

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

Example input:

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

Resulting output:

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

NOTE: The syntax for starting a marker list was formerly documented as \ml. This value continues to work with current releases of apigen but the \ms syntax is preferred and should be used in any new development.
Tables

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

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

Example input:

```
\ts
<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>ampersand</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>\</td>
<td>pipe / bar</td>
<td>bitwise OR</td>
</tr>
<tr>
<td>#</td>
<td>pound sign</td>
<td>bitwise NAND</td>
</tr>
</tbody>
</table>
\te
```

Resulting output:

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>ampersand</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>\</td>
<td>pipe / bar</td>
<td>bitwise OR</td>
</tr>
<tr>
<td>#</td>
<td>pound sign</td>
<td>bitwise NAND</td>
</tr>
</tbody>
</table>

Code Examples, Syntax Displays, and Diagrams

Code Examples

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

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

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

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

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

Resulting output:

```
/* Get file status information */

struct stat statStruct;
fd = open ("file", READ);
status = ioctl (fd, FIOFSTATGET, &statStruct);
```
Because backslashes are not interpreted as an escape in \cs blocks, the backslash itself must not be escaped. For example:

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

Resulting output:

-> copy < DOS1:\subdir\file1

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

Command Syntax

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

Example input:

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

Resulting output:

deflate < infile > outfile

Board Diagrams

Board diagrams are required for the BOARD LAYOUT section of a BSP’s target.ref file.

In VxWorks 5.5, board diagrams are typically provided in plain-text format. Bracket plain-text board diagrams with the \bs and \be tags. Start a diagram with \bs; end it with \be.

For VxWorks 6.x BSPs, you strongly are encouraged to include a JPEG image file in place of the plain-text board diagram (although the plain-text style is still acceptable). If you include an image file, Wind River provides the following guidelines:

- Include your image in a /images directory in your BSP directory.
- Label the location of the console port, network port, and power socket in your image.

**NOTE:** The image notation should include text describing the port as well as a circle (in white or black) surrounding the port or an arrow (in white or black) pointing to the connector.

- If ports are stacked, be sure that your notation includes a description of which port is the relevant one (for example, “serial console (top connector”).
- Do not label jumper locations in your image unless your board is not silk-screened or the silk-screen does not include jumper labels.

To include your JPEG image in the HTML output of the target.ref file, use the \IMAGE apigen directive. For more information, see Graphics, p.240.
NOTE: VxWorks 5.5 users can also include a JPEG image in place of the plain-text diagram as described above. However, the IMAGE directive is not available in refgen, so you must include a text reference to the location of the JPEG file manually.

Graphics

VxWorks 6.x users can insert graphics files using the directive \IMAGE filename. If the output format is HTML, this image file is inserted into an <img> tag. If the output format is anything else, the file is referenced by name in the text. The path of filename must be relative to the directory containing the source file. Currently this directive is used only in BSP target.ref files. Examples:

\IMAGE images/board.jpg
\IMAGE images/switches.gif

B.6 Directives

Directives are apigen controls for special non-formatting actions, such as including information from other files, hiding internal information, or overriding default behavior. This section provides information about the apigen directives available for BSP documentation.

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

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

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

The directives recognized by apigen are summarized in Table B-7. Details are provided in the sections that follow.

Table B-7 Summary of apigen Directives

<table>
<thead>
<tr>
<th>Directive Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>\IMAGE filename</td>
<td>Include an image (a reference to an image file) in the output—normally used only for BSP board diagrams. This directive is available for apigen only.</td>
</tr>
</tbody>
</table>
| \INTERNAL [title]    | Do not print the following title and section unless the -internal flag is specified when apigen is executed. If no title is given, the title is “INTERNAL”.

|

240
Blocking Text from Publication

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

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

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

Examples:

\INTERNAL
\INTERNAL IMPLEMENTATION DETAILS

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

Use of this directive in a library section is rare. Putting \NOMANUAL in a library section does not make its routines internal. The standard way to prevent an entire file from being processed is to omit it from a makefile variable.

Example:

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

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

\TITLE name - shortdescription
The \TITLE directive replaces a file’s title line (the title in the file’s first comment block) with whatever follows the directive. Currently it is used exclusively in BSPs to give the file a more descriptive name than target.ref. The new name is used in searching for possible hyperlinks to the output file, but does not affect the name of the output file. Example:
\TITLE pc386/486 - BSP for PC 386/486

\NOTE: This directive must include the backslash in both \apigen and \refgen.

Image Files

The \IMAGE directive provides a means for including an image from another file.

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

\NOTE: This directive is available in \apigen only.

B.7 Converting target.nr Files

Beginning with VxWorks 6.0, the target-information file has a new name and a new format. The file target.nr has been renamed target.ref and is formatted with \apigen markup language instead of \nroff/\troff markup.

In a UNIX installation of VxWorks, you can convert existing target.nr files by running the shell script mg2ref as follows:
% /host/host/bin/mg2ref target.nr
This generates a target.ref file in the current directory.

In a Windows installation of VxWorks, you can convert existing target.nr files by running the Tcl script mg2ref.tcl as follows:
c:\> host\x86-win32\bin\mg2ref.bat target.nr
This generates a target.ref file in the current directory.

The resulting target.ref files require some cleanup, however minimal. Tables always require that columns be manually aligned using the column delimiter (pipe character, |). If the target.nr included nroff-style column format commands, these are not removed and must be removed manually.
In addition, `mg2ref` also leaves a `\TITLE` directive just below the initial comment block. Both `apigen` and `refgen` create the NAME line based on information from the `\TITLE` directive, if it exists, or information from line 1 of the file, if `\TITLE` does not exist. A manually typed NAME section is never used.

Always inspect the `target.ref` generated by `mg2ref` and test your changes by running it through `apigen`.

For backward compatibility, the makefile system for generating VxWorks 6.x BSP documentation is set up to look first for `target.ref`; if `target.ref` is not found, it instead processes `target.nr`.

### B.8 Generating Reference Entries

This section discusses the mechanics of generating BSP documentation: the files and tools used, the text formatting tags, and the makefile system used to process documentation from source code to printable reference entries.

#### Files

Filename extensions indicate the following types of files:

- `.s` assembly-language source files.
- `.c` C-language source files.
- `.ref` text file containing `apigen` markup.
- `.nr` text file containing `mangen` markup (no longer used)
- `.html` generated HTML file.

```
installDir/vxworks-6.x/target/config/bspname
```

This directory contains the C and assembly sources and documentation sources for a particular BSP; `bspname` is a directory name reflecting the maker and model of the board. For example: `mv2603` = Motorola MVME2603. The files relevant to documentation are:

- **Makefile**
  Master makefile for building BSP and VxWorks modules. Three constants must be defined for documentation: `TARGET_DIR`, `VENDOR`, and `BOARD`. See `Processing`, p.244, for more information.

- **sysLib.c**
  Library of board-dependent C routines.

- **target.ref**
  Source for the target-information reference entry, containing general information about a board’s installation requirements or capabilities as they relate to VxWorks. Note that this file replaces the `target.nr` file of previous releases; however, BSP makefiles beginning with VxWorks 5.5 can also recognize and process `target.nr` files for backward compatibility. If both files exist, the make system gives precedence to `target.ref`.
The generated HTML reference-entry files for BSPs are output to the
`docs/extensions/eclipse/plugins/com.windriver.ide.doc.vxworks_6.x/vx
works_bsp_api_reference_6.x/bspname` directory for VxWorks 6.x (or the
docs/vxworks/bsp/bspname directory in VxWorks 5.5). All files are
generated by the make process from the source files in:

```
installDir/vxworks-6.x/target/config/bspname
```

Which runs `apigen` (or `refgen -mg` for VxWorks 5.5).

The files are:

- `bspname.html`
  Target-information reference entry generated from `target.ref` (or
  `target.nr`). As shown in the following example, you must include
  `/target.ref` after `bspname` to generate this file correctly:

  ```
  \" `bspname/target.ref - BSP target specific documentation
  ```

- `sysLib.html`
  Reference entry for `sysLib.c`.

- `libIndex.html`
  Index of the BSP’s library-level reference entries.

- `rtnIndex.html`
  Index of the BSP’s routine reference entries.

- `sysTffs.html` (optional, VxWorks 6.x only)
  Reference entries for TrueFFS, if installed.

### Tools

```
host/host/bin/apigen
```

The `apigen` tool is a Perl script (`refgen` is a Tcl script) that generates HTML files
from specially formatted source code, which may be C language modules,
assembly language modules, `target.ref` files, or `target.nr` files. The
command-line syntax and options for `apigen` are summarized in the reference
documentation.

### Processing

The steps below describe how to generate BSP documentation using the makefiles
based on the templates delivered with the VxWorks.

1. In `installDir/vxworks-6.x/target/config/bspname`, check for the existence of
   the three required constants in `Makefile`:

   ```
   TARGET_DIR
   target directory name (`bspname`). For example: “mv2603”
   VENDOR
   vendor name. For example: “Motorola”
   BOARD
   board model name. For example: “MVME2600”
   ```
2. Generate the reference entries by running the following command in the BSP directory:

```
make man
```

This does the following:
- Builds an appropriate `depend.bspname`.
- Runs `sysLib.c` through the C preprocessor to collect any drivers included by `#include` directives.
- Runs `sysTffs.c` through the C preprocessor (if applicable).
- Runs `apigen` (or `refgen -mg`) on `sysLib.c` (output of the previous step) and `target.ref` (or `target.nr` if no `target.ref` is present).

```
NOTE: refgen requires the -mg option for backward compatibility in case there are files with old-style nroff markup.
```
- Distributes HTML reference entries to the appropriate `docs` directories.

The flow chart in Figure B-1 shows how the make process distributes BSP reference entries in the `docs` directory.

---

**Figure B-1 Production Flow for BSP Documentation**

```
target/config/bspname

[Diagram: Flowchart showing the process from `target/config/bspname` to `docs/bspname`]
```

```target/config/bspname

| target.ref |
| sysLib.c |
| sysTffs.c |
```

```
docs/bspname

| bspname.html |
| sysLib.html |
| libIndex.html |
| rtnIndex.html |
| sysTffs.html |
```
C.1 Introduction

In a VxWorks AMP configuration, a hardware device, such as an interrupt controller, may need to be shared by all CPUs. To make this possible, the VxWorks image on each CPU must contain its own copy of the driver for the device, and the driver must be implemented so that its activity on one CPU does not interfere with or corrupt the state of the driver on another CPU. In the current release, this is only possible for devices that provide a separate set of registers for each CPU (see C.4 Run-Time Device Sharing, p.249).
C.2 Hardware Device Sharing

Figure C-1 illustrates hardware device sharing for an interrupt controller.

In Figure C-1, each CPU has a copy of the device driver for the interrupt controller and access to its own register set. At startup, CPU0 initializes all register sets in the interrupt controller (see C.3 Boot-Time Device Sharing, p.248). After startup, each CPU accesses its own register set in the interrupt controller (see C.4 Run-Time Device Sharing, p.249).

C.3 Boot-Time Device Sharing

To prevent interference between device drivers at boot time, the primary CPU needs to be responsible for initializing any shared hardware. The primary CPU is always the first CPU to boot and, at this point, should have full access to any shared devices, since no other AMP CPUs are running. The code in the following example tests this condition. It is taken from the initialization code for an AMP-capable interrupt controller driver.

```
LOCAL void <driver>InstInit2
(  
    VXBPDEVICEID pInst
 )
{
    unsigned int cpuid;
    /* early exit if not CPU 0 */
    if (vxcpuIdGet() != 0)
        return;

    /*
    * Walk through all CPUs, disable all interrupts. Note that
    * on this CPU, CPU's ID's are equivalent to CPU indexes.
    */
```
for (cpuid = 0; cpuid < MAX_NUM_CORES; cpunum++)
{
  disableCpuInterupts(cpuid);
}

In the example, the code first tests to see whether it is running on CPU 0. If it is, the
driver disables interrupts for all other CPUs that could have connected to the
interrupt controller. When a driver on a secondary CPU executes the same code,
the code will simply return, since initialization has already been performed by the
primary CPU.

C.4 Run-Time Device Sharing

Once an AMP system is fully booted, the device driver for a shared device must
make sure that its CPU does not attempt to access the device at the same time as
another CPU does. To code for this, one approach might be to synchronize accesses
using spinlocks. However, spinlocks are not supported in the current release of
VxWorks AMP. The only currently available approach requires that a shared
device provide a separate set of registers for each CPU in the system. This is typical
of interrupt controller devices. The driver for such a device can ensure that it only
accesses its own registers by first querying for its CPU ID, and then using the ID
number to determine which register-set within the device to use. The following is
an example:

LOCAL void <drive>IntDemux
{
  int intr_line  /* line that caused the interrupt */
}
{  
mbox_t m;

  union {
    UINT64 all;
    UINT32 half[2];
  } intSourceEn0, intSourceEn1, intSource;

  unsigned int id = vxCpuIdGet();

  /* read the interrupt registers and the status registers
   * mask the status register with the enabled register to
   * get the valid interrupt(s)
   */

  intSourceEn0.all = <device>_read64(<device>_INTX_SUM0(id)) &
                    <device>_read64(<device>_INTX_EN0(id));
  intSourceEn1.all = <device>_read64(<device>_INTX_SUM1(id)) &
                    <device>_read64(<device>_INTX_EN1(id));

  /* ... */
D.1 Introduction

BSPs for 64-bit systems must implement `sysMemDescInit()` and `sysMemDescGet()` routines to describe the physical address map of the hardware to the kernel. For information in this regard, see 9. Migrating BSPs from 32-Bit to 64-Bit. For code examples, see D.2 Example sysMemDescInit() Code, p.252 and D.3 Example sysMemDescGet() Code, p.253.
Example sysMemDescInit( ) Code

```c
/*
* sysPhysRamDesc contains one region of RAM that must be mapped at
* LOCAL_MEM_LOCAL_ADRS, and another region that can be mapped at the
* discretion of the kernel. The 2nd entry is only used if the BSP
* discovers a 2nd RAM block during system boot.
*/
LOCAL PHYS_MEM_DESC sysPhysRamDesc [] =
{
    {
        (VIRT_ADDR) LOCAL_MEM_LOCAL_ADRS,
        (PHYS_ADDR) LOCAL_MEM_PHYS_ADRS,
        LOCAL_MEM_SIZE, /* may be modified at runtime */
        BSP_RAM_ATTRIBUTES_MASK,
        BSP_RAM_ATTRIBUTES
    },
    /* if BSP discovers a 2nd RAM block, this entry is used */
    {
        (VIRT_ADDR) MEM_DESC_ADDR_KERNEL_ASSIGNED,
        (PHYS_ADDR) 0,
        0,
        BSP_RAM_ATTRIBUTES_MASK,
        BSP_RAM_ATTRIBUTES
    }
};
LOCAL sysPhysMemDescCount = NELEMENTS(sysPhysMemDesc);

/*
* sysDma32MemDesc is used to provide 32-bit DMA memory
* if INCLUDE_CACHE_DMA32_LIB is defined.
*/
LOCAL PHYS_ADDR sysDma32MemDesc[] =
{
    /* if 32-bit DMA is needed, this entry is used */
    {
        (VIRT_ADDR) MEM_DESC_ADDR_KERNEL_ASSIGNED,
        (PHYS_ADDR) 0, /* computed at runtime */
        0 /* computed at runtime */
        BSP_DMA_ATTRIBUTES_MASK,
        BSP_DMA_ATTRIBUTES
    }
};
LOCAL sysDma32MemDescCount = NELEMENTS(sysDma32MemDesc);

#ifdef INCLUDE_USER_RESERVED_MEMORY
/*
* sysUserMemDesc is used to provide user reserved memory
* if INCLUDE_USER_RESERVED_MEMORY is defined.
*/
LOCAL PHYS_MEM_DESC sysUserMemDesc[] =
{
    {
        (VIRT_ADDR) MEM_DESC_ADDR_KERNEL_ASSIGNED,
        (PHYS_ADDR) 0, /* computed at runtime */
        0, /* computed at runtime */
        BSP_RAM_ATTRIBUTES_MASK,
        BSP_RAM_ATTRIBUTES
    }
};
LOCAL int sysUserMemDescCount = NELEMENTS(sysUserMemDesc);
#endif /* INCLUDE_USER_RESERVED_MEMORY */

/*
* sysPmMemDesc is used to provide persistent memory
* if INCLUDE_EDR_PM is defined.
*/
LOCAL PHYS_ADDR sysPmMemDesc[] =
{
```
D.3 Example sysMemDescGet() Code

/*********************************************************************************/
/* sysMemDescInit - Perform initialization required by sysMemDescGet(). */
/* */
/* This routine is used during system boot to perform any initialization */
/* required by the BSP so that sysMemDescGet() can return correct */
/* information about the memory configuration. */
/* */
/* RETURNS: N/A. */
/* */
STATUS sysMemDescInit(void) {
    /* Patch the descriptors based upon run-time tests of the */
    /* various memory segments. The details of how the memory */
    /* sizing tests are performed is outside the scope of this */
    /* document. */
    /* */
    size_t ramSize = sysRamSizeProbe(sysPhysRamDesc[0].physicalAddr);
    size_t dma32Size = DMA32_RESERVED_MEM;
    size_t pmSize = PM_RESERVED_MEM;
    #ifdef INCLUDE_USER_RESERVED_MEMORY
        size_t userSize = USER_RESERVED_MEM;
    #else
        size_t userSize = 0;
    #endif /* INCLUDE_USER_RESERVED_MEMORY */
    /* shrink 0th segment by reserved sizes */
    /* */
    sysPhysRamDesc[0].len = ramSize - dma32Size - userSize - pmSize;
    /* set physical address and size for 32-bit DMA */
    sysDma32MemDesc[0].len = dma32Size;
    sysDma32MemDesc[0].physicalAddr =
        (sysPhysRamDesc[0].physicalAddr + sysPhysRamDesc[0].len);
    #ifdef INCLUDE_USER_RESERVED_MEMORY
        /* set physical address and size for user reserved memory */
        /* */
        sysUserMemDesc[0].len = userSize;
        sysUserMemDesc[0].physicalAddr =
            (sysDma32MemDesc[0].physicalAddr + sysDma32MemDesc[0].len);
    #endif /* INCLUDE_USER_RESERVED_MEMORY */
    /* set physical address and size for persistent memory */
    /* */
    sysPmMemDesc[0].len = pmSize;
    #ifdef INCLUDE_USER_RESERVED_MEMORY
        sysPmMemDesc[0].physicalAddr =
            /* */
    #endif /* INCLUDE_USER_RESERVED_MEMORY */
}
(sysUserMemDesc[0].physicalAddr + sysUserMemDesc[0].len);
#else
  sysPmMemDesc[0].physicalAddr =
        (sysDma32MemDesc[0].physicalAddr + sysDma32MemDesc[0].len);
#endif /* INCLUDE_USER_RESERVED_MEMORY */

/*
* This board might have a 2nd memory segment, so test
* for it, and if present, describe it in the second
* RAM descriptor
*/

if (sysRamSize2ndBlockPresent())
{
  sysPhysRamDesc[1].virtualAddr = sysRamSize2ndBlockPhysAddr();
  sysPhysRamDesc[1].ramSize = sysRamSize2ndBlockSize();
}

/******************************************************************************/
/* sysMemDescGet - Return memory descriptors describing memory layout.       */
/*                                                                         */
/* This routine is used during system boot to describe the memory          */
/* layout to the kernel and included components. It supports               */
/* memory autoconfiguration. It also supports carving memory from          */
/* the first contiguous block of RAM for 32-bit DMA and persistent         */
/* memory. This routine assumes that enough memory is provided             */
/* in the 0th descriptor to support all of these segments.                 */
/* NOTE - To simplify the example, this routine presumes that              */
/* LOCAL_MEM_AUTOSIZE is defined. A proper implementation                  */
/* would include appropriate ifdef's to exclude the autoconfig             */
/* code when it is not needed.                                             */
/* RETURNS: OK, if a valid PHYS_MEM_DESC is copied to the caller.          */
/******************************************************************************/
STATUS sysMemDescGet
{
  enum memDescType, /* type of memory being queried */
  int index, /* index of memory block for the type */
  PHYS_MEM_DESC *pMemDesc /* to return the description */

  PHYS_MEM_DESC *pDesc;
  int descCount;

  /*
  * Provide the correct array and size
  * parameters based upon the query type provided by the caller.
  *
  */

  if (memDescType == MEM_DESC_RAM)
  {
    pDesc = sysPhysMemDesc;
    descCount = sysPhysMemDescCount;
  }
  #ifdef INCLUDE_USER_RESERVED_MEMORY
  else if (memDescType == MEM_DESC_USER_RESERVED_RAM)
  {
    pDesc = sysUserMemDesc;
    descCount = sysUserMemDescCount;
  }
  #endif /* INCLUDE_USER_RESERVED_MEMORY */
  else if (memDescType == MEM_DESC_PM_RAM)
  {
    pDesc = sysPmMemDesc;
    descCount = sysPmMemDescCount;
  }
else if (memDescType == MEM_DESC_DMA32_RAM)
{
    pDesc = sysDma32MemDesc;
    descCount = sysDma32MemDescCount;
}
else
{
    return ERROR; /* no support for MEM_DESC_USER_RESERVED_RAM in this BSP */
}

if (index >= descCount) /* no descriptor at requested index */
    return ERROR;

/*
 * copy the descriptor. Note that it's OK to return a descriptor that
 * has a length of 0. This allows the BSP to allocate "empty" entries
 * in the descriptor arrays, and fill them in at run-time without having
 * to alter the logic of this routine.
 */

*pMemDesc = pDesc[index];
return OK;
}
### Index

#### Symbols

<table>
<thead>
<tr>
<th>Symbol/Caption</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ARCH_MULTIPLE_CACHELIB</td>
<td>76</td>
</tr>
<tr>
<td>_romInit( )</td>
<td>7</td>
</tr>
<tr>
<td>_sysInit( )</td>
<td>8, 22</td>
</tr>
<tr>
<td>_wrload_bias_high_32</td>
<td>175</td>
</tr>
<tr>
<td>_wrload_bias_low_32</td>
<td>175</td>
</tr>
<tr>
<td>_WRS_ARCH_SUPPORTS_VX_CPU_STATE_INIT</td>
<td>138</td>
</tr>
<tr>
<td>_WRS_CONFIG_SMP</td>
<td>125, 126</td>
</tr>
</tbody>
</table>

#### Numerics

<table>
<thead>
<tr>
<th>Name</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10wrload.cdf</td>
<td>176</td>
</tr>
<tr>
<td>20bsp.cdf</td>
<td>126</td>
</tr>
<tr>
<td>AMP</td>
<td>161, 179</td>
</tr>
<tr>
<td>small-footprint</td>
<td>195</td>
</tr>
<tr>
<td>22comp_mipc.cdf</td>
<td>187</td>
</tr>
</tbody>
</table>

#### A

- adding
  - Ethernet devices 97
  - other timers 72
  - support for MIPC 185
  - support for the wrload utility 170
  - VxBus devices 96
- address space library 80
- adrspaceLib 80
- AIM cache library 77
- AIM MMU library 77
- AIM_MMU_INIT_PT_INCR 92
- AIM_MMU_INIT_PT_NUM 92
- AIM_MMU_INIT_RT_INCR 92
- AIM_MMU_INIT_RT_NUM 92
- aimCacheInit( ) 77
- aimCacheLib 77
- aimMmuLib 77
- aimMmuLibInit( ) 78
- allocating
  - address space for AMP 177
  - address space for MIPC shared memory 185
  - AMP 149
  - 20bsp.cdf 161, 179
  - adding
    - support for MIPC 185
  - allocating
    - address space 177
    - address space for MIPC shared memory 185
  - assigning devices 181
  - boot loader strategy 155, 162, 163
  - booting
    - a secondary CPU 163
    - the primary CPU 162
    - the secondary image 165
  - bootLineParse( ) 159
  - BSP
    - development overview 150
    - documentation 159
    - implementation overview 154
    - testing and validation 160
  - BSP VTS restrictions and limitations 160
  - build-time (static) configuration of devices 183
  - CDF 179
  - configuring
    - exception vectors 156
    - memory 156
  - creating an initial BSP 155, 161
  - custom VxWorks components for 153
  - development prerequisites 151
  - device allocation 157
  - exception and interrupt handling 157
  - hardWareInterFaceInit( ) 183
  - hcfDeviceList[ ] 181
  - holding the secondary CPU 163, 169
  - hwconf.c 181
  - initializing
    - secondary image exception vectors 166
    - secondary image memory 165
  - LOCAL_MEM_LOCAL_ADRS 156, 165, 166
  - LOCAL_MEM_SIZE 156, 165, 166
  - Makefile 161
  - managing interrupts 167
Index

sysLb.c 23
target.ref 23
usrConfig.c 24
finalizing 71
getting a minimal kernel running 56
hardware considerations 44
macros 41
BSP_REV 42
BSP_VERSION 42
LOCAL_MEM_SIZE 43
RAM_HIGH_ADDRS 42
optional routines 44
reference entries, writing 215
required macros 41
routines 32
optional 44
required routines 32
sysBspRev() 33
sysClkConnect() 33
sysClkDisable() 33
sysClkEnable() 33
sysClkInt() 33
sysClkRateGet() 33
sysClkRateSet() 33
sysHwInit() 39
sysHwInit2() 39
sysMemDescGet() 35
sysMemDescInit() 34
sysMemTop() 36
sysModel() 38
sysNvRamGet() 38
sysNvRamSet() 38
sysSerialChanGet() 39
sysSerialHwInit() 39
sysSerialHwInit2() 39
sysToMonitor() 40
support
for optional devices 96
for VxWorks 95
ti_omap1137evm 193
time required for development 3
Bsp data structure 102, 104
Bsp statement 127
Bsp VTS
building and running small-footprint tests 195
restrictions and limitations for AMP 160
BSP_REV 42
BSP_VER_1_1 42
BSP_VER_1_2 42
BSP_VERSION 42
bspname.h 20, 21
building
a VxWorks image 67
SMP BSPs 147
bus support
PCI 95
USB 96
VxBus 95

C

cache
cache locking 84
CDF configuration 85
common problems 211
defining
buffers for cache read and flush 82
cache configuration variables 82
the configuration 85
enabling 73
implementing
L2 cache support 83
L3 cache support 84
initializing
cache configuration variables 82
VxWorks support layers 76
managing during reset and initialization 74
modifying handling in SMP 134
module hierarchy 77
providing architecture-specific support 82
required handling
romInit() 75
sysInit() 75
sysToMonitor() 75
small-footprint considerations 192
supporting L2 and L3 cache 83
timing issues 212
CACHE_LIB_INIT_FUNC 88
cacheArchLib.c 83, 84
cacheArchLibInit() 76
cacheClear() 134
CACHECONFIG 77
cacheDisable() 134
cacheEnable() 134
cacheFlush() 134
cacheInvalid() 134
cacheLib 76
cacheLibInit() 76
cacheLock() 135
cacheMipsLib 88
cacheMipsLibInit 88
cachePpcReadOrigin 82
caches
supporting cache coherency in SMP 135
cacheType 84
cacheUnlock() 135
CDF
22comp_mipc.cdf 187
adding code for SMP 126
AMP 153, 179
Bsp statement 127
configuration for cache and MMU 85
PROFILE_BOOTAPP 102
common problems
cache 211
contlicts between virtual and physical memory 213
debug tools 211
exception handling 211
MMU 211
reusing code 212
volatile variables 212
common.vxconfig 191
compiler choosing 49
component and configuration preprocessor macros for SMP 125
COMPONENT_VXTESTV2_PSF 194
components custom components for AMP 153
INCLUDE_AIM_MMU_CONFIG 91
INCLUDE_AIM_MMU_MEM_POOL_CONFIG 92
INCLUDE_AIM_MMU_PT_PROTECTION 92
INCLUDE_AMP 154
INCLUDE_AMP_CPU_00 153
INCLUDE_AMP_CPU_nn 154
INCLUDE_BOOT_APP 104
INCLUDE_BOOT_FS_DEVICES 102
INCLUDE_BOOT_MMU_BASIC 90
INCLUDE_BOOT_NET_DEVICES 102
INCLUDE_BOOT_RAM_IMAGE 103
INCLUDE_CACHE_ARCH 87
INCLUDE_CACHE_BSP 87
INCLUDE_CACHE_ENABLE 86
INCLUDE_CACHE_L2 87
INCLUDE_CACHE_L2_ENABLE 88
INCLUDE_CACHE_SUPPORT 85
INCLUDE_DEBUG_KPRINTF 146
INCLUDE_DEBUG_KWRITE_USER 146
INCLUDE_MAPPED_KERNEL 91
INCLUDE_MIPC_UNSUPPORTED 187
INCLUDE_MMU_ARCH 93
INCLUDE_MMU_ARCH_ARM1136 93
INCLUDE_MMU_ARCH_PPC60X 93
INCLUDE_MMU_BASIC 89
INCLUDE_MMU_BSP 93
INCLUDE_PAGE_SIZE_OPTIMIZATION 90
INCLUDE_PROTECT_IDLE_TASK_STACK 128
INCLUDE_SHOW_ROUTINES 174
INCLUDE_WRLOAD 153
config.h 21
BSP_REV 42
BSP_VERSION 42
LOCAL_MEM_SIZE 43
macros 41
modifications for small-footprint 191
RAM_HIGH_ADRS 42
configAll.h 24, 31
configuring exception vectors for AMP 156
controlling access to devices with ifdef 184
Copying a BSP 125
CPC, see cross-processor call
CPU macro 41
CPU ID 135
CPU index 135, 136

see also CPU number, cpuNum
CPU interrupt controller inputs 143
CPU number 135
see also CPU index
CPU_VARIANT 41
CPU-generated interrupts 168
cpuIndex 137, 138, 139
see also CPU index
cpuNum 135
see also CPU number
creating a BSP for SMP 125
additional diagnostic routines 62
cross-processor call 133

data address in virtual memory environments 99
data coherency problems 99
data structures Bsp 102, 104
data variables required 25
DATASEGPAD 19
dataSegPad.c 19
DEBUG_KWRITE_USR_RTN 147
debugging 46 advanced techniques 115
basic techniques 108
breakpoints 116
forever loop 118
ICE 48
initialization code 58
interrupts 65
logic analyzer 47
on-chip debugging (OCD) 47
oscilloscope 47
ROM emulator 46
ROM monitor 46
ROM-based image initialization code 61
small-footprint BSPs 194
SMP 146
system-mode debugging techniques 66
tools 211
using LEDs 46, 108
using NVRAM 46
VxBus device drivers 97
WDB agent 48
with LEDs 112
defining buffers for cache read and flush 82
cache and MMU configuration 85
cache configuration variables 82
derived files
target.ref 25
VxWorks images 25
determining the number of physical CPUs available on a
processor 139
development environment
common problems 209
introduction 45
device allocation
for VxWorks AMP 157
device drivers
debugging VxBus drivers 97
interrupt controller 68
registration with VxBus 60
VxBus initialization sequence 60
devices
assigning for AMP 181
configuring at build-time for AMP 183
configuring at runtime for AMP 182
diagnostic routines 62
direct memory access, see DMA
 disabling secondary CPUs 139
DMA 68, 99
document terminology 3
documentation 23
guidelines 215
make man 25
requirements for VxWorks AMP 159
documenting your BSP 23
download path 50
downloading images 50, 67
driver resource table 145
drvIndex 143
drvUnit 143

E

EBASE 166
ENABLE_ALL_CPUS 127
enabling
  cache 73
  MMU 73
  NVRAM 72
endianness configurations 13
entry point for secondary CPUs 139
Ethernet devices
adding 97
MAC address 98
examples
  small-footprint 193
EXC_MSG_ADRS 31
exception vectors
configuring for AMP 156
initializing secondary image 166
executing
as a task 18
kernelInit() 17
romInit() 14
romStart() 15
sysInit() 15
usrInit() 16
usrRoot() 18

F

ffsMsb 134
files
  bootInit.c 24
  bspname.h 21
  common.vxconfig 191
  config.h 21
  configAll.h 24
  hex 68, 109
  hwconf.c 24
  Makefile 21
  README 20
  romInit.s 22
  S-record 68
  sysALib.s 22
  sysDev.c 23
  sysLib.c 23
  target.ref 23
  usrConfig.c 24, 31
  vmLib.h 27
finalizing your BSP 71
finding
  addresses in the flash image 109
  code block addresses 109
  symbol addresses 109
forever loop 118

G
global interrupt controller inputs 144

H

hardware configuration device list 145
  hardWareInterFaceInit() 70
  AMP 183
  hcfDeviceList 31
  hcfDeviceList[ ] 141, 145, 181
  hcfDeviceNum 26, 31
  hcfResource 145
header files
  bspname.h 21
  config.h 21
  configAll.h 24
  vmLib.h 27
  HEX_FLAGS 42
holding a secondary CPU on startup or reset 131, 163, 169
hwconf.c 141
  AMP 181
  modifications for small-footprint 191
VxBus 24
HWMEM_POOL_SIZE 146

I

I/O device configuration 145
IA-32
sysCacheLibInit 76
ICE 48
identifying CPUs in SMP 135
IDLE_TASK_EXC_STACK_OVERFLOW_SIZE 128
IDLE_TASK_EXC_STACK_UNDERFLOW_SIZE 128
IDLE_TASK_EXCEPTION_STACK_SIZE 128
image types
bootrom 12
bootrom_res 12
bootrom_res_high 13
bootrom_uncomp 12
vxWorks 11
vxWorks.res_rom 12
vxWorks.res_rom_nosym 12
vxWorks.res_rom_nosym.res_low 12
vxWorks.res_rom.res_low 12
vxWorks.st 12
vxWorks.st_rom 12
vxWorks.sym 11
vxWorks_rom 11
implementing
boot logic for SMP 129
L2 cache support 83
L3 cache support 84
in-circuit emulator, see ICE
include files 19
INCLUDE_AIM_MMU_CONFIG 91
INCLUDE_AIM_MMU_MEM_POOL_CONFIG 92
INCLUDE_AIM_MMU_PT_PROTECTION 92
INCLUDE_AMP 154
INCLUDE_AMP_CPU_00 153
INCLUDE_AMP_CPU_nn 154, 179
INCLUDE_BOOT_APP 104
INCLUDE_BOOT_FS_DEVICES 102
INCLUDE_BOOT_MPU_BASIC 90
INCLUDE_BOOT_NET_DEVICES 102
INCLUDE_BOOT_RAM_IMAGE 103
INCLUDE_CACHE_ARCH 87
INCLUDE_CACHE_BSP 87
INCLUDE_CACHE_ENABLE 86
INCLUDE_CACHE_L2 87
INCLUDE_CACHE_L2_ENABLE 88
INCLUDE_CACHE_SUPPORT 85, 193
INCLUDE_DEBUG_KPRINTF 146
INCLUDE_DEBUG_KWRITE_USER 146
INCLUDE_L2_CACHE 88
INCLUDE_L3_CACHE 88
INCLUDE_MAPPED_KERNEL 91
INCLUDE_MIPC_UNSUPPORTED 187
INCLUDE_MMU 193
INCLUDE_MMU_ARCH 93
INCLUDE_MMU_ARCH_ARM1136 93
INCLUDE_MMU_ARCH_PPC60X 93
INCLUDE_MMU_BASIC 89
INCLUDE_MMU_BSP 93
INCLUDE_MMU_MPU 91
INCLUDE_MMU_P6_32BIT 93
INCLUDE_MMU_P6_36BIT 93
INCLUDE_PAGE_SIZE_OPTIMIZATION 90
INCLUDE_PROTECT_IDLE_TASK_STACK 128
INCLUDE_SHOW ROUTINES 174
INCLUDE_SMP_DEMO 126
INCLUDE_WRLOAD 153
initialization
generic 64
RAM 64
VxBus routines 70
initialization code
debugging 58
initialization sequence
VxBus 60
initializing
cache configuration variables 82
secondary image exception vectors 166
secondary image memory 165
the WIND_CPU_STATE structure 137
Intel 64
sysCacheLibInit 76
interprocessor interrupts
preassigned interrupt levels in SMP 133
prioritizing in SMP 133
interrupt controller 68
CPU routing for SMP 142
cross bar 143
resources 144
interrupt controller drivers
configuring for SMP 141
interrupt handling
for VxWorks AMP 157
interrupt signal routing for SMP 141
interrupts
CPU-generated 168
finding the source of 65
message-signalled 169
PCI-device on AMP 168
PLB device 168
supported for AMP 168
unsupported for AMP 168
intPrioTable 134
IPFs, see interprocessor interrupts
IPVR 166

J

JTAG 47
managing during reset and initialization 74
module hierarchy 78
providing architecture-specific support 82, 83
required handling
romInit() 75
sysHwInit() 75
sysInit() 75
small-footprint considerations 192
mmuADisable() 193
mmuLib 192
MP_OPTIONS 127

N
NetROM 50, 68
non-maskable interrupts (NMI) 53, 67
non-volatile RAM
debugging with 46
enabling 72
NUM_TTY 43
nvRamToFlash.c
sysFlashGet() 73
sysFlashSet() 73

O
objdump 108
OCD 47, 115
on-chip debugging (OCD), see OCD
optional devices
support for 96
troubleshooting and debugging 99
OS image 10
outConsole() 62

P
page table entry
AMP 166
parameter
WRLoad_IMAGE_BUILD_PHYS_BIAS 175
parameters
AIM_MMU_INIT_PT_INCR 92
AIM_MMU_INIT_PT_NUM 92
AIM_MMU_INIT_RT_INCR 92
AIM_MMU_INIT_RT_NUM 92
BOOT_DISABLE_MMU_BEFORE_ENTRY 90
CACHE_LIB_INIT_FUNC 88
ENABLE_ALL_CPUS 127
IDLE_TASK Exc_STACK_OVERFLOW_SIZE 12
8
IDLE_TASK_Exc_STACK_UNDERFLOW_SIZE 128
IDLE_TASK_EXCEPTION_STACK_SIZE 128
SW_MMU_ENABLE 91
USER_D_CACHE_ENABLE 86
USER_D_CACHE_MODE 86
USER_D_MPU_ENABLE 90
USER_I_CACHE_ENABLE 86
USER_I_CACHE_MODE 86
USER_I_MPU_ENABLE 90
USER_L2_CACHE_ENABLE 88
VM_PAGE_SIZE 89
VX_ENABLE_CPU_TIMEOUT 128
VX_SMP_NUM_CPUS 127
WRLoad_CPU_ENABLE_FUNC 177
WRLoad_CPU_MAP_FUNC 175
WRLoad_CPU_MEM_MAP_FUNC 174
WRLoad_CPU_MEM_READ_FUNC 175
WRLoad_CPU_MEM_UNMAP_FUNC 174
WRLoad_CPU_PREPARE_FUNC 177
WRLoad_CPU_UNMAP_FUNC 175
WRLoad_IMAGE_BUILD_PHYS_BIAS 174
PCI
AMP 168
bus support 95
PHYS_MEM_DESC 26, 78
physical CPU identification 135
see also CPU ID
physical CPU index 136
returning the value of 136
physical memory descriptor table
defining and initializing 77
sysPhysMemDesc[] 78
pIdleStackBase 137, 138
PLB device interrupts 168
position-independent code (PIC) 210
post-processed compiler output 113
PowerPC
AMP 166, 170
INCLUDE_CACHE_L2 87
INCLUDE_CACHE_L2_ENABLE 88
INCLUDE_L2_CACHE 88
INCLUDE_L3_CACHE 88
INCLUDE_MMU_ARCH_PPC60X 93
supporting L2 and L3 cache 83
sysCacheLibInit 76
USER_D_MPU_ENABLE 90
USER_I_MPU_ENABLE 90
USER_L2_CACHE_ENABLE 88
printf() 46, 63
prjComps.h 125
prjParams.h 126
processor local bus 168
PROFILE_BOOTAPP 101
adding support for 102
CDF requirements 102
console port requirements 102
migrating boot loader customizations to 104
RAM image 103
setting RAM_LOW_ADRS and
RAM_HIGH_ADRS 103
PROFILE_SMALL_FOOTPRINT 194
providing architecture-specific cache and MMU support 82
PTE, see page table entry

Q
quiescent state 3, 64

R
RAM initialization 64
RAM_HIGH_ADRS 42, 103
setting in a CDF 103
RAM_LOW_ADRS 42, 103, 177
AMP 165, 180
modifications for small-footprint 192
setting in a CDF 103
README 20
real-time processes (RTPs) 213
reference entries 215
filename extensions 243
generating 243
processing 244
target.ref 228
tools 244
written style 216
refgen tool 244
REG_SET 137
removing a device driver from a VIP 184
required hardware support for SMP 121
required macros 41
see also macros, BSP macros
required routines 32
see also routines
returning
the physical CPU index value 136
the value of the logical CPU index 136
reusing code 212
ROM emulator 46
ROM monitor 46
ROM_AUTO_CACHE_DETECT 89
ROM_BASE_ADRS 43
ROM_LINK_ADRS 41
ROM_SIZE 42
ROM_TEXT_ADRS 41
ROM_WARM_ADRS 41, 43
ROM-based image type initialization 61
romCacheAutoDetect() 89
romInit() 5, 14, 22, 61, 74, 131
AMP 162
executing 14
romInit.s 22, 51
romInit() 22, 61
romStart() 15, 24, 61
routines 32
aimCacheInit() 77
aimMmuLibInit() 78
bootLineCrack() 31
bootLineParse() 159
cacheArchLibInit() 76
cacheClear() 134
cacheDisable() 134
cacheEnable() 134
cacheFlush() 134
cacheinvalidate() 134
cacheLibInit() 76
cacheLock() 135
cacheUnlock() 135
diagnostic 62
hardWareInterFaceInit() 70
implementing BSP routines for SMP 135
kernelCpuEnable() 128
kprintf() 146, 194
mmuADisable() 186
sysAmpCpuEnable() 158, 171, 172, 173, 176, 177
sysAmpCpuPrep() 158, 171, 173, 176, 177
sysBspRev() 33
sysClkConnect() 33
sysClkDisable() 33
sysClkEnable() 33
sysClkInit() 33
sysClkRateGet() 33
sysClkRateSet() 33
sysCpuAvailableGet() 139
sysCpuDisable() 139
sysCpuEnable() 130, 131, 133, 138
sysCpuInit() 131, 138, 139
sysHwInit() 5, 39, 60, 82, 83, 84, 137, 169
sysHwInit0() 5, 193
sysHwInit1() 5
sysHwInit2() 6, 18, 39, 60
sysInit() 74, 130, 162, 163, 165, 166
sysL2CacheInit() 83
sysL3CacheInit() 84
sysMemDescGet() 35
sysMemDescInit() 34
sysMemTop() 36, 80, 159, 178
sysMmuInit() 193
sysModel() 38
sysNvRamGet() 38
sysNvRamSet() 38
sysPhysMemTop() 178
sysProcNumGet() 159
sysProcNumSet() 159
sysSerialChanGet() 39
sysSerialHwInit() 39
sysSerialHwInit2() 39
sysToMonitor() 40, 74, 133, 139, 164, 169, 193
taskRegsInit() 137
timer driver 33
sysCpuInit() 139  
sysHwInit() 137  
sysToMonitor() 139  
using component and configuration preprocessor macros 125  
VxBus support 123  
WIND_CPU_STATE 137  

source files 19  
S-record files 68  
starting a secondary CPU 138  

structures  
PHYS_MEM_DESC 26  
REG_SET 137  
WIND_CPU_STATE 138  

support for DMA driver model 68  
supporting L2 and L3 cache 83  
SW_MMU_ENABLE 91  
sysAbsSymPhysMemBottom 166  
sysAbsSymPhysMemSize 166  
sysALib.s 22, 52, 135  
_sysInit() 22  
sysAmpCpuEnable() 171, 172, 173, 176, 177  
AMP 158  
sysAmpCpuPrep() 171, 172, 176, 177  
AMP 158  
sysAtalInit() 98  
sysBatDesc[] 83  
sysBootLine 26, 31  
sysBspDevFilter()  
AMP 182  
sysBspRev() 32, 33  
sysCache.c 83, 84  
sysCacheFlushReadArea 82  
sysCacheLibInit 76  
sysCacheLockALib.s 84  
sysCacheLockLib.c 84  
sysClkConnect() 32, 33  
sysClkDisable() 32, 33  
sysClkEnable() 32, 33  
sysClkInt() 32, 33  
sysClkRateGet() 32, 33  
sysClkRateSet() 32, 33  
sysComms.c  
AMP 166  
sysCpuArgsTable 138  
sysCpuAvailableGet() 139  
sysCpuDisable() 139  
sysCpuEnable() 130, 131, 133, 138  
sysCpuEnableFlag 138  
sysCpuInit() 131, 138, 139  
sysDev.c 23  
sysDma32MemDesc 25, 28  
sysDma32MemDescCount 26, 29  
sysExcMsg 26, 31  
sysFlags 26, 31  
sysFlashGet() 73  
sysFlashSet() 73  
sysHwInit() 5, 32, 39, 64, 82, 83, 84, 137  
AMP 169  

VxBus 60  
sysHwInit() 5, 32, 193  
sysHwInit1() 5, 32  
sysHwInit2() 6, 18, 32, 39  
VxBus 60  
sysIbcInit() 99  
sysInit() 15, 74, 130  
AMP 162, 163, 165, 166  
sysL2Cache.c 83  
sysL2CacheInit() 83, 99  
sysL3Cache.c 84  
sysL3CacheInit() 84  
sysLib.c 23, 64, 135  
AMP 165, 166  
modifications for small-footprint 192  
sysMemDescGet 201, 253  
sysMemDescGet() 32, 35  
sysMemDescInit 201, 252  
sysMemDescInit() 32, 34  
sysMemTop() 32, 36, 80  
AMP 159, 178  
sysMmuInit() 193  
sysModel() 32, 38  
sysNvRamGet() 32, 38, 73  
sysNvRamSet() 32, 38, 73  
sysPhysMemDesc 25  
sysPhysMemDesc[] 78, 165, 166, 193  
sysPhysMemDescNumEnt 25, 27, 80  
sysPhysMemTop() 37  
AMP 178  
sysPhysRamDesc 25, 27  
sysPhysRamDescCount 25, 28  
sysPmMemDesc 26, 30  
sysPmMemDescCount 26, 31  
sysProcNum 159  
sysProcNumGet()  
AMP 159  
sysProcNumSet()  
AMP 159  
sysSesInit() 99  
sysSerialChanGet() 32, 39  
sysSerialHwInit() 32, 39  
sysSerialHwInit2() 32, 39  
system-mode debugging techniques 66  
sysTffsInit() 99  
sysToMonitor() 32, 40, 74, 133, 139, 193  
AMP 164, 169  
sysUserMemDesc 26, 29  
sysUserMemDescCount 26, 30  

T  
target hardware 44  
target.nr  
running the mg2ref script 242  
target.ref 23, 96, 228  
TARGET_DIR 41
taskRegsInit() 137
terminology 3
TGT_DIR 41
ti omapl137evm 193
timer drivers 33
timers
  auxiliary clock driver 72
timestamp driver 72
TOOL 41
tools
debugging 211

U

USB bus interface 96
USER_D_CACHE_ENABLE 86
USER_D_CACHE_MODE 86
USER_D_MMU_ENABLE 90
USER_I_CACHE_ENABLE 86
USER_I_CACHE_MODE 86
USER_I_MMU_ENABLE 90
USER_L2_CACHE_ENABLE 88
USER_RESERVED_MEM 43
usrAppInit() 163
usrConfig.c 24, 31
  usrInit() 24, 64, 65
  usrRoot() 24, 65
usrInit() 16, 24, 64, 65
  AMP 166
exectuing 16
usrMmuInit() 80
usrRoot() 18, 24, 65, 174, 176
VxBus 60
usrToolsInit() 174, 176
usrWrloadInit() 174, 176
usrWrloadPeek() 174
utilities
  wrload 140

V

variables
  sysProcNum 159
VENDOR 41
verifying OS components in the image 113
VIP
  removing a device driver from 184
VM_ENABLE 193
VM_PAGE_SIZE 80, 89
VMA_START 43
vmBaseLib 77
VME bus interface 95
vmGlobalMap 80
vmGlobalMapInit() 80
vmLib 77, 192
vmLib.h 27
vmPageLock() 78
vmPageOptimize() 78
VSB
  enabling small-footprint testing 194
options
  COMPONENT_VXTESTV2_PSF 194
  PROFILE_SMALL_FOOTPRINT 194
  SFOOTPRINT_VXTESTV2 195
vsbConfig.h 125
WX_ENABLE_CPU_TIMEOUT 128
VX_SMP_NUM_CPUS 126, 127, 139
VX_TASK_INSTANTIATE() 195
vxbDeviceConnect() 70
vxbDeviceInit() 70
vxbInit() 70
VxBus
  adding devices 96
  BSP support for 95
  configuring for SMP 141
  debugging device drivers 97
  device driver registration 60
  hwconf.c 24
  initialization routines 70
  initialization sequence 70
  memory allocation for SMP 146
  MIPC 167
  support for SMP 123
vxCpuEnabled 139
vxCpuIdGet() 136, 137
vxCpuIndexGet() 136
vxCpuLib.h 136
vxCpuPhysIndexGet() 136
vxCpuStateInit() 137
vxTestBenchmark() 197
vxTestMdlFdGet() 197
vxTestParamGet() 197
vxTestUsDelay() 197
VXTESTV2_SF_DRIVER.Putchar 195
VXTESTV2_SF_DRIVER.Putchar.IS_RAW 195
vxTestWindBaseGet() 197
VxWorks
  image types 25
    bootrom 12
    bootrom_res 12
    bootrom_res_high 13
    bootrom_uncmp 12
    vxWorks 11
    vxWorks.res_rom 12
    vxWorks.res_rom_nosym 12
    vxWorks.res_rom_nosym.res_low 12
    vxWorks.res_rom.res_low 12
    vxWorks.st 12
    vxWorks.st_rom 12
    vxWorks.sym 11
    vxWorks_rom 11
  small-footprint configuration 189
vxWorks 11
VxWorks AMP, see AMP
VxWorks bus analyzer 48
VxWorks SMP, see SMP
vxWorks.res_rom 12
vxWorks.res_rom_nosym 12
vxWorks.res_rom_nosym.res_low 12
vxWorks.res_rom.res_low 12
vxWorks.st 12, 147
vxWorks.st_rom 12
vxWorks.sym 11
vxWorks_rom 11

W

WDB agent
debbuging with 48
starting the WDB agent before the kernel 66
Wind River MIPC, see MIPC
WIND_CPU_STATE 137, 138
Workbench 49
writing initialization code 51, 52
wrload utility 140, 158, 163
adding support for 170
components for AMP 153
providing sufficient system memory 179
wrload(), see wrload utility
WRLOAD_CPU_ENABLE_FUNC 177
WRLOAD_CPU_MAP_FUNC 175
WRLOAD_CPU_MEM_MAP_FUNC 174
WRLOAD_CPU_MEM_READ_FUNC 175
WRLOAD_CPU_MEM_UNMAP_FUNC 174
WRLOAD_CPU_PREPARE_FUNC 177
WRLOAD_CPU_UNMAP_FUNC 175
WRLOAD_ID_TYPE_GET 176
WRLOAD_ID_VALUE_GET 176
WRLOAD_IMAGE_BUILD_PHYS_BIAS 174, 175
wrloadCpuRtnSet() 173
wrloadSymValueGet() 173, 175

X

XScale
sysCacheLibInit 76